COST OPTIMIZED DATA ACCESS USING RANK JOIN

[1]Mr.Pravin L Satore, [2]Mrs.Shital Y Gaikwad

[1]P G Student & Researcher MPGI SOE, [2]Assistant Professor
[1]Computer Science & Engineering, [2]Computer Science & Engineering
[1] MPGI SOE, Nanded, India, [2] MPGI SOE, Nanded, India

Abstract: The prime task of search computing is to join the results of complex query plans. In this survey we describe further views on topic by emphasizing the study on algorithms that operate with joining the ranked results produced by search services. This type of problem is categorized in the conventional rank aggregation, i.e., combining different ranked lists of objects to produce a single valid ranking. Rank-join algorithms provide best overall results without accessing total objects in ranked list. The rank-join problem has been dealt by extending rank aggregation algorithms to the case of join in the setting of relational databases. However, previous methods of top-k queries did not consider the distinctive features of search engines on the Web. However, in the context of search computing, joins differs from conventional relational concepts in many ways. Different access patterns are used to access the services i.e. some inputs needs to be provided; accessing services is costly, because usually they are remote. The output is returned in pages of answers and normally by some ranking criterion; multiple search services to answer the same query, in that user can interact to refine the search criteria. This survey reviews the challenges that are very important in the context of designing rank join algorithms for efficiency of search computing.

***IndexTerms* – top-k, rank-join, query optimization, cost optimization, TA algorithm.**

I. INTRODUCTION

Search services uses different types of techniques to rank query answers. Most of the time, users are only looking for most important query answers, i.e. top-k answers, from bulk of answers. Currently many emerging applications assure that the effective support for top-k queries is there. For example, the success rate of meta-search engines is directly proportional to the use of effective rank aggregation methods. The next challenge is to firmly combine the ranked list of objects and create a single unanimous ranking for the objects. Many applications answer the queries that have to join and aggregate the multiple inputs to give the top-k results.

We will concentrate on a special kind of top-k processing techniques, i.e. rank join algorithms, which fetches the top-k combinations from a data set that comes from joining multiple data sources. These kind of top-k processing techniques are very significant for answering multi-domain queries. This involves the answers to be extracted and combined from domain specific search system. Finally, a aggregation function used to form a global ranking for every combined answers, so as to provide the answer to user having the top combined scores.

The data set produces the output tuples sorted by some score, in this context the score is definite field of tuples. The ranked list may consist of large number of items represented in pages and cost of accessing these kinds of pages is sometimes becomes intolerable. The objects in the list are accessed by some methods like sorted i.e. resulting a large list of objects ranked by some function, or random i.e. resulting a limited set of objects, not ranked but some condition over attribute is satisfied.

II. ISSUES OF SEARCH COMPUTING

Search computing concentrates on answering complex search queries combining data from several multidomain search services on web or other platform. These combinations are ranked and joined by some score attached to them. Every combination has a score, usually computed by some aggregation function over scores of every data elements. Mostly users only browse the top answers sorted by score. A simple but effective way is, first fetching the data elements from the data sets, second answers are joined to form the combinations, third compute the score of every combinations and finally ordering the combinations by their scores. The basic concepts of rank join algorithms are to explore situation where the input data sets, i.e. relational data table, are already sorted by some score. That's why a solution can be proposed to avoid the retrieving of tuples because the top-k combinations of answers can already be formed. The major task then of the conventional rank join algorithms is to optimize or minimize the input/output cost with respect to extremely simple join and sort approach discussed earlier.

Rank join algorithms are very important aspects of search computing along with that we also need to analyze the individual characteristics search computing especially when operating data sets are dynamic search services rather than relational database tables. In next section we will see a overview of core concepts that elaborate search computing.

2.1Access Patterns

Search services have some limitation like input to some fields is compulsory to give to get the results. To deal with this kind of restriction, we consider that each service is characterized by a given number of combinations of input and output parameters, called *access patterns*, pointing it to different ways that it can be retrieved. Access pattern is not handled by search computing framework therefore search computing must define a access strategy for its requirements.

2.2Access Costs

Extraction of data is costly and should be handled by effective rank join algorithms i.e. fetching data must be cost optimized. Access cost is partially depends on invoked services and applied access methods. Random and sorted access methods are available methods in context of rank join problem.

Sorted access method retrieves tuples that are sorted by some score and result is open to all search services, but for every new call results in a page of data elements instead of a tuple. In some situation, only one ranking criteria might be present for sorted access to search service. For example, consider a query that intends to find hotels by its stars, like 3 stars or 5 stars, but instead available service can only retrieve the results ordered by its nearest location from airport. In such situation if relation between the stars and location can be mapped then rank join algorithms might find themselves useful to answer such queries.

Random access extracts tuples that relates to a given object, e.g. all the Chinese cuisine restaurants in the city of Mumbai and permits to terminate rank join algorithm early when it is available, and thus narrowing the number of input/output operation. In a relational setting, random access can be provided by building an index on top of one of the attributes of a table. This is not a suitable option when operating in search computing. However, when a search service, say s_1,only provides sorted access, it is to some extent possible to obtain random access by invoking another search service, say s_2,returning data items of the same kind, although s_2 might be characterized by a different access cost and contain only a subset of the data items of s_1.

2.3Redundancy of Data Sources

There may be different search services which can be potentially invoked to answer identical or similar queries. Consider, for example, the excessive number of identical services that search for movies. Such a redundant availability of data sources comes with no additional cost in the context of search computing. If correctly managed, it can be positively used in two ways: one, improving the system response time and second, the robustness to time-varying access costs or service failures. Regarding first goal, parallel invocation of multiple services can be implemented. The availability of multidomain search services, each referred by its access cost, might provide random access when single service is not able to do so.

2.4Users in the Loop

The queries proposed by users of a search computing system can be shaped at runtime to help satisfying the user's information requirement. The liquid queries paradigm concept portrays a set of operations that can be performed at the client side. Some of these operations do not require communication with the remote search services, since they only influence the visualization of data already available at the client side. Others require extraction of additional data from remote services. For example, the user might want to dynamically adjust the aggregation function. In a weighted sum, this is accomplished by changing the weights assigned to the different search services. In order to preserve the guarantee of displaying the top results, further data might need to be fetched.

III. RANK JOIN ALGORITHM FOR TOP K QUERY PROCESSING

Now we will review the main algorithms that were designed for the rank join problem. We also discuss some top-k processing techniques that inspired many rank join algorithms. In the following section, we introduce taxonomy to classify top-k techniques based on two design dimensions.

3.1Data Access

Many top-k processing techniques involve accessing multiple data sources with different cost estimation of the underlying data objects. The ways in which these sources are accessed have large influence on the design of the underlying top-k processing techniques. For example, ranked lists of objects could be scanned in score order. We call this access method as sorted access. On the other side, the score of some object might be required directly without traversing the objects with higher/smaller scores. We call this access method as random access. If an index is built on object keys the random access could be provided through index lookup operations.

The taxonomy of top-k processing techniques based on the data access methods in the underlying data sources, as follows:

Both Sorted and Random Access: In this category, top-k processing techniques assume the availability of both sorted and random access in all the underlying data sources.

No Random Access: In this type, top-k processing techniques assume that data sources provide only sorted access to objects based on their scores.

Sorted Access with Controlled Random Probes: In this category, top-k processing techniques assume the availability of at least one sorted access source. Random accesses are used in a controlled manner to reveal the overall scores of candidate answers.

3.1.1Both Sorted and Random Access

Top-k processing techniques in this category assume data sources that support both access methods, sorted and random. The Threshold Algorithm (TA) and Combined Algorithm (CA) [3] belong to this category.

Algorithm 1 explains the details of TA. The algorithm scans multiple lists, showing different rankings of the same set of objects. An upper bound *T* is maintained for the overall score of unseen objects. The upper bound is computed by applying the scoring function to the partial scores of the last seen objects in different lists. Each newly seen object in one of the lists is looked up in all other lists, and its scores are aggregated using the some scoring function to get the overall score. All objects with total scores that are greater than or equal to *T* can be reported. The algorithm halts after returning the K^{th} output.

ALGORITHM 1: **Threshold Algorithm (TA)** [3]

1. Do sorted access in parallel to each of the *m* sorted lists L_i. As a new object *o* is seen under sorted access in some list, do random access to the other lists to find $p_i(o)$ *in* every other list L_i. Compute the score $F(o)= F(p_1,\ldots, p_m)$ of object *o* . If this score is among the *k* highest scores seen so far, then remember object o and its score *F(o)*.
2. For each list L_i, let p_i be the score of the last object seen under sorted access. Define the threshold value *T* to be $F(\bar{p}_1,\ldots, \bar{p}_m)$. As soon as at least *k* objects have been seen with scores at least equal to *T* , halt.
3. Let A_k be a set containing the *k* seen objects with the highest scores. The output is the sorted set $\{(o, F(o))|o\in A_k\}$.

TA algorithm assumes that the costs of different access methods are the same; the CA algorithm [3] on the other hand assumes that the costs of different access methods are different. The CA algorithm defines a ratio between the costs of the two access methods to control the number of random accesses, since random access has higher costs than sorted accesses. The CA algorithm regularly performs random accesses to collect unknown partial scores for objects with the highest score lower bounds. A score lower bound is computed by applying the scoring function to object's known partial scores and worst known score. On the other hand, a score upper bound is computed by applying the scoring function to object's known partial scores, and the *best* possible unknown partial scores.

3.1.2No Random Access

The techniques we discuss in this section assume random access is not supported by the underlying sources. The Rank-Join algorithm [6] and the *J** algorithm [11] are examples of the techniques that belong to this category.

The Rank-Join algorithm [6] integrates the ranking and joining tasks in one efficient operator. Algorithm 2 describes the main Rank-Join procedure. The Rank-Join algorithm scans input lists (the joined relations) in the order of their scoring predicates. Join results are discovered incrementally as the algorithm moves down the ranked input relations. For each join result *j* , the algorithm computes a score for *j* using a score aggregation function F. The algorithm maintains a threshold T bounding the scores of join results that are not discovered yet. The top-k join results are obtained when the minimum score of the k join results with the maximum *F*() values is not below the threshold *T*.

ALGORITHM 2: **Rank join Algorithm** [2]

1. Retrieve tuples from input relations in descending order of their individual scores p_i's. For each new retrieved tuple *t*:
   i) Generate new valid join combinations between *t* and seen tuples in other relations.
   ii) For each resulting join combination *j*, compute *F*(*j*).
   iii) Let p_i max be the top score in relation *i*, i.e., the score of the first tuple retrieved from relation *i*. Let $\bar{p}_i$ be the last seen score in relation *i*. Let *T* be the maximum of the following *m* values:
   $F(\bar{p}_1, p_2^{max},\ldots, p_m^{max})$,
   $F(p_1^{max}, \bar{p}_2,\ldots, p_m^{max})$,
   ….
   $F(p_1^{max}, p_2^{max},\ldots, \bar{p}_m)$.
   iv) Let A_k be a set of *k* join results with the maximum *F*(.) values, and M_k be the lowest score in A_k. Halt when $M_k \geq T$.
2. Report the join results in A_k ordered on their *F*(.) values.

A two-way hash join implementation of the Rank-Join algorithm, called *Hash Rank Join Operator* (HRJN), is introduced in [6]. HRJN is based on symmetrical hash join. The operator keeps a hash table for each relation involved in the process of join, and a priority queue is also maintained to buffer the join results in the order of their scores. The hash tables hold input tuples seen so far and are used to compute the valid join results. The HRJN operator implements the traditional iterator interface of query operators, which include two methods: *Open* and *GetNext*. The *Open* method is responsible for initializing the necessary data structure; the priority queue *Q*, and the left and right hash tables.

The *GetNext* method remembers the two top scores, p_1^{max} and p_2^{max} , and the last seen scores, $\bar{p}_1$ and $\bar{p}_2$ of its left and right inputs. At any time during query execution, the threshold *T* is computed as the maximum of $F(p_1^{max}, \bar{p}_2)$ and $F(\bar{p}_1, p_2^{max})$. At each step, the algorithm reads tuples from either the right or left inputs, and probes the hash table of the other input to generate join results. The algorithm decides which input to poll at each step, which gives flexibility to optimize the operator for fast generation of join results based on the joined data. A simple strategy is accessing the inputs in a round-robin fashion. A join result is reported if its score is not below the scores of all discovered join results, and the threshold *T*.

In [12], an enhancement of HRJN algorithm is provided where a different bounding scheme is used to compute the threshold *T*. This is achieved by computing a *feasible region* in which unseen objects may exist. Feasible region is computed based on the objects seen so far, and knowing the possible range of score predicates. The algorithm reports the next top join result as soon as the join result at queue top includes an object from each ranked input.

3.1.3Sorted Access with Controlled Random Probe

Top-k processing methods in this category assume that at least one source provides sorted access, while random accesses are performed only when needed. The Upper and Pick algorithms [2, 10] are examples of these methods.

Upper and Pick assume that each source can provide a sorted and/or random access to its ranked input, and that at least one source supports sorted access. The main purpose of having at least one sorted-access source is to obtain an initial set of candidate objects. In the Upper algorithm, candidate objects are retrieved first from sorted sources, and inserted into a priority queue based on their score upper bounds. The upper bound of unseen objects is updated when new objects are retrieved from sorted sources.

3.2Implementation Level

Integrating top-k processing with database systems is addressed in different ways by current techniques. One approach is to embed top-k processing in an outer layer on-top of the database engine. The capabilities of database engines, like storage,

indexing and query processing, are leveraged to allow for efficient top-k processing.

3.2.1Application Level

Top-k query processing techniques that are implemented at the application level, without major modification to the underlying database system. We divide application level top-k methods into Filter-Restart methods and Indexes/Materialized Views methods.

Filter-Restart techniques formulate top-k queries as range selection queries to limit the number of retrieved objects. That is, a top-k query that ranks objects based on a scoring function F, defined on a set of scoring predicates $p_1,...,p_m$, is formulated as a range query of the form find objects with $p_1 > T_1$, *and*,... , $p_m > T_m$, where T_i is an estimated cutoff threshold for predicate p_i. Using a range query aims at limiting the retrieved set of objects to the necessary objects to answer the top-k query. Incorrect estimation of cut-off threshold yields one of two possibilities: one, if the cutoff is overestimated, the retrieved objects may not be sufficient to answer the top-k query and the range query has to be restarted with looser thresholds, or two, if the cutoff is underestimated, the number of fetched objects will be more than necessary to answer the top-k query.

Materialized views have been studied in the context of top-k processing as a means to provide efficient access to scoring and ordering information that is expensive to gather during query execution. Using materialized views for top-k processing has been studied in the PREFER system [4, 5]. The score of a certain tuple is captured by an arbitrary weighted summation of the scoring predicates. The proposed method keeps a number of materialized views based on different weight assignments of the scoring predicates. The best view is the one with the least number of tuples to be fetched.

3.2.2Engine Level

The main theme of these techniques is their tight coupling with the query engine. This tight coupling has been realized through multiple approaches. Some approaches focus on design of efficient specialized rank-aware query operators. A third category addresses modifying query optimizers, e.g., changing optimizers plan enumeration and cost estimation procedures.

Query optimizers need to be able to enumerate and cost plans with rank-aware operators as well as conventional query operators. The size of the consumed input depends on the operator implementation rather than the input itself. A probabilistic model has been proposed in [8] to estimate the Rank-Join inputs' depths, i.e., how many tuples are consumed from each input to produce the top-k join results.

IV. QUERY AND COST OPTIMIZATION

In this section we analyze several optimization requirements and opportunities in the context of rank-join algorithms for search computing. We will see cost models used for optimization and, finally discuss proper adjustments of strategies when the actual statistics extracted during the execution of rank join differ from those estimated in advanced.

4.1Role of Query Optimization in Search Computing

One of the important tasks of rank join in search computing is to obtain query results quickly by limiting the number of requests pass to search services to extract data and create combinations ensuring that the combinations are top-k. We can characterize such a successful execution of a rank-join between two services, say s_1 and s_2, with a descent retrieving n_1 and, respectively, n_2 tuples. In this context, we can express the expected number of formed combinations $K(n_1, n_2)$ and the expected cost $C(n_1, n_2)$ incurred by such a descent as functions over n_1 and n_2. More specifically, $K(n_1, n_2)$ will relate the numbers of tuples extracted with available information about the join selectivity, the distribution of values in the data returned by s_1 and s_2, and, possibly, the score distributions in s_1 and s_2 as well as the aggregation function used to compose the individual scores; similarly, $C(n_1, n_2)$ will take into account the costs of accessing s_1 and s_2. Result of this, the optimization problem at hand may be formulated as follows:

$$\begin{aligned} &\text{Minimize} && C(n_1, n_2) \\ &\text{subject to} && K(n_1, n_2) \geq k \qquad (1) \\ & && n_1 \in \mathrm{N} \cap [0, \mathrm{N}_1],\ n_2 \in \mathrm{N} \cap [0, \mathrm{N}_2] \end{aligned}$$

Where N is the set of natural numbers, k is the desired number of top combinations, and N_i is the maximum number of tuples that can be output by s_i for i=1, 2. A solution to the above optimization problem will provide estimates for n_1 and n_2 and, as a result, an estimate of the cost incurred by the rank-join operation.

The user can request a "query re-ranking", i.e., an update of the results of the query by modifying the aggregation function, e.g., by changing the weights of the individual scores when the aggregated score is expressed as a weighted sum thereof. Such an operation is not equivalent to a simple rearrangement of the combinations obtained with the previous aggregation function, but generally requires computing a new and different set of combinations, possibly by deepening the previous descent to the services. Instead of limiting ourselves to computing a set of combinations that includes the top k ones with respect to a given aggregation function; it might be preferable to compute a larger set that includes the top k combinations no matter what the aggregation function is. When both sorted and random accesses are available, this behavior may be obtained, e.g., by adapting to the rank-join case the original rank aggregation algorithm by Fagin's algorithm [14].

4.2Cost Models for Query Optimization

In order to solve the query optimization problem formulated above, which refers to a single rank-join operator on two services, we need to define a cost model that characterizes the cost function $C(n_1, n_2)$. Most of the previous literature on rank-join adopts a simple additive model, whereby the cost is defined as the sum of the costs of all I/O operations. Both sorted and random access (whenever available) costs need to be taken into account, since they are possibly characterized by heterogeneous costs, due to the fact that random accesses might potentially refer to data that is stored in other external data sources. While this approach is still applicable in the context of search computing, we want to take advantage of the fact that services are typically available at remote servers.

4.3Need of Adaptive Algorithms

Before query execution starts, the optimizer uses the available information collected about the search services to be joined, in order to devise: i) a cost model, as explained earlier, which defines the number of tuples to be fetched; ii) the rank-join execution strategy, also known pulling strategy, which determines the optimal order of service invocations during the descent in the two services. It might be the case that statistics collected in advanced, e.g. join selectivity, score distributions, etc., do not match the actual extracted data. This is specifically relevant in the context of search computing, where the actual statistics depends on the query prototype and specific user provided keywords.

A different kind of adaptive execution might arise when considering complex queries in the large, i.e. in the context of a global query optimization framework. We refer to this feature as inter-operator adaptive execution. In this scenario, adaptive execution might be necessary in order to cope with time-varying availability of search services. During query execution, search services might stop responding to invocations and alternative execution strategies might be devised on-the-fly, possibly preserving the data already fetched up to that point in time.

V. CONCLUSION

In this survey we have defined and notified role of rank join in search computing. New challenges for rank join algorithm related to adaptability and new cost models are described very precisely. We also mentioned the integration of rank join in query optimization framework. By properly applying rank join algorithms within the context of data access methods into the framework of search computing the cost can be minimized or optimized.

VI. ACKNOWLEDGMENT

Authors are thankful to The Director, Sanmati Engineering College, Washim and The publisher of the journal for arranging the these national conference. Also, I am great thankful to Director, Dean, HOD, Guide from MPGI SOE, Nanded and finally The Principal, Government Polytechnic, Hingoli for moral support in research and publication of these article.

REFERENCES

[1] Bruno, N., Chaudhuri, S., Gravano, L.: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM Transactions on Database Systems 27(2), 153–187 (2002)

[2] Bruno, N., Gravano, L., Marian, A.: Evaluating Top-k Queries over Web-Accessible Databases. In: Proceedings of ICDE 2002, pp. 369–378 (2002)

[3] Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. Journal of Computer and System Science 1(1), 614–656 (2001)

[4] Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In: Proceedings of ACM SIGMOD 2001, pp. 259–270 (2001)

[5] Hristidis, V., Papakonstantinou, Y.: Algorithms and applications for answering ranked queries using ranked views. VLDB Journal 13(1), 49–70 (2004)

[6] Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. VLDB Journal 13(3), 207–221 (2004)

[7] Ilyas, I.F., Aref, W.G., Elmagarmid, A.K., Elmongui, H.G., Shah, R., Vitter, J.S.: Adaptive rank-aware query optimization in relational databases. ACM Transactions on Database Systems 31(4), 1257–1304 (2006)

[8] Ilyas, I.F., Shah, R., Aref, W.G., Vitter, J.S., Elmagarmid, A.K.: Rank-aware query optimization. In: Proceedings of ACM SIGMOD 2004, pp. 203–214 (2004)

[9] Li, C., Chang, K.C.-C., Ilyas, I.F., Song, S.: RankSQL: query algebra and optimization for relational top-k queries. In: Proceedings of ACM SIGMOD 2005, pp. 131–142 (2005)

[10] Marian, A., Bruno, N., Gravano, L.: Evaluating top-k queries over web-accessible databases. ACM Transactions on Database Systems 29(2), 319–362 (2004)

[11] Natsev, A., Chang, Y.-C., Smith, J.R., Li, C.-S., Vitter, J.S.: Supporting Incremental Join Queries on Ranked Inputs. In: Proceedings of VLDB 2001, pp. 281–290 (2001)

[12] Schnaitter, K., Polyzotis, N.: Evaluating rank joins with optimal cost. In: Proceedings of PODS 2008, pp. 43–52 (2008)

[13] Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-query processing techniques in relational database systems. ACM Comput. Surv. 40(4) (2008)

[14] Fagin, R.: Combining Fuzzy Information from Multiple Systems. J. Comput. Syst. Sci. 58(1), 83–99 (1999)