Processing of Rank Joins in Highly Distributed Systems

Christos Doulkeridis[1], Akrivi Vlachou[1], Kjetil Nørvåg[1], Yannis Kotidis[2] and Neoklis Polyzotis[3]

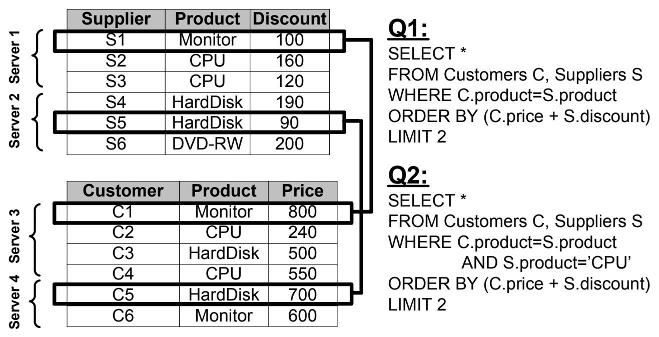
[1]*Norwegian University of Science and Technology (NTNU), Trondheim, Norway*
[2]*Athens University of Economics and Business (AUEB), Athens, Greece*
[3]*UC Santa Cruz, California, USA*
{cdoulk,vlachou,noervaag}@idi.ntnu.no, kotidis@aueb.gr, alkis@cs.ucsc.edu

***Abstract*—In this paper, we study efficient processing of rank joins in highly distributed systems, where servers store fragments of relations in an autonomous manner. Existing rank join algorithms exhibit poor performance in this setting due to excessive communication costs or high latency. We propose a novel distributed rank-join framework that determines the subset of each relational fragment that needs to be fetched to generate the top-k join results. At the heart of our framework lies a score bound estimation algorithm that utilizes statistics to produce sufficient score bounds for each relation, which guarantee the correctness of the rank join result set. Furthermore, we propose a generalization of our framework that supports approximate statistics, in the case that the exact statistical information is not available. In addition, we demonstrate how our algorithms can utilize distributed statistics to process top-k join queries without any modifications. An extensive experimental study validates the efficiency of our framework and demonstrates its advantages over existing methods.**

I. Introduction and Motivation

Rank-aware query processing has attracted much interest lately, as users are often overwhelmed by the size of query results. Ranked queries help users to identify a limited set of the most interesting results, thereby enabling effective decision-making. From the aspect of the database system, the challenge associated with ranked queries is to efficiently process the query by inspecting carefully selected tuples, without examining all stored data exhaustively.

As data management becomes inherently distributed due to massive content generation and storage at disparate locations, the importance of distributed processing of rank-aware queries is even more evident. In this paper, we consider a generic distributed setup where servers store relational fragments individually, resembling horizontal partitioning of relations. Contemporary applications increasingly adopt this storage model when deployed either on widely distributed networks or on cloud computing platforms. We focus on the evaluation of distributed *rank join* queries. In general, such queries join m relations R_i that may be fragmented into several parts stored at different servers. The resulting tuples are ordered using a scoring function $f()$ (*order by* clause) and the top-k answers based on their scores are returned to the user (*limit* clause). Rank join queries adhere to the following template, where the relations R_i are widely distributed to different servers:



Supplier	Product	Discount
S1	Monitor	100
S2	CPU	160
S3	CPU	120
S4	HardDisk	190
S5	HardDisk	90
S6	DVD-RW	200

Customer	Product	Price
C1	Monitor	800
C2	CPU	240
C3	HardDisk	500
C4	CPU	550
C5	HardDisk	700
C6	Monitor	600

```
Q1:
SELECT *
FROM Customers C, Suppliers S
WHERE C.product=S.product
ORDER BY (C.price + S.discount)
LIMIT 2

Q2:
SELECT *
FROM Customers C, Suppliers S
WHERE C.product=S.product
        AND S.product='CPU'
ORDER BY (C.price + S.discount)
LIMIT 2
```

Fig. 1. Example of distributed top-k join queries.

SELECT some attributes
FROM R_1,R_2,...,R_m
WHERE join condition AND selection predicates
ORDER BY $f(R_1.s_1, R_2.s_2, ..., R_m.s_m)$
LIMIT k

Applications of a distributed rank join query can be derived from the scientific domain. For example, astronomers tend to have a distributed measurement infrastructure, in that different telescopes may point to the same area of the sky and collect information about a set of metrics (attributes). We can imagine each server storing information about different "slices" of the sky with corresponding measurements (tuples). Then, a query may join the relations of different telescopes using the slice identifier as the join key, so as to assemble together different measurements corresponding to the same positions in the sky and rank the results according to a user-specified function.

Another application is decision-making in an international company that maintains geographically dispersed departments over the world. Departments include, among others, production, inventory, sales, etc. Each department (server) produces large amounts of data on a daily basis; for instance information about customer orders is recorded at sales departments. Decision-making over widely distributed enterprise data can be facilitated by efficient support for top-k join queries, for example retrieval of the top-k opportunities for product sale. Consider a *Suppliers* relation and a *Customers* relation, which are fragmented in an horizontal manner over the servers (multiple inventory and sales departments respectively) at different locations around the globe. If all data were available in a traditional centralized database, it would be depicted as in Fig. 1, where a bracket shows the server that stores a subset of tuples. A product is sold to a customer at a certain price, as shown at the *Customers* table. The company buys these products from suppliers, as depicted in the *Suppliers* table. A sale maximizes the associated profit, if the amount paid by a customer plus the discount offered by a supplier is maximized. In this scenario, the sales manager is interested in finding the 2 most profitable sales for any product (Query $Q1$). Similarly, in Query $Q2$ the manager is only interested in sales of a specific product (*CPU* in this example). Our algorithms support both queries and we will use them as examples in the paper.

In this paper, we propose a framework for rank join query processing over widely distributed data, which is a challenging problem. For instance, one major difference compared to centralized rank join processing is that tuples (or fixed-size blocks of tuples) cannot be iteratively read from each relation. This results in several communication round-trips in the network, which is prohibitively expensive even for moderate number of servers. Moreover, the number of tuples that need to be fetched and joined to compute the correct result, can be significantly more than the value of k requested by the user. Thus, our premise is to compute a bound on the scores of tuples (*score bound*), which restricts the tuples that need to be retrieved from each relation, and then contact once each server that stores tuples within the bound to fetch the relevant tuples.

In existing approaches for distributed top-k queries, such as Fagin's algorithm [1] or its improvements (e.g. KLEE [2]), the joining attribute is a unique identifier present in all relations. In contrast, we address a generalization of this problem, focusing on arbitrary user-defined join attributes between relations. The state-of-the-art algorithms for this problem [3]–[5] rely on local sampling for score bound estimation, which is ineffective in the general case of non-uniformly distributed data. In summary, the contribution of our work is manifold:

- We provide the first thorough investigation of rank join query processing in large-scale distributed systems. To this end, we propose the *DRJN* (Distributed Rank JoiN) framework for efficient processing of rank join queries over horizontally fragmented data (Section III).
- We propose a novel score bound estimation algorithm that derives sufficient bounds on the scores of tuples for each relation participating in the query. The algorithm exploits statistics in order to produce score bounds that guarantee the generation of the correct result set (Section IV).
- We generalize the basic *DRJN* framework to work with approximate statistics, in the case that the statistics do not accurately describe the underlying data (Section V). We show that the generalized *DRJN* framework still produces the correct result at the cost of more than a single phase of score bound estimation, and supports a wide variety of complex queries (Section VII).
- We demonstrate the suitability of our framework without any modifications, for applications that require distributed storage of statistics over the servers (Section VI).
- We show that our algorithm consistently outperforms the state-of-the-art algorithms [3]–[5] (Section VIII).

II. Problem Statement

The system consists of N_S servers, where each server S_i stores fragments of one or more relations R_j. The fragment of R_j stored at S_i contains a subset of the tuples that belongs to R_j. In the following, we assume that we have m relations R_i $i \in [0 \dots m)$. Each relation R_i has a scoring attribute s_i and a join attribute a_i. We emphasize that the join relationship is many-to-many and our framework supports more than one join and scoring attributes for each relation, but we only use one in our discussion in order to keep the notation simple. In case of multiple join or scoring attributes, our techniques are applicable assuming that the necessary statistics (described by more dimensions) are available. We also stress that servers store relation fragments in an autonomous manner.

Our framework supports top-k join queries (such as Query $Q1$ or $Q2$) posed by any querying server S_i, henceforth mentioned as S_Q, which is responsible for query processing. Each query may refer to a different subset of the m relations R_i. Also, each query combines scoring attributes from different relations by means of a scoring function f that is monotone. The restriction of monotonicity is a common property [1] and it conveys the meaning that whenever the score of all attributes of a tuple τ_1 is at least as good as another tuple τ_2, then we expect that the overall score of τ_1 is as least good as τ_2. The result of a top-k join query is the ranked list of the k objects with best score values. Without loss of generality, in this paper we are interested in retrieving the k objects with the minimum score values, therefore lower scores indicate better rank.

Naive algorithms. One plain solution is to follow a *data shipping* approach [6], by having S_Q assemble locally all fragments of the relations R_i that participate in the query, and then process the rank join in a centralized way. However, the incurred network cost is high, as the complete relations R_i are transferred to S_Q, whereas only few tuples are typically necessary for producing the top-k results. Another naive algorithm is for S_Q to repetitively contact all servers that store fragments of the relations R_i, retrieve batch of tuples, and process them using any centralized rank join algorithm [7]–[9], thus reducing communication costs. However, even if knowledge of which servers store fragments of R_i was available to S_Q (which is not trivial in large-scale distributed systems), the total induced latency is high due to multiple communication round-trips.

III. The DRJN Framework

In the aforementioned setup, the challenge is to process rank join queries in an efficient way, such that the required communication cost and the round trips are minimized. In order to achieve these goals, it is necessary to determine the tuples that should be retrieved from each relation fragment. The *DRJN* framework utilizes statistics in the form of histograms to derive sufficient *score bounds* that produce the correct top-k join results. The bounds define the (range of score values of) tuples that need to be retrieved from the relation fragments. Fig. 2 illustrates the notion of score bounds for a (centralized) rank join algorithm, where tuples are sorted based on score,

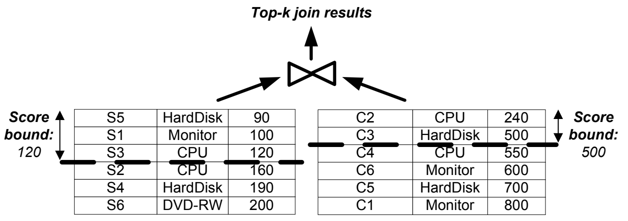


Fig. 2. Example of score bounds.

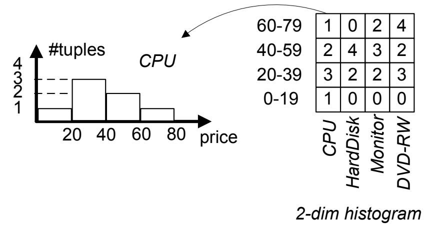


Fig. 3. Example histogram of relation *Suppliers*.

Algorithm 1 The $DRJN$ algorithm.

```
1: Input: $k$, Function $f$, $m$ relations $R_i$
2: Output: Ranked join result $res$
3: $tuples_{R_i} \leftarrow \emptyset$, $0 \leq i < m$
4: $\{(e_i, L_i), \gamma_k\} \leftarrow$ BoundEstimation($\{R_i | i \in [0, m)\}$, $k$, $f$)
5: for ($R_i \in [R_0 \ldots R_m]$) do
6:     for ($S_j \in L_i$) do
7:         $tuples_{R_i} \leftarrow tuples_{R_i} + S_j$.getTuples($e_i$)
8:     end for
9: end for
10: $res \leftarrow$ RankJoin($\{tuples_{R_0}\}, \ldots, \{tuples_{R_{m-1}}\}$)
11: return $res$
```

and the bound determines the tuples that need to be accessed. The exact way of storage of the histograms is orthogonal to our framework, but for simplicity in the following we will assume a single master server S_M that is responsible for collecting and storing the statistics. As will be demonstrated in Section VI, the $DRJN$ framework is applicable without modifications when the statistics are collaboratively stored by the participating servers in a distributed fashion.

Algorithm 1 provides the pseudocode for distributed rank join query processing on S_Q. Initially, S_Q computes a score bound for each participating relation using *BoundEstimation()* (line 4). Together with the bound e_i for each relation R_i, the function returns the list L_i of servers that store the tuples specified by the bound and the estimated score γ_k of the k-th join result. Then, for a relation R_i, S_Q issues a $getTuples()$ request to each server in L_i using bound e_i to restrict the tuples that need to be fetched, and assembles all relevant tuples (lines 6-8). Actually, all tuples τ of R_i that have a scoring value $\tau.s_i$ smaller or equal to e_i ($\tau.s_i \leq e_i$) are retrieved in a batch. When all necessary tuples are retrieved, S_Q performs a local centralized rank join algorithm [7]–[9] (line 10) and is able to report the result (res) (line 11). Obviously, the main challenge related to this algorithm is the procedure used for estimating appropriate bounds for the scoring values.

IV. Distributed Bound Estimation

In this section, we present the *distributed score bound estimation* algorithm, which derives score bounds e_i of relations R_i based on the score distribution captured in histograms. Notice that existing techniques for centralized rank join processing, e.g. [10], typically employ data synopsis such as histograms for capturing the score distribution.

A. Data Model and Statistics

For each relation R_i, a histogram H^{R_i} is maintained. This is a 2-dimensional histogram on (a_i, s_i) that essentially records the number of tuples in R_i that correspond to each distinct value v_z of a_i, that fall in a range of values of s_i (defined by the bins' low and high value). Thus, as far as the join attributes are concerned, we assume that each bin represents only one join value (we drop this restriction later in Section V). Let n denote the number of distinct values v_z ($0 \leq z < n$) of the join attribute. For each distinct value v_z, the set of bins $H^{R_i}_{v_z}$ can be viewed as a one-dimensional histogram that approximates the distribution of scoring values for v_z. For example, a histogram of relation *Suppliers* and join attribute *product* equal to *'CPU'* captures the number of products of type CPU for different ranges of *price* (scoring attribute), as depicted in Fig. 3. Furthermore, in our examples and implementation, equi-width histograms are employed, even though better performance can be achieved with optimized histograms [11], [12].

B. Joining Histograms

Differently to traditional rank-join algorithms, our algorithm performs a rank-join on histogram bins instead of tuples. Histogram bins are accessed sorted in ascending order of their score range. Moreover, histograms bins of different relations (also mentioned as *individual bins*) are joined and produce join combinations of bins. A valid join combination of bins is produced by a set of bins with the same value of the join attribute. Henceforth, this new bin is referred to as *joined bin*. For each joined bin, the number of tuples is computed by multiplying the number of tuples in the individual bins. Furthermore, the score of a joined bin is estimated by applying the scoring function f on the higher value of score range of each individual bin. Thus, the score of the joined bin is an upper bound of the score of any join tuple produced by the individual bins.

Example 1: (Joining histograms) Consider again the example of Fig. 1 and assume that the histograms depicted in Fig. 4 capture the data distribution of relations R_0=Suppliers *and* R_1=Customers *respectively. Each bin of a histogram represents the number of suppliers (resp. customers) that handle (resp. request) a particular product for the range of prices specified by the boundaries of the bin. On the right, some joined bins are depicted for product 'CPU' when the scoring function f is the sum of discount and price. For instance, the combination of the first two bins of each histogram for 'CPU', which contain 6 and 10 tuples respectively, produce a joined bin of 60 (=6 × 10) tuples with score in the range 0-28. The*

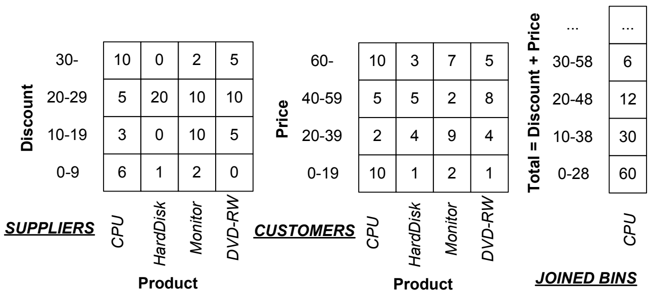


Fig. 4. Example of joining histograms.

score of the tuples in this joined bin is set to 28, i.e., the high value of the range. Recall that the aim is to retrieve the top-k results with minimum scores.

The objective of our algorithm is twofold: first, to identify the histogram bins that produce at least k join tuples with the smallest scores, and second, to ensure that no other combination of histogram bins can produce join tuples with smaller score values. To this end, histogram bins are joined until: (1) the number of join tuples exceeds k, and (2) the score of any join tuple produced by any unseen histogram bin is not smaller than the score (γ_k) of the current k-th join tuple.

C. The Bound Estimation Algorithm

Algorithm 2 describes our distributed bound estimation method for ranked multi-way join queries on a server S_Q based on the available histograms. The histograms of the m relations R_i are explored in a round-robin fashion. In each iteration, S_Q retrieves the next bin of histogram H^{R_i} (line 7) of R_i from the master server S_M. The retrieved bin is added to a list with retrieved bins from R_i (line 8). Then, the newly retrieved bin is combined with the retrieved bins of the other relations (line 9), and the valid join combinations of bins are kept in a queue. The $getScore()$ method checks the queue and returns the score value of the k-th join tuple (γ_k), such that the number of tuples of the joined bins with score smaller or equal to γ_k sum up to more than k. The actual score of the k-th join tuple is smaller than or equal to γ_k. Also, the total number of tuples of all joined bins in the queue is computed (line 11). Moreover, the estimated score bound e_i for relation R_i is set equal to the high score of the bin (line 12), indicating that tuples of the last retrieved histogram bin of R_i contribute to the join result. Then, the list L_i of servers per relation R_i is updated (line 13). Furthermore, a threshold value t is computed (line 14), which is the best possible (minimum) score that can be produced by unseen histogram bins. The algorithm terminates when the condition of line 15 holds, namely that k join results have been produced and none of the unseen bins can produce a tuple with better score than the k-th best join result retrieved thus far.

Example 2: (Bound Estimation Algorithm) Consider the case of Fig. 4, and assume that the sales manager is interested to find the k=50 most profitable combinations of products for v_0='CPU' in terms of total price (Query Q2). Initially, the querying server S_Q retrieves the first bins $H^{R_0}_{v_0}[0]$ and $H^{R_1}_{v_0}[0]$ of the respective histograms (where v_0='CPU'), thus producing res=60 results. Furthermore, S_Q sets the estimated bounds e_0 and e_1 for each relation, based on the maximum value of the bin range. In this case, e_0=9 and e_1=19. Although the number of results is more than 50 ($res \geq k$), the threshold is t=9 and the 50-th score in the queue is in worst case γ_k=28, so the algorithm does not terminate ($\gamma_k > t$). Therefore, S_Q retrieves the next bins of the histograms, namely $H^{R_0}_{v_0}[1]$ and $H^{R_1}_{v_0}[1]$ and computes the number of join results based on all bins retrieved so far. A total of res=108 results is computed, and the bounds are updated to e_0=19 and e_1=39. The threshold t is now set to 19. Still, another round-trip is required, which updates e_0=29 and e_1=59, and eventually sets t=29. Then the algorithm terminates ($\gamma_k \leq t$) and reports e_0=29 and e_1=59.

Algorithm 2 Bound Estimation.

```
1: Input: Relations $\{R_i\}$, $k$, Function $f$
2: Output: Bounds $e_i$, $0 \leq i < m$
3: $halt \leftarrow false$, $j \leftarrow 0$, $t \leftarrow 0$, $queue \leftarrow \emptyset$
4: $binsR_i \leftarrow \emptyset$, $e_i \leftarrow 0$, $0 \leq i < m$
5: while ($!halt$) do
6:   for ($R_i \in [R_0 \ldots R_m)$) do
7:     $bin_z \leftarrow get(H^{R_i}_{v_z}[j])$, $0 \leq z < n$
8:     $binsR_i$.add($bin_z$), $0 \leq z < n$
9:     $queue$.add($binsR_0 \bowtie \ldots \bowtie binsR_{i-1} \bowtie bin_z \bowtie binsR_{i+1} \bowtie \ldots \bowtie binsR_{m-1}$)
10:    $\gamma_k \leftarrow$ getScore($queue, k$)
11:    $res \leftarrow$ getResultsNo($queue$)
12:    $e_i \leftarrow H^{R_i}_{v_z}[j].high$
13:    $L_i$.update($bin_z$)
14:    $t \leftarrow min\{f(0,..,0,e_x,0,..,0)\}$, $0 \leq x < m$
15:    if ($res \geq k$ and $\gamma_k \leq t$) then
16:      $halt \leftarrow true$
17:    end if
18:  end for
19:  $j \leftarrow j+1$
20: end while
21: return $\{(e_i, L_i), \gamma_k\}$
```

Using Algorithm 2, S_Q computes sufficient bounds of the scoring values e_i for retrieving only the necessary tuples from fragments of R_i. Algorithm 1 relies on this bound estimation to produce the correct result set. The following theorem proves the correctness of the bound estimation.

Theorem 1: (Correctness) The $DRJN$ algorithm using the score bound estimation reports the correct top-k join result set.

Proof: For simplicity, we will show the proof for 2 relations, but the proof can be extended to handle the case of m inputs. Assume that the bound estimation algorithm halts at score bounds e_1 and e_2. Algorithm 1 produces at least k join tuples and the score of the k-th tuple $\tau = \tau_1 \bowtie \tau_2$ is smaller than the threshold $f(\tau) \leq t$, where $t = min\{f(0, e_2), f(e_1, 0)\}$.

The proof is by contradiction. Assume there exists a join tuple $\tau' = \tau'_1 \bowtie \tau'_2$ with $f(\tau') < f(\tau)$ that belongs to the top-k join result set, but it is not produced by our $DRJN$ algorithm (i.e., $\tau'_1.s_1 > e_1$ or $\tau'_2.s_2 > e_2$). Since $f(\tau') < f(\tau)$, this implies that $f(\tau') < t$, hence $f(\tau'_1, \tau'_2) < f(0, e_2)$ and $f(\tau'_1, \tau'_2) < f(e_1, 0)$. From the first inequality, due to the monotonicity of f and since the score $\tau'_1.s_1 \geq 0$, we derive that $\tau'_2.s_2 < e_2$. Similarly, we conclude that $\tau'_1.s_1 < e_1$, which leads to a contradiction. Thus, the join tuple τ' must have been produced by $DRJN$. ■

D. Estimation of the Number of Fetched Bins

When the querying server S_Q accesses the histogram bins from the master server S_M one bin at a time, this may result in many messages and unnecessary networking overhead. An alternative idea is to facilitate access to bins in batches, thereby reducing the cost of bound estimation and making the $DRJN$ framework more efficient. The remaining issue is the estimation of the number of bins (c) required to probe from each histogram in one batch, in order to retrieve the top-k join results. The aim of the querying server S_Q is to estimate c in order to fetch only the necessary bins with one message, and avoid transferring too many bins. We first present an estimation model for two relations, namely R_0 and R_1 and then we extend the model for more relations in a straightforward manner.

Let N_{R_0} and N_{R_1} represent the number of tuples in each relation. Furthermore, let n denote the number of distinct values of the joining attribute and let b denote the number of bins in each histogram of each joining attribute. Assuming uniform distribution of values of the joining attribute, there exist $\frac{N_{R_0}}{n}$ and $\frac{N_{R_1}}{n}$ tuples with the same value of joining attribute, respectively.

Similarly, each bin of relation R contains, on average, $\frac{N_R}{nb}$ join results, assuming uniform distribution over the scoring attribute. As a result, each pair of bins of R_0 and R_1 that join together produces, on average, $\frac{N_{R_0}}{nb} \times \frac{N_{R_1}}{nb}$ results for a specific value of the joining attribute. If x represents the number of join attributes values requested in the selection predicate of query $Q2$ ($x \in [1 \dots n]$), then when joining the corresponding pairs of bins produces, on expectation, $x\frac{N_{R_0}N_{R_1}}{n^2b^2}$ results.

In the case of symmetric join evaluation, assuming that c bins are accessed from each relation, then a total number of c^2 bins can be joined together. We need to identify the value of c, such that:

$$c^2 x \frac{N_{R_0}N_{R_1}}{n^2b^2} = k$$

which leads to:

$$c = nb\sqrt{\left(\frac{k}{xN_{R_0}N_{R_1}}\right)}$$

Extending this formula to m relations, we derive:

$$c = nb\sqrt[m]{\left(\frac{k}{xN_{R_0}\cdots N_{R_{m-1}}}\right)}$$

This estimated value c is used by S_Q as desired number of bins that need to be fetched from S_M.

E. Supporting Different Join Strategies

Algorithm 2 employs a *symmetric join* evaluation as join strategy. Thus, the histograms of the different relations R_i are accessed in a round-robin fashion. Nevertheless, the proposed distributed score bound estimation algorithm supports other join strategies as well. The join strategy influences not only the efficiency of the bound estimation in terms of computational cost, but more importantly the accuracy of the estimated bounds.

For example, a beneficial strategy is HRJN* [7] that prioritizes access to the most promising relations in terms of producing join results. HRJN* intentionally retrieves histograms bins of the relation that has a higher potential to generate results. In more detail, HRJN* observes the retrieved bins and the current score bounds, in order to decide which relation can give a bin that will increase the overall threshold, so that the top-k join tuples can be determined without accessing more bins. Therefore, HRJN* may retrieve fewer histogram bins from some relations leading to more accurate score bounds. In the experimental evaluation, we study in detail the performance gains that can be attained, when the HRJN* strategy is employed.

V. DRJN With Approximate Statistics

The $DRJN$ framework provides sufficient score bounds for each relation for retrieving the top-k join results, in the case that the number of join tuples are accurately estimated. However, in the general case, the *estimated number of join tuples* that will be produced may be inaccurate. We use the term *approximate statistics* to refer to this general case.

For example, consider the case where the domain of the join attribute is of high cardinality, resulting in non-negligible cost if one histogram is created for each join value. Let us assume an enumeration of all join attribute values $\{v_0, \dots, v_{n-1}\}$. Assuming a fixed budget n_a of histograms per relation, we can partition the join attribute values into n_a subsets. When the master server S_M creates the histogram, a plain way to combine the bins of each subset is by employing the uniform frequency assumption [13]. Thus, the indexed bins capture the average distribution of the scores of all join attribute values in this particular subset, leading to inaccurate estimates on the number of join tuples. This comprises a simple example of using approximate statistics, while other examples are examined in Section VII.

In this section, we generalize our framework for handling approximate histograms. The *generalized DRJN algorithm* produces the correct rank-join result at the cost of more than a single phase of score bound estimation.

A. Determining Sufficient Score Bounds

Algorithm 2 can be employed for estimating the score bounds using the available approximate statistics, however two problematic situations can occur after retrieving the tuples based on the estimation:

1) The retrieved tuples produce k' join results, which are fewer than k ($k' < k$), since Algorithm 2 may overestimate the number of join tuples produced by the retrieved bins. In this case, Algorithm 1 cannot return k join tuples.
2) More than k join results are produced, but the score (γ_k) of the k-th join tuple is higher than the estimated k-th score ($\overline{\gamma_k}$), i.e., $\gamma_k > \overline{\gamma_k}$. Algorithm 2 may overestimate the number of join tuples, which in turn leads to the underestimation of the score of the k-th join result. Even though more than k results are produced, the correctness

of the algorithm is not guaranteed, because there may exist join tuples with score in the interval $[\overline{\gamma_k}, \gamma_k]$ that have not been retrieved.

Fortunately, both these situations can be easily detected by checking the number of retrieved tuples, and comparing the estimated score $\overline{\gamma_k}$ to the actual score γ_k of the k-th join tuple. Then, these cases are efficiently handled, as described in the following.

For the first case, the number of the missing values can be computed as $k_m = k - k'$ and the bound estimation (Algorithm 2) is invoked for a second time for $k + k_m$ join tuples. Notice that the number of joined tuples may again be overestimated, leading to fewer than k join results. As long as fewer than k total results have been obtained, another phase of score bound estimation is required. When k join results have been retrieved, either the correct result is already retrieved, or the situation of the second case occurs.

For the second case, at least k join tuples have been retrieved, but our algorithm must check whether there exist join tuples with score in the interval $[\overline{\gamma_k}, \gamma_k]$ that have not been retrieved. Since an upper bound of the score γ_k of the k-th tuple is known, Algorithm 2 is slightly modified in order to take as input the score γ_k and provide score bounds sufficient to produce all join tuples that have scores smaller than or equal to γ_k. Thus, if γ denotes the input score, the only minor modification of Algorithm 2 is to replace the conditions of line 15 by: $\gamma \leq t$, and line 21 should return: $\{(e_i, L_i),\gamma\}$.

Obviously, as the bound estimation needs to be invoked more than a single time, an incremental version of Algorithm 2 can be employed that resumes processing from its previous state of termination. Then, in each score bound estimation phase, only the additional histogram bins are retrieved. Due to space limitations, we omit further implementation details of the incremental algorithm as it is straightforward to implement.

B. The Generalized DRJN Algorithm

To address the aforementioned situations, we introduce a generalized version of the *DRJN* algorithm. Algorithm 3 describes the distributed rank join algorithm in the presence of approximate statistics. The algorithm performs bound estimation, fetches tuples from the servers and executes rank join (lines 6-12), similar to Algorithm 1. If more than k join results are retrieved (line 13) and the score of the k-th result (γ_k) is higher than the estimated score ($\overline{\gamma_k}$) (lines 16-18), then more tuples need to be fetched to ensure the correctness of the result set. More precisely, all tuples that produce join results with score in the interval $[\overline{\gamma_k}, \gamma_k]$ have to be retrieved. For this purpose, a call to the overloaded function of bound estimation is used (line 17), which takes as input the score γ_k instead of a number of tuples k.

On the other hand, if k' ($< k$) results are produced (lines 19-23), then the score bounds need to be updated in order to get additional ($k - k'$) results. In both cases (lines 17, 22), the incremental version of the bound estimation algorithm is invoked. Algorithm 3 terminates and returns the result set, when k join tuples with score lower than $\overline{\gamma_k}$ are retrieved (line 15).

Algorithm 3 The generalized *DRJN* algorithm.

```
1:  Input: Relations $R_i$, $k$, Function $f$
2:  Output: Ranked join result $res$
3:  $halt \leftarrow false$
4:  $tuples_{R_i} \leftarrow \emptyset$, $0 \leq i < m$
5:  $\{(e_i, L_i), \overline{\gamma_k}\} \leftarrow$ BoundEstimation($\{R_i | i \in [0, m)\}$, $k$, $f$)
6:  while (!$halt$) do
7:      for ($i \in [0 \dots m)$) do
8:          for ($P_j \in L_i$) do
9:              $tuples_{R_i} \leftarrow tuples_{R_i} + P_j$.getTuples($e_i$)
10:         end for
11:     end for
12:     $res \leftarrow$ RankJoin($\{tuples_{R_i}\}$)
13:     if ($res \geq k$) then
14:         if ($\gamma_k \leq \overline{\gamma_k}$) then
15:             $halt \leftarrow true$
16:         else
17:             $\{(e_i, L_i), \overline{\gamma_k}\} \leftarrow$BoundEstimation($\{R_i | i \in [0, m)\}$, $\gamma_k$, $f$)
18:         end if
19:     else
20:         $k_m \leftarrow k - k'$
21:         $k \leftarrow k + k_m$
22:         $\{(e_i, L_i), \overline{\gamma_k}\} \leftarrow$BoundEstimation($\{R_i | i \in [0, m)\}$, $k_m$, $f$)
23:     end if
24: end while
25: return $res$
```

Notice that Algorithm 3 returns the correct and complete result set, at the expense of (potentially) more than a single phase of bound estimation and tuple fetching. The following theorem proves that as soon as any k join tuples have been produced, Algorithm 3 terminates with at most one additional phase of bound estimation.

Theorem 2: (Stopping Condition) After any k join tuples have been produced, the generalized *DRJN* algorithm reports the correct top-k join result set with *at most* one additional phase of bound estimation.

Proof: Based on the proof of Theorem 1, *DRJN* reports the correct top-k join result, if the k-th tuple $\tau = \tau_1 \bowtie \tau_2$ has a score γ_k, such that: $f(\tau) = \gamma_k \leq t = min\{f(0, e_2), f(e_1, 0)\}$. We distinguish two cases:

1) $\gamma_k \leq \overline{\gamma_k}$: From the last call of Algorithm 2, we have $\overline{\gamma_k} \leq t$, therefore $f(\tau) = \gamma_k \leq t$. Consequently, the algorithm halts and, since $f(\tau) \leq t$, the reported result is correct. No additional phase of bound estimation is required.
2) $\gamma_k > \overline{\gamma_k}$: The call of Algorithm 2 in line 17 takes as input γ_k and returns $\overline{\gamma_k}$. After this phase of bound estimation, the returned value $\overline{\gamma_k}$ has two properties, namely $\overline{\gamma_k} \leq t$ and $\overline{\gamma_k} = \gamma_k$. Therefore, the algorithm halts because $\overline{\gamma_k} = \gamma_k$, and reports the correct result because $\gamma_k = \overline{\gamma_k} \leq t$. Only a single additional phase of bound estimation was required.

■

VI. DRJN With Distributed Statistics

In several applications it is not feasible to maintain histograms that describe all available data at a single master server S_M. One reason may be that servers are autonomous. Another reason is that the server may become a bottleneck when the query workload increases, since every querying server S_Q needs to access the master server for each posed query, thereby affecting the scalability of the system [14]. As will be demonstrated presently, the $DRJN$ framework is independent of the storage model for histograms, and does not rely on the existence of the master server. Therefore, we propose an alternative storage model based on distributed indexing of histograms over the servers.

In order to create and maintain the histograms in a distributed fashion, we organize the servers in an overlay network. The histograms are created in a completely decentralized manner and stored using a distributed hash table (DHT) that defines the overlay network. The reason for using a DHT as opposed to any other type of overlay network is mainly because DHTs provide guarantees for efficient retrieval, with logarithmic cost with respect to the network size. In addition, DHTs offer advantages with respect to scalability as well as fault tolerance. The DHT provides a simple interface that consists of $put(key, value)$ and $get(key)$ functions. A hash function $h()$ is used to hash histogram bins to servers. Any DHT can be employed to realize our framework, however in our examples and implementation we use Chord [15] as the underlying DHT. Notice that our algorithm for bound estimation is applicable in the case of distributed statistics, by simply plugging in the appropriate $get()$ function of the DHT (line 7 of Algorithm 2).

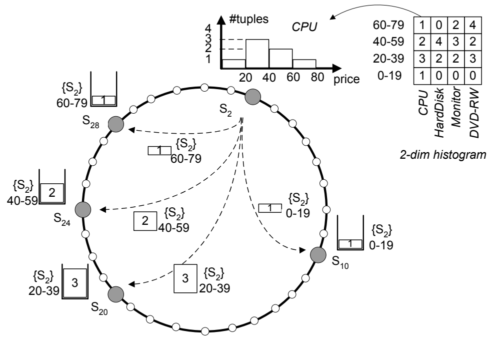
A. Distributed Histogram Indexing

Let us assume that b denotes the number of bins for each join attribute. We assume that each histogram $H_{v_z}^{R_i}$ has the same number of bins. Obviously this is not a requirement, but we use it for ease of presentation. We can define a total ordering (enumeration) of relations and histogram bins. Any bin of the histogram that belongs to a particular relation is assigned a unique identifier ℓ, which belongs to the range of values $\ell \in [0 \ldots m \times n \times b)$. The j-th bin $H_{v_z}^{R_i}[j]$ of the z-th histogram of the i-th relation is assigned the unique identifier: $\ell(i, z, j) = i \times n \times b + z \times b + j$.

The hash function maps each bin to a server S_H and is defined as:

$$h() : [0 \ldots m \times n \times b) \rightarrow [0 \ldots N_S), \text{ such as}$$
$$h(H_{v_z}^{R_i}[j]) = \ell(i, z, j) \bmod N_S$$

In a nutshell, each server adds its own piece of statistics to the network, by placing bins of histograms (of its own data) to specific servers, according to an appropriate hash function $h()$, as shown in Fig. 5. This does not compromise server autonomy, since only metadata (statistics) of restricted size are indexed by the DHT. As several servers may store tuples of same relation and their scoring values may belong to the same histogram bin, S_H aggregates the information in the individual bins and maintains the total number of tuples, as well as the list of servers that store the respective tuples. Notice that the hash function may use a coarse partitioning of the score domain, while the servers create bins independently with the only restriction being that each bin is completely contained within one partition used in hashing.



	CPU	HardDisk	Monitor	DVD-RW
60-79	1	0	2	4
40-59	2	4	3	2
20-39	3	2	2	3
0-19	1	0	0	0

Fig. 5. Distributed indexing of histogram bins in Chord.

In addition, S_H maintains the individual histogram bins, in order to efficiently handle updates of servers. Maintenance of histograms follows a soft-state approach, similar to the technique employed in [16].

Additionally, standard techniques used for load balancing [17] and replication in DHTs [18] can be used in the $DRJN$ framework. Another plain solution is to employ multiple hash functions to share the load to multiple servers.

B. Reducing the Cost of Bound Estimation

The technique used for distributed indexing of the histograms uses as its basic indexing unit a histogram bin. In many situations, retrieval of the histograms during query execution is improved by enabling batch access to multiple consecutive histogram bins using a single message (as already described in Section IV-D). For this purpose, an alternative indexing scheme is required that intentionally places groups of bins to the same server. Although several alternatives for grouping histogram bins do exist, we demonstrate how the indexing can be adapted to support queries similar to query $Q2$ (see Fig. 1), which are very common in practical applications[1]. Such queries restrict the values v_z of the join attribute(s) using selection predicates. In the case that the observed query workload in the system consists mainly of queries like $Q2$ that refer to different selection predicates, it is beneficial to group bins by their corresponding v_z value and place them on the same server.

The alternative indexing scheme of bins in the DHT is implemented by simply changing the bins enumeration. In particular, all bins that belong to the histogram of joining value v_z are assigned the same unique identifier. Algorithm 2 can be easily adapted to support accessing histogram bins in a batch. In each iteration, instead of requesting one bin of each histogram $H_{v_z}^{R_i}$, a group consisting of c bins is requested. In

[1]We emphasize that the particular grouping of bins is used merely as a showcase, and other groupings can be supported by the $DRJN$ framework by adapting the indexing scheme.

```
SELECT *
FROM Customers C, Suppliers S
WHERE C.product = S.product
  AND C.product = 'CPU'
  AND S.location = 'New York'
ORDER BY (C.price+S.discount)
LIMIT k
```
(a)

```
SELECT *
FROM Customers C, Suppliers S, Factory F
WHERE C.product = S.product
  AND S.location = F.location
  AND C.product = 'CPU'
  AND S.location = 'New York'
ORDER BY (C.price+S.discount+F.quality)
LIMIT k
```
(b)

Fig. 6. Query with: (a) additional predicates, (b) multiple join attributes.

this way, we can reduce communication cost, by using one message to transmit a group of bins, instead of one bin.

VII. Support for More Complex Queries

In this section, we examine the case of more complex queries, which are commonly encountered in practice. In particular, we demonstrate that our framework supports such queries by means of the generalized $DRJN$ algorithm.

A. Queries with Additional Predicates

So far, our examples and algorithms assumed that selection predicates were applied only on the join attributes. However, quite often a query may contain additional predicates on other (non-join) attributes. In the following, we show that the generalized $DRJN$ framework supports such queries effectively. As an example, consider the query in Fig. 6(a) which uses a predicate on attribute *location* of relation S. Notice that any other attribute of S could be used in the query instead of *location*. Obviously, we can no longer assume the existence of a histogram that captures the distribution of the score values per joining attribute of relation S for the tuples that *S.location = 'New York'* holds. Such an assumption would make the number of required histograms explode, since the possible combinations of predicates may be huge.

Instead, our premise is to perform bound estimation using only the histogram on *Discount* and *Product*, similarly to the previous sections. Algorithm 2 for bound estimation cannot be employed, as its derived bounds would not be sufficient for retrieving the top-k join results. This is because the estimated number of join tuples overestimates the actual number of join tuples that satisfy the predicate on *S.location*. Clearly, this is a case of approximate statistics. Therefore, if we employ the generalized $DRJN$ algorithm (without any modifications) that handles approximate statistics, then we can effectively process the query.

B. Queries with Multiple Join Attributes

In the general case, different attributes of relation R_i can be used as join attributes in different queries. Therefore, it is natural to index one histogram for each join attribute of R_i. Then, for any query that uses a single join attribute of R_i, Algorithm 1 can be applied to produce the correct result. This corresponds to queries, such as $Q1$ and $Q2$, that have been examined so far in previous sections.

Nevertheless, if more than one join attributes of R_i appear in the same query, then the correctness of Algorithm 1 is not guaranteed. For example, consider the query in Fig. 6(b) that uses two join attributes *product* and *location* of S. Let us assume the existence of histograms on *product* and *discount*, and *location* and *discount*. For instance, the first histogram provides information on the number of tuples of S that have a score (*discount*) in a particular range and refer to a specific *product* (e.g. 'CPU'). However, given these histograms, it is not possible to infer how many tuples of those belong to a particular *location* (e.g. 'New York'). Thus, the number of tuples that satisfy both join conditions cannot be estimated accurately.

In order to have enough information to compute the exact number of join tuples, a histogram is required that captures the number of tuples for any combination of join values of different join attributes. Unfortunately, as the number of potential join attributes increases, the number of such combinations grows rapidly. Thus, this solution becomes quite costly in practice. Instead, in the following, we show how the existing histograms can be used to estimate the number of join tuples.

The proposed solution is to take the minimum value of the bins for 'CPU' and 'New York' as the number of estimated join tuples. This is because in the best case these tuples will have both values 'CPU' and 'New York'. Therefore, the estimated number of join tuples will be an overestimation of the real number of join tuples. Consequently, the generalized $DRJN$ algorithm can be applied to retrieve the complete and correct result set.

VIII. Experimental Evaluation

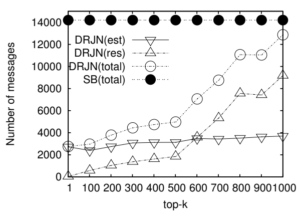
In this section, we provide an extensive experimental evaluation of our proposed framework. We implemented in Java all algorithms and simulated the server interconnections. In all experiments, we adopt the more challenging setup that uses distributed statistics, which incurs higher cost of estimation when compared to using a single master server. The DHT employed in our experiments is Chord [15].

A. Experimental Setup

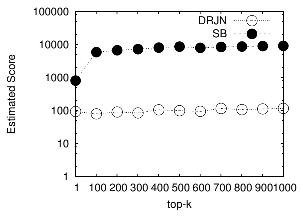
Datasets and queries. We use the following synthetic data distribution for generating the scoring attributes of relations: a) uniform (UN), and b) skewed (zipf distribution) with varying parameter of skewness 0.5 and 1, denoted as ZI0.5 and ZI1.0 respectively. After data generation, the tuples are assigned to servers uniformly at random. We evaluate two generic types of queries $Q1$ and $Q2$. We use the weighted sum function as scoring function, and each query differs from another, due to the random generation of different weights. In all cases, 100 queries of each type are generated at random and we present average results in all charts.

Comparative evaluation. We compare our distributed bound estimation framework against a bound estimation approach based on *sampling* (denoted as SB), which is the state-of-the-art for ranked join processing in highly distributed systems [3]–[5]. SB estimates score bounds by examining only local data tuples on the querying server S_Q, and then SB retrieves tuples from all servers using the estimated bounds.

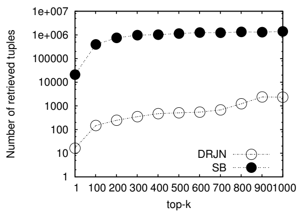
In addition, we study the effect of different rank join strategies on the performance of the $DRJN$ framework. We use symmetric join evaluation (denoted as $DRJN$) and the



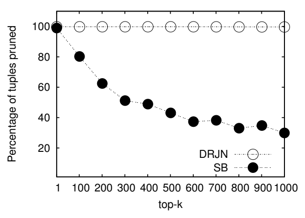
(a) Number of messages.



(b) Estimated scores.

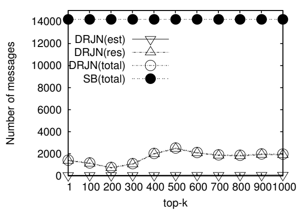


(c) Retrieved tuples.

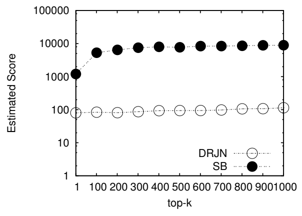


(d) Percentage of tuples pruned.

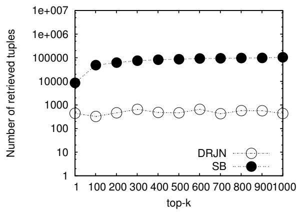
Fig. 7. Comparative performance of $DRJN$ and SB for $Q1$, s=0.02 and zipfian dataset (ZI1.0).



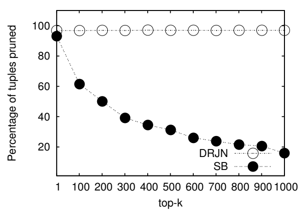
(a) Number of messages.



(b) Estimated scores.



(c) Retrieved tuples.



(d) Percentage of tuples pruned.

Fig. 8. Comparative performance of $DRJN$ and SB for $Q2$, s=0.06 and zipfian dataset (ZI0.5).

HRJN* join strategy (denoted as $DRJN*$). Furthermore, we explore the quality of the results obtained using the variant of our framework that employs approximate statistics.

Metrics. Our metrics include: a) the number of messages required for score bound estimation and result retrieval, b) the estimated score bounds, c) the number of retrieved tuples, d) the percentage of pruned tuples at each server locally, and e) (in the case of approximate statistics) the quality of the results in terms of completeness (recall) after a *single phase* of bound estimation. We point out that all metrics used can be accurately measured using simulations, since they are independent of the actual networking environment used for deployment. This makes the method of simulation particularly appropriate for our experimental evaluation, as it additionally facilitates the scalability study of our framework.

Parameters. Unless mentioned explicitly, we use the following *default setup* in our experiments: number of servers N_S=1024, cardinality of each relation $|R_i|$=1M, join selectivity s=0.02, uniform dataset, query $Q2$ and the join strategy is symmetric join. We vary the network size (N_S) from 512 to 4096 servers. We also vary the join selectivity (s) from $2{\cdot}10^{-3}$ to 10^{-1}. In order to test the effect of k we vary its value from 1 to 1000. We emphasize that the size of the relations does not affect the performance of $DRJN$, as the size of the result set is always k, therefore we do not vary the values of $|R_i|$.

B. Experimental Results

Experiments with Query Q1. In Fig. 7, we examine the comparative performance of $DRJN$ and SB for zipfian dataset with skew equal to 1 (ZI1.0) and join selectivity s=0.02. Recall that query $Q1$ does not have any selection predicate. Thus, processing $Q1$ is quite demanding, since any join combination of any predicate value can appear in the top-k results.

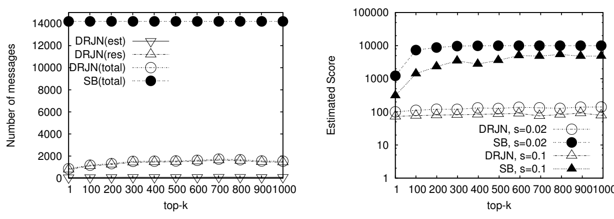
First, in Fig. 7(a), we show the number of messages that $DRJN$ requires for estimation ($DRJN$(est)), for retrieval of results ($DRJN$(res)), as well as the total number of messages ($DRJN$(total)). In addition, we depict the total number of messages for SB (SB(total)). Notice that the total cost of SB is due to result retrieval, as SB has zero cost for estimation because it uses only local data. $DRJN$ needs always almost 5 times fewer messages than SB for queries with $k \leq 500$, which are more common in practice. We observe that for $Q1$ the cost of estimation dominates the total cost of $DRJN$. This is attributed to the fact that the estimation of $DRJN$ is accurate enough to prune many tuples during result retrieval in contrast to SB, as will be shown in Fig. 7(c). Also notice that the cost of estimation remains practically stable as k increases.

In addition, we show the estimated score bounds of each algorithm in Fig. 7(b). As we are interested for top-k join results with minimum scores, the estimated score values should be as small as possible. The estimation of $DRJN$ is almost two orders of magnitude better than SB. This enables $DRJN$ to retrieve only few tuples that have scores smaller than the estimated score bounds, and it explains the significant gain in terms of number of messages attained by $DRJN$. For the same experiment, we also report the number of tuples retrieved by each approach in Fig. 7(c). The loose bound estimation of SB results in the retrieval of too many tuples, which makes the networking cost excessive. Moreover, the individual processing cost at each server increases, as redundant work is performed to retrieve unnecessary tuples. In contrast, $DRJN$ needs to retrieves 3 orders of magnitude fewer tuples, to produce the top-k join result. This gain is because: a) the estimated score bound is tighter, thus reducing the number of tuples that are retrieved from each server, and b) only a limited set of servers that actually store tuples with scores smaller than the bound are contacted. In Fig. 7(d), we also measure the percentage of tuples pruned at servers due to the estimated scores. In all cases, $DRJN$ manages to prune almost 100% of the local tuples that do not affect the top-k join result, while SB performs worse and its performance deteriorates with increasing values of k.

We repeated the same experiment using the zipfian (ZI0.5) and the uniform (UN) datasets (figures omitted due to space limitations). In all setups, $DRJN$ consistently outperforms SB and the relative gain is always high.

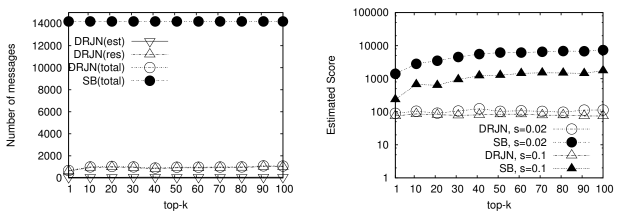
Experiments with Query Q2. In the following, we test the performance of our framework using query $Q2$, which uses a predicate additionally to the join condition. Fig. 8 depicts the results for synthetic dataset (ZI0.5) and we set s=0.06.

As shown in Fig. 8(a), $DRJN$ needs always one order of magnitude fewer messages than SB in total for processing the top-k join. We observe that $DRJN$ can process $Q2$ with smaller cost than $Q1$, when comparing to Fig. 7(a), because the presence of a predicate in $Q2$ reduces the number of candidate join tuples that may appear in the top-k results. The chart in Fig. 8(b) shows the estimated scores. Again, $DRJN$ clearly outperforms the competitive method for all values of k. In Fig. 8(c), the number of retrieved tuples is depicted. $DRJN$ needs to retrieve about two orders of magnitude fewer tuples, thus providing a feasible solution compared to SB, which needs to transfer too many tuples. Finally, in Fig. 8(d), the percentage of pruned tuples is shown. $DRJN$ manages to prune eagerly almost 98% of the local tuples that qualify the predicate condition of $Q2$, thereby also saving processing costs at individual servers.



(a) Number of messages. (b) Estimated scores.

Fig. 9. Experiments with uniform (UN) dataset and $Q2$.



(a) Number of messages. (b) Estimated scores.

Fig. 10. Experiments with $k \leq 100$, UN and $Q2$.

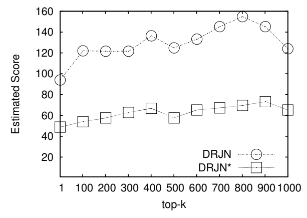
Experiments with Uniform (UN) data. In Fig. 9, we test the performance of $DRJN$ using a uniform dataset and $Q2$. As shown in Fig. 9(a), we set the join selectivity to s=0.02. Even though the uniform data distribution is not likely to appear in real applications, we examine the uniform case as it benefits the competitor sampling approach (SB). SB performs best in the case of uniformity, while for skewed datasets the local samples cannot accurately provide good score estimates. However, we observe that $DRJN$ is still more efficient in terms of required number of messages by one order of magnitude. In Fig. 9(b), we compare the estimated scores for s=0.02 and s=0.1. Even in the case of the uniform dataset, $DRJN$ is more than one order of magnitude better in estimating scores for all tested values of join selectivity.

Experiments with $k \leq 100$. In Fig. 10, we examine the case of smaller values of k, which are quite common in practice. Again, we use the uniform dataset, in order to study the setup where the competitor approach SB performs best. In Fig. 10(a), we show the number of messages required by each algorithm for processing the distributed top-k join query. $DRJN$ still outperforms SB even for smaller values of $k \leq 100$. In Fig. 10(b), we show the estimated scores of each algorithm for different values of s=0.02 and s=0.1. SB achieves its best estimates for s=0.1, however even in that case, $DRJN$ provides significantly more accurate score estimates, as shown in the chart. In general, we conclude that the benefits of $DRJN$ over SB are sustained for smaller values of k.

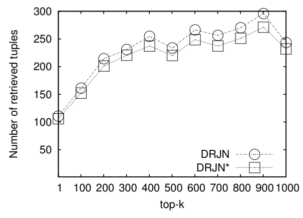
Comparison of Join Strategies. We also compare the performance of our framework using two different join strategies: symmetric join (denoted $DRJN$) vs. HRJN* [7] (denoted $DRJN$*) for $Q2$. First, in Fig. 11(a), we assess the accuracy of estimation between $DRJN$ and $DRJN$*. For all values of k, $DRJN$* computes more accurate bounds. This is in accordance with the results of centralized settings. In Fig. 11(b), the improvement in score estimation is reflected in the reduced number of transferred tuples by $DRJN$*. Thus, the tighter scores computed by $DRJN$* reduce the communication costs at query processing, by retrieving fewer tuples over the network. Moreover, we depict in Fig. 11(c) the improvement of $DRJN$* over $DRJN$ in terms of number of messages required for the estimation. This quantifies the cost of retrieval of histogram bins, in order to estimate the scores. As shown in the chart, $DRJN$* manages to significantly decrease the number of messages required for bound estimation. The same conclusion is drawn by inspecting Fig. 11(d) for the ZI0.5 dataset. The only difference is that in this case the absolute number of messages is smaller than for the UN dataset.

Scalability with Network Size. In Fig. 12, we study the scalability of our approach with respect to network size (from 512 to 4096 servers). We increase the size of each relation to have the same number of tuples per server. The number of tuples ($|R_i|$) in each relation that participates in query ranges from 500K to 4M tuples. We set top-k=500, join selectivity s=0.02, we use the uniform dataset and $Q2$.

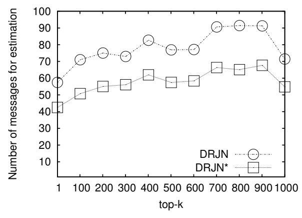
First, we compare the quality of estimation in Fig. 12(a). Again, our framework outperforms estimation based on sampling by two orders of magnitude, irrespective of the network size for both join strategies ($DRJN$ or $DRJN$*). This verifies the scalability of our framework with network size. These benefits are also reflected in the number of retrieved tuples (Fig. 12(b)) that each approach needs to transfer over the network, in order to produce the correct top-k join result. Our framework outperforms the approach based on sampling consistently. We emphasize that the number of retrieved tuples for producing the top-k join results is the dominant factor for communication costs, therefore our approach scales gracefully with increasing number of servers.



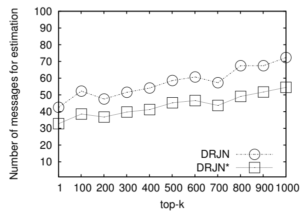
(a) Estimated scores (UN).



(b) Retrieved tuples (UN).

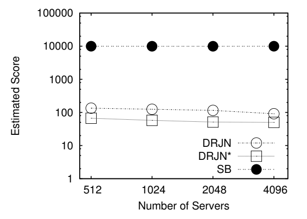


(c) Cost of estimation (UN).

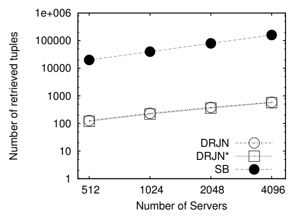


(d) Cost of estimation (ZI0.5).

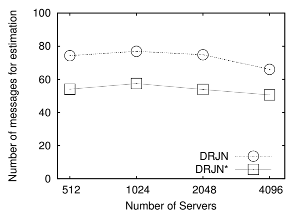
Fig. 11. Comparative performance of different join strategies ($DRJN$ vs. $DRJN$*) for $Q2$.



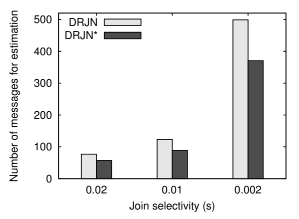
(a) Estimated scores.



(b) Retrieved tuples.

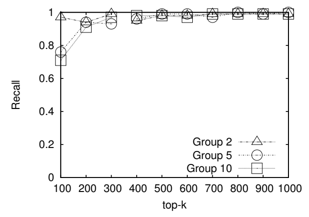


(c) Cost of estimation.

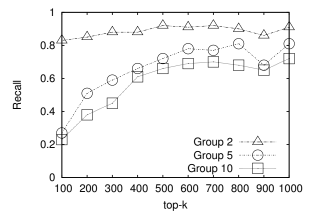


(d) Varying join selectivity.

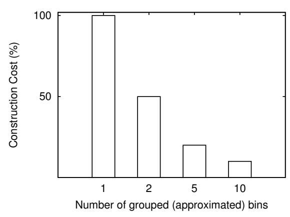
Fig. 12. Scalability study with network size and varying join selectivity.



(a) Uniform dataset (UN).



(b) Zipf dataset (ZI0.5).



(c) Savings in construction cost.

Fig. 13. Experiments with approximate statistics.

Furthermore, we compare the number of messages required for estimation only for the two join strategies employed in our framework. As sampling does not need to exchange messages in order to make the estimate, it is not included in the chart of Fig. 12(c). Notice that our framework requires only a limited set of messages for performing the estimation, and this number is not significantly affected by the actual size of the network. However, this overhead is trivial compared to the number of tuples transferred by the competitor approach based on sampling. This also demonstrates the scalability of our approach. Moreover, in terms of the individual join strategies, $DRJN$* requires fewer messages to estimate the necessary score bounds that produce top-k join results.

In addition, in Fig. 12(d), we study the effect of varying the join selectivity (s) on the number of messages required for estimation. We use 1024 servers, top-k=500, the uniform dataset and $Q2$. We observe that for small values of join selectivity ($2 \cdot 10^{-3}$), both join strategies require more messages for estimation, because the finding the top-k join results becomes harder. However, the comparative gain of $DRJN$* over $DRJN$ also increases, and for $s = 2 \cdot 10^{-3}$, $DRJN$* saves more than 25% of the cost of $DRJN$. This demonstrates that $DRJN$* is a viable solution for setups with small values of s.

Experiments with Approximate Histograms. In Fig. 13, we present the results obtained using the approximate statistics. We test three different numbers (2, 5 and 10) of bins that are grouped into one using the uniform frequency assumption, as described in Section V. We evaluate the performance of the $DRJN$ framework with symmetric join, by measuring the recall achieved after a single phase of bound estimation. Recall quantifies the number of retrieved join results that actually belong to the real top-k join results. We stress that the algorithm produces the correct result and we plot the recall achieved after the first phase of bound estimation only.

In Fig. 13(a), we use the uniform dataset. As depicted in the chart, the algorithm achieves high recall values almost for all values of k. This is due to the fact that the uniform distribution allows a good approximation. Therefore, we also try the performance of our algorithm using a skewed dataset. We use the zipfian distribution ZI0.5, in order to measure the effect of approximate statistics in a skewed dataset. Fig. 13(b) shows that for 'Group 2' the recall is always over 80% after the first phase of estimation, regardless of k. Also for the other setups, we manage to achieve significant recall values, especially for higher values of k. Then, in Fig. 13(c), we show the savings in construction costs, when grouping histogram bins. The cost relates to the number of messages required to index the histograms in the DHT. For the case of 'Group 2', we save almost 50% of the construction cost required when no grouping is used.

IX. Related Work

Rank-aware query processing and optimization [8], [19]–[21] has attracted much interest in the database community lately. Fagin et al. [1] focus on equi-joins of ranked data when the joining attribute is a unique identifier present in all relations. KLEE [2] is proposed to efficiently handle the same query type in a much more distributed setting. However, in this work, we are interested in a generalization of this problem, focusing on arbitrary user-defined join attributes between relations. It is not straightforward how to adapt Fagin's algorithms nor KLEE to address this problem. A variety of rank join algorithms have been proposed for centralized settings, including J^* [9], *NRA-RJ* [22], *rank-join* algorithm [7], and DEEP [10].

Despite the importance of rank join query processing, only few studies focus on widely distributed data sources, including *PJoin* [3], [4] and the approach described in [5]. All those approaches are based on sampling to estimate score bounds that prune tuples which cannot belong to the result set. This is the state-of-the-art in distributed rank join processing.

In contrast, we propose an approach that exploits statistics in the form of histograms, which can be stored in either a centralized or distributed manner, in order to derive tight score bounds that eagerly prune irrelevant tuples. Hence, we propose query processing algorithms that do not rely on sampling, thus making our approach robust independently of the underlying data distribution.

We recognize existing work on joins in P2P systems, such as PeerDB [23], PIER [16], or continuous joins [24]. However, these systems cannot be straightforwardly adapted to support rank joins. Distributed statistics in the form of equi-width histograms have also been used to support different queries or operations, such as distributed top-k queries [2] or cardinality estimation [25]. A recent related work is [26], however the focus is on the overhead of accessing each data source.

X. Conclusions

In this paper, we studied rank join query processing in highly distributed environments. Our framework ($DRJN$) relies on statistics that enable the establishment of a bound for the score value of each relation that is sufficient to retrieve the necessary tuples for producing the final results. The $DRJN$ framework exploits the stored statistics and estimates score bounds that guarantee the correctness of the result. Moreover, we generalized our framework to support approximate statistics, still producing the correct result, at the expense of more than a single phase of score estimation. In addition, we showed that $DRJN$ is applicable even when the statistics are stored in a distributed manner. Finally, we demonstrated the efficiency of our framework through extensive experiments.

Acknowledgment

Christos Doulkeridis was supported under the Marie-Curie IEF grant number 274063. Akrivi Vlachou was partially supported by the Greek State Scholarship Foundation (IKY). Yannis Kotidis was partially supported by the Basic Research Funding Program, AUEB.

References

[1] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proceedings of PODS*, 2001, pp. 102–113.

[2] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: A framework for distributed top-k query algorithms," in *Proceedings of VLDB*, 2005, pp. 637–648.

[3] K. Zhao, S. Zhou, K.-L. Tan, and A. Zhou, "Supporting ranked join in peer-to-peer networks," in *DEXA Workshops*, 2005, pp. 796–800.

[4] K. Zhao, S. Zhou, and A. Zhou, "Towards efficient ranked query processing in peer-to-peer networks," in *Cognitive Systems*, 2005, pp. 145–160.

[5] J. Liu, L. Feng, and C. He, "Semantic link based top-k join queries in P2P networks," in *Proceedings of WWW*, 2006, pp. 1005–1006.

[6] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[7] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases," in *Proceedings of VLDB*, 2003, pp. 754–765.

[8] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter, "Adaptive rank-aware query optimization in relational databases," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1257–1304, 2006.

[9] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter, "Supporting incremental join queries on ranked inputs," in *Proceedings VLDB*, 2001, pp. 281–290.

[10] K. Schnaitter, J. Spiegel, and N. Polyzotis, "Depth estimation for ranking query optimization," in *Proceedings of VLDB*, 2007, pp. 902–913.

[11] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: building histograms without looking at data," in *Proceedings of SIGMOD*, 1999, pp. 181–192.

[12] D. Donjerkovic, R. Ramakrishnan, and Y. Ioannidis, "Dynamic histograms: Capturing evolving data sets," in *Proceedings of ICDE*, 2000, p. 86.

[13] V. Poosala, V. Ganti, and Y. E. Ioannidis, "Approximate query answering using histograms," *IEEE Data Eng. Bull.*, vol. 22, no. 4, pp. 5–14, 1999.

[14] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proceedings of SIGMOD*, 2010, pp. 591–602.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of SIGCOMM*, 2001, pp. 149–160.

[16] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi, "The architecture of PIER: an internet-scale query processor," in *Proceedings of CIDR*, 2005, pp. 28–43.

[17] B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proceedings of INFOCOM*, 2004.

[18] R. Akbarinia, E. Pacitti, and P. Valduriez, "Data currency in replicated DHTs," in *Proceedings of SIGMOD*, 2007, pp. 211–222.

[19] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation," *ACM Trans. Database Syst.*, vol. 27, no. 2, pp. 153–187, 2002.

[20] C.-M. Chen and Y. Ling, "A sampling-based estimator for top-k selection query," in *Proceedings of ICDE*, 2002, pp. 617–627.

[21] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, 2008.

[22] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Joining ranked inputs in practice," in *Proceedings VLDB*, 2002, pp. 950–961.

[23] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou, "PeerDB: a P2P-based system for distributed data sharing," in *Proceedings of ICDE*, 2003, pp. 633–644.

[24] S. Idreos, E. Liarou, and M. Koubarakis, "Continuous multi-way joins over distributed hash tables," in *Proceedings of EDBT*, 2008, pp. 594–605.

[25] N. Ntarmos, P. Triantafillou, and G. Weikum, "Counting at large: Efficient cardinality estimation in internet-scale data networks," in *Proceedings of ICDE*, 2006, p. 40.

[26] B. Arai, G. Das, D. Gunopulos, V. Hristidis, and N. Koudas, "An access cost-aware approach for object retrieval over multiple sources," in *Proceedings of VLDB*, 2010.