Depth Estimation for Ranking Query Optimization

Karl Schnaitter
UC Santa Cruz
karlsch@soe.ucsc.edu

Joshua Spiegel *
BEA Systems, Inc.
jspiegel@bea.com

Neoklis Polyzotis
UC Santa Cruz
alkis@cs.ucsc.edu

ABSTRACT

A relational ranking query uses a scoring function to limit the results of a conventional query to a small number of the most relevant answers. The increasing popularity of this query paradigm has led to the introduction of specialized rank join operators that integrate the selection of top tuples with join processing. These operators access just "enough" of the input in order to generate just "enough" output and can offer significant speed-ups for query evaluation. The number of input tuples that an operator accesses is called the *input depth* of the operator, and this is the driving cost factor in rank join processing. This introduces the important problem of *depth estimation*, which is crucial for the costing of rank join operators during query compilation and thus for their integration in optimized physical plans.

We introduce an estimation methodology, termed DEEP, for approximating the input depths of rank join operators in a physical execution plan. At the core of DEEP lies a general, principled framework that formalizes depth computation in terms of the joint distribution of scores in the base tables. This framework results in a systematic estimation methodology that takes the characteristics of the data directly into account and thus enables more accurate estimates. We develop novel estimation algorithms that provide an efficient realization of the formal DEEP framework, and describe their integration on top of the statistics module of an existing query optimizer. We validate the performance of DEEP with an extensive experimental study on data sets of varying characteristics. The results verify the effectiveness of DEEP as an estimation method and demonstrate its advantages over previously proposed techniques.

1. INTRODUCTION

Ranking queries, which have been popularized mainly in the context of information retrieval, have become increasingly popular in the context of relational databases. In a nutshell, a relational ranking query (also referred to as a top-K query) specifies a scoring function over the results of a SELECT query and limits the result set to the K tuples with the highest scores. Ranking queries are invaluable in the interactive exploration of large data stores, as they allow a user to reduce a massive result set to a few of the most relevant answers. As an example, consider a database comprising information on a touristic area. A visitor that wants to plan out a short stay in the area can issue the query shown in Figure 1 to retrieve the top 10 combinations of a hotel, a restaurant, and an event that are located in the same city. The scoring function encodes the preferences of the visitor and combines the rating of the restaurant and the hotel with a boolean indicator on the event title (whether it matches the keyword "music").

SELECT *hl.name*, *rt.name*, *ev.title*
FROM *Hotel hl*, *Restaurant rt*, *Event ev*
WHERE *hl.city* = *rt.city* **AND** *rt.city* = *ev.city*
RANK BY $0.4*$*hl.rating* + $0.2*$*rt.rating*+ $0.4*$**matches**(*ev.title*,"music")
LIMIT 10

Figure 1: An example ranking query.

Clearly, it is possible to evaluate a ranking query by generating the results of the join, ordering them by score, and retaining the first K tuples. This approach can result in wasted work, however, as the physical plan may need to process all of the input tuples in order to return only a few results. As a potentially more efficient alternative, recent studies [6, 7] have introduced specialized *rank join* physical operators that enable a physical plan to integrate top-K selection with join processing. Figure 2 illustrates the main idea behind this approach. The figure shows a sample physical plan for the above query that uses rank join operators. (The specifics of the plan and the notation are described later.) The ordered answers are obtained by performing K pull requests at the top operator. Each pull request, in turn, accesses and combines tuples from the base tables in descending order of their "importance" as specified by the query. (For instance, restaurant tuples will be accessed in descending order of their rating.) Overall, the idea is to generate the answers by accessing only a subset of each input table that contains the most "important" tuples. (The figure illustrates the accessed subsets as shaded regions over the rectangles that represent the complete tables.) Depending on the input data and the specifics of the query, this rank-aware processing can offer significant performance improvements compared to a conventional plan that generates and sorts the results of the join [6, 7, 11].

The use of a rank join operator in a physical plan hinges on the ability of the optimizer to obtain estimates for the cost factors of the operator. In this specific context, the driving cost factor is the number of tuples that the operator accesses from each input, which we refer to as the *input depth* of the operator. This raises the important problem of *depth estimation* that is crucial for the integration of rank join operators in query optimization. At an abstract level, depth estimation is analogous to the problem of estimating the output cardinality of a conventional join operator, for which previous

*Work performed while the author was a graduate student at UC Santa Cruz.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

studies have introduced a host of effective techniques (e.g., based on histograms [15], wavelets [14], or samples [12]). The formulation of the problem, however, is essentially reversed: the output cardinality of the physical plan is known to be the number K of requested tuples, and we wish to estimate the amount of the input that is accessed.

Prior Work. As hinted earlier, previous works on conventional selectivity estimation are not well suited for the semantics of depth estimation. The same observation applies to estimation techniques in more relevant domains, such as quantile estimation for single tables [13] and the cost optimization of skyline queries [3], since they essentially ignore the specifics of rank joins and their operation within a complex query plan. To address this issue, recent studies have introduced specialized techniques that are tailored to the problem of depth estimation. More concretely, Ilyas et al. [7] have proposed a technique that derives estimates of input depths using a probabilistic model of the data. The particular data model, however, assumes that scores are uniformly distributed and joins are independent, and it is thus not obvious whether a similar estimation method can be applied to a more general model of score distributions and join dependencies. Moreover, the estimator requires that all relations have equal size and the scoring function is a weighted sum. Extending the methodology to a different class of inputs requires a non-trivial modification to the estimators. A different approach is taken in the work of Li et al. [11], where the estimator uses independent samples of the base relations as the core summarization method. The key idea is to process the physical plans over the samples, and then scale up the observed depths in order to obtain estimates. Independent table samples, however, are known to perform poorly in capturing key/foreign key joins or complex join dependencies, which makes this method likely to exhibit high estimation errors in practice.

The main shortcoming of previously proposed methods is the lack of a formal framework that links depth estimation to the statistical characteristics of the data distribution. Such a framework determines the type of data statistics that need to be maintained by the optimizer, and also provides a principled method for computing depth estimates based on the stored statistics. In addition, the estimation framework can clarify whether the depth statistics are realizable with existing data summarization techniques, or if we need to invest in new types of data synopses. Overall, it becomes clear that this missing link is essential for the effective integration of rank join operators in the query optimizer.

Contributions. Motivated by the previous observations, we present a new technique for estimating the input depth of physical operators used by ranking queries. Our individual contributions can be summarized as follows:

• DEEP **Estimation Framework.** We introduce DEEP (short for DEpth Estimation for Physical plans), a formal framework for depth estimation in rank-aware physical plans that employ monotonic scoring functions. The key novelty of DEEP is the rigorous computation of input depths in terms of aggregate data statistics. The proposed framework relies on frequency tensors in order to capture score distributions, and formalizes depth computation as a set of operations over the tensors that describe the output and inputs of a join operator. This computation is designed for two rank join operators that represent the state-of-the-art: Nested Loops Rank Join, and the HRJN* variant of Hash Rank Join algorithms.

• **Realization of** DEEP **with Efficient Estimation Algorithms.** We develop estimation algorithms that provide an efficient implementation of the formal DEEP framework. Our algorithms take advantage of the monotonicity of the scoring function in order to process only the relevant portion of the data statistics, and thus significantly improve the performance of depth estimation. We describe a simple interface that these algorithms use to access the statistics module of the query optimizer, and demonstrate how it can be implemented on top of summarization techniques that are commonly used in relational systems. This demonstrates the practicality of our approach and its feasibility as an estimation method.

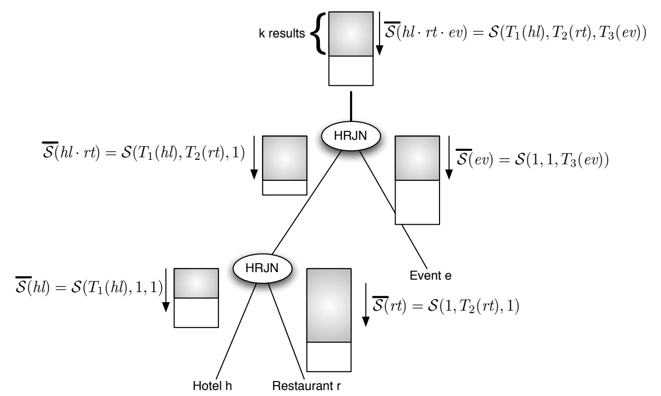


Figure 2: A physical plan with rank-join operators.

• **Theoretical Analysis of Hash Rank Join.** The design of DEEP focuses on HRJN* rather than other Hash Rank Join algorithms that have been studied. In order to justify this decision, we perform a theoretical analysis of the HRJN* join algorithm and show that it exhibits strong optimality properties compared to other variants of Hash Rank Join. Our analysis corroborates the empirical results of previous studies, while making major extensions to the known theoretical properties of HRJN*. Hence our results are also of general interest for the further development of rank join algorithms.

• **Experimental Study Validating the Effectiveness of** DEEP**.** We validate the effectiveness of the proposed estimation methodology with an extensive experimental study on data sets and workloads of varying characteristics. The results verify the effectiveness of DEEP in computing accurate depth estimates for complex physical plans and show that, in several cases, DEEP offers between one to two orders of magnitude improvement in relative error over previously proposed estimation techniques.

2. PRELIMINARIES

In this section, we introduce some basic concepts related to relational ranking queries and formally define the estimation problem.

2.1 A Primer in Ranking Query Processing

Query Model. We adopt the notation of previous works and express a ranking query in the following SQL-like language:

SELECT *Target List*
FROM $R_1\ \tau_1, R_2\ \tau_2, \ldots, R_n\ \tau_n$
WHERE *Join Predicates* **AND** *Selection Predicates*
RANK BY $\mathcal{S}(T_1(\tau_1), T_2(\tau_2), \ldots, T_n(\tau_n))$
LIMIT K

Let τ be a tuple resulting from the SPJ portion of the query, and let τ_i denote its witness in relation R_i ($1 \leq i \leq n$). Each function T_i, termed a *ranking criterion*, assigns a *base score* $T_i(\tau_i) \in [0, 1]$ to the witness τ_i of τ. (We often use $T_i(\tau)$ to denote this base score.) The base scores are combined with the *scoring function* $\mathcal{S}$

Procedure NRJN.$getNext()$
Data structures: priority queue O
Initializations: $t \leftarrow \infty$
begin
1.**while** L is not exhausted **and** (O is empty **OR** $\overline{\mathcal{S}}(O.top()) < t$)
2. $\lambda \leftarrow L.getNext()$
3. Join λ with R and push results in O
4. $t \leftarrow \overline{\mathcal{S}}(\lambda)$
5.**done**
6.**return** $O.pop()$
end

Figure 3: NRJN **Algorithm.**

in order to assign a final score $\mathcal{S}(T_1(\tau_1), \ldots, T_n(\tau_n)) \equiv \mathcal{S}(\tau)$ to τ. The result of the query comprises the top K tuples of the SPJ query, ranked according to their scores. Returning to the example of Figure 1, the ranking criteria are defined as $T_1(hl) = hl.rating$, $T_2(rt) = rt.rating$, $T_3(ev) =$ matches($ev.title$,"music"), and the overall scoring function is a weighted average of the base scores.

Following common practice, we assume that $\mathcal{S}$ is monotonic, i.e., $\mathcal{S}(x_1, x_2, \ldots, x_n) \geq \mathcal{S}(y_1, y_2, \ldots, y_n)$ if $x_i \geq y_i \ \forall 1 \leq i \leq n$. This property is essential in the context of ranking query processing as it allows the computation of upper bounds on the scores of final tuples. More specifically, let τ' be a partial result tuple that is generated by processing a subset of relations, say, $R_1, \ldots, R_m$, and let τ be a complete result tuple that results from processing τ' further with the remaining relations. Clearly, it holds that $T_i(\tau) = T_i(\tau')$ for the corresponding criteria $T_1, \ldots, T_m$, while $T_i(\tau) \leq 1$ for the remaining criteria $T_{m+1}, \ldots, T_n$. By virtue of the monotonicity property of $\mathcal{S}$, we can use $\overline{\mathcal{S}}(\tau') = \mathcal{S}(T_1(\tau'), \ldots, T_m(\tau'), 1, \ldots, 1)$ as an upper bound on the result score $\mathcal{S}(\tau)$, i.e., $\mathcal{S}(\tau) \leq \overline{\mathcal{S}}(\tau')$. We note that it is possible to compute a tighter bound if we substitute the missing scores with the actual maxima of functions T_i, $i = m+1, \ldots, n$, but we omit this optimization to simplify the presentation.

Query Processing. Recent studies [6, 7] have introduced specialized *rank join* physical operators that enable a physical plan to integrate top-K selection with join processing. As mentioned earlier, the main idea is to generate the desired results by accessing only the high-scoring tuples of the input tables. In this paper, we focus on two well known rank join operators that represent the state-of-the-art: *Nested Loops Rank Join* (NRJN) [6], and *Hash Rank Join* (HRJN) [6]. The next paragraphs provide a brief overview of the two operators.

Figure 3 shows the pseudo-code for the NRJN operator. The implementation uses the well known iterator interface [5]. The operator accesses tuples from L in descending order of the upper bound $\overline{\mathcal{S}}$ and generates the result of $L \bowtie R$ (via successive calls to the $getNext()$ method) also in descending order of $\overline{\mathcal{S}}$. The operator works similar to a nested loops join, reading a tuple from L and joining it with all the tuples in R. One important difference is that the results are not returned immediately to the parent operator, but instead are buffered in a priority queue O based on their $\overline{\mathcal{S}}$ bound. The algorithm also maintains a bound t (termed the threshold) on the score of any result tuple that can be generated by accessing more tuples from L. A call to NRJN.$getNext()$ returns the top tuple ω in the output buffer O if $\overline{\mathcal{S}}(\omega) \geq t$, i.e., if it is guaranteed that the operator cannot generate a result with a higher score by accessing more tuples in L. Otherwise, NRJN generates more join results in the output buffer until the condition on the threshold is met. The threshold is set to $t \equiv \overline{\mathcal{S}}(\lambda)$, λ being the last accessed tuple, based on the property that all tuples after λ have an upper bound less than or equal to $\overline{\mathcal{S}}(\lambda)$.

Procedure HRJN$\langle P \rangle.getNext()$
Operator parameter: pull strategy P
Data structures: priority queue O; hash tables HT_L and HT_R
Initializations: $t \leftarrow \infty$;$\lambda \leftarrow \emptyset$; $\rho \leftarrow \emptyset$
begin
1.**while** inputs are not exhausted **AND** (O is empty **OR** $\overline{\mathcal{S}}(O.top()) < t$)
2. $choice \leftarrow P.chooseInput(L, R, \lambda, \rho)$
3. **if** ($choice = L$) **then**
4. $\lambda \leftarrow L.\text{next}()$
5. Join λ with HT_R and push results in O
6. Insert λ in HT_L
7. **else**
8. Perform symmetric actions based on R
9. **endif**
10. **if** $\rho \neq \emptyset$ **AND** $\lambda \neq \emptyset$ **then** $t \leftarrow max(\overline{\mathcal{S}}(\lambda), \overline{\mathcal{S}}(\rho))$
11.**end while**
12.**return** $O.pop()$
end

Figure 4: HRJN **Algorithm.**

Figure 4 depicts the pseudo-code for the HRJN operator. HRJN is an abstract operator that is instantiated by a parameter P, which we call a *pull strategy*. The role of the pull strategy will be described shortly. The idea of HRJN is similar to a symmetric hash-join as it uses two hash tables HT_L and HT_R to hold the tuples accessed from the child operators. One difference is that both L and R are accessed in descending order of $\overline{\mathcal{S}}$. Each time an input tuple is read, the pull strategy determines whether the tuple comes from L or R, possibly taking the scores of the previously pulled tuples into account. As was the case with NRJN, the output is buffered to a priority queue O and the top tuple is returned only if it satisfies the threshold condition. The difference is that the threshold is defined based on both inputs as $t = \max(\overline{\mathcal{S}}(\lambda), \overline{\mathcal{S}}(\rho))$, where λ and ρ are the most recent tuples accessed from L and R respectively. Thus, compared to NRJN, HRJN places more assumptions on its inputs (both have to be accessed in order of $\overline{\mathcal{S}}$), but it will access a prefix of R instead of performing repeated scans.

As mentioned earlier, rank join operators allow the composition of physical plans that generate the top K results without exhausting their inputs. Figure 2 shows a sample physical plan with two hash rank join operators for the query of Figure 1. The top operator Op_1 accesses the output of Op_2 in descending order of $\overline{\mathcal{S}}(hl \cdot rt) \equiv \mathcal{S}(T_1(hl), T_2(rt), 1)$ and the *Event* table in descending order of $\overline{\mathcal{S}}(ev) \equiv \mathcal{S}(1, 1, T_3(ev))$, and generates output tuples $hl \cdot rt \cdot ev$ in descending order of $\overline{\mathcal{S}}(hl \cdot rt \cdot ev) \equiv \mathcal{S}(T_1(hl), T_2(rt), T_3(ev))$. Thus, the result of the ranking query is formed by performing K calls to $Op_1.getNext()$. Operator Op_2 works similarly to Op_1, accessing its inputs *Hotel* and *Restaurant* in descending order of $\overline{\mathcal{S}}(hl) \equiv \mathcal{S}(T_1(hl), 1, 1)$ and $\overline{\mathcal{S}}(rt) \equiv \mathcal{S}(1, T_2(rt), 1)$ respectively, and generating joined tuples $hl \cdot rt$ in decreasing order of $\overline{\mathcal{S}}(hl \cdot rt) \equiv \mathcal{S}(T_1(hl), T_2(rt), 1)$. The main difference is that the pull requests to Op_2 are controlled by operator Op_1 and their total count is thus not necessarily equal to K. Overall, the physical plan generates the answers by accessing only a "prefix" of each input table R_i that contains the high scoring tuples from criterion T_i. This is illustrated in Figure 2 as the shaded regions over the rectangles that represent the complete relations.

2.2 Depth Estimation: Problem Statement

We use the term *input depth* to refer to the number of tuples that a rank join operator accesses from a ranked input in order to answer all the $getNext$ requests of its parent operator. (Thus, NRJN has a left depth only as L is its only ranked input.) In what follows,

we use l and r to denote the left and right input depth respectively of a rank join operator, and $L[l]$ and $R[r]$ to denote the prefixes of L and R respectively up to the specific depths. The notion of input depth extends also to selection operators that consume and return results in sorted order of $\overline{S}$. As an example, consider again the physical plan of Figure 2, and assume that there is a selection operator *cuisine*="Italian" over the access method for *Restaurant*[1]. The selection operator will still access *Restaurant* in sorted order of $\overline{S}$ but it will return only tuples that match the predicate. The depth of the selection is defined as the number of accessed *Restaurant* tuples in order to satisfy r requests to $getNext()$, where r is the right depth of the parent join operator Op_2. In general, the input depth of the selection operator depends on the correlation between the base scores and attribute values of *Restaurants*.

The input depths of rank join operators directly affect the cost of a physical plan. First and foremost, the depths determine how many requests are issued at the access methods of the input tables and thus determine the I/O cost of the plan. Moreover, the depths can characterize the memory requirements of the rank join operators during execution and thus provide some indication of the expected memory footprint of the plan. To illustrate this idea, consider an HRJN operator with depths l and r. These depths indicate exactly the size of the two hash tables HT_L and HT_R, and enable an upper bound on the size of the output buffer as $|L[l] \bowtie R[r]|$. Similar observations can be made for an NRJN operator, where the size of the output buffer can be bound as $|L[l] \bowtie R|$.

As the previous discussion indicates, the ability of the optimizer to cost a rank-aware physical plan, and thus to integrate rank join operators in query compilation, hinges upon the existence of an estimation framework that can approximate the input depths for every operator in the plan. This is precisely the problem that we tackle in this paper.

Depth Estimation Given a physical plan P that consists of HRJN, NRJN, and selection operators, estimate the input depth(s) of any operator in the plan.

As mentioned in Section 1, depth estimation has unique features that set it apart from conventional selectivity estimation and thus render existing estimation techniques ineffective for ranking plans. A major difference is that depth estimation concerns the number of accessed *input* tuples, whereas conventional selectivity estimation concerns the number of *output* tuples. Another distinguishing feature is that rank join algorithms are closely coupled in a pipelined physical plan, since the input depths of an operator depend on the number of $getNext()$ requests from the parent operator. This feature is absent in conventional selectivity estimation, where the output size of an operator depends only on the statistical characteristics of its input tables. These points, raised also by previous studies on ranking query processing [7, 8, 11], motivate the development of new estimation techniques that are tailored to the problem of depth estimation.

3. THE DEEP ESTIMATION FRAMEWORK

In this section, we introduce the proposed DEEP framework for estimating the input depths of operators in a rank-aware physical plan. The distinguishing feature of DEEP is that it employs a principled methodology in order to link depth computation to the distribution of scores in the inputs. This approach results in a systematic framework that takes directly into account the characteristics of the data distribution and thus yields accurate approximations.

[1]If the selection is pushed in the access method, we may model it as a separate operator in order to determine the plan cost.

Procedure $chooseInput(L, R, \lambda, \rho)$
Input: The inputs L and R; the corresponding last accessed tuples λ and ρ.
Output: Indicator L or R.
begin
1.**if** $\rho = \emptyset$ **then return** R
2.**else if** $\lambda = \emptyset$ **then return** L
3.**end if**
4.**if** $\overline{S}(\lambda) > \overline{S}(\rho)$ **then return** L
5.**else if** $\overline{S}(\lambda) < \overline{S}(\rho)$ **then return** R
6.**else if** left depth $<$ right depth **then return** L
7.**else return** R
8.**end if**
end

Figure 5: Pull strategy for HRJN*.

One complicating factor in the estimation problem is that the depths of an HRJN operator are directly affected by the choice of the pulling strategy (Figure 4). Since it is clearly impractical to design an estimation framework for all possible strategies, this leads to the problem of determining a "good" set of supported pulling strategies. To address this issue, we have performed a theoretical analysis of pulling strategies for HRJN and we have identified that a particular variant, termed the *threshold-adaptive* strategy [6], has several desirable properties. The pseudo-code of the strategy is shown in Figure 5. The main idea is to pull from the input with the highest upper bound $\overline{S}$ in an attempt to lower the overall threshold t and thus allow the operator to return the top buffered result without accessing more of its inputs. The instantiation of HRJN with this strategy is commonly referred to as HRJN*. Our analysis shows the optimality of HRJN* in several scenarios and corroborates previous empirical studies that have verified the good performance of HRJN*. (Section 5 provides a more detailed discussion.) Overall, these results provide strong evidence for the use of HRJN* as the default HRJN realization, and essentially allow us to focus DEEP on this particular pulling strategy. We note that we can easily extend our techniques to the variant of HRJN* that accesses tuples in batches, and also to pulling strategies that alternate deterministically between the two inputs. We discuss these extensions further in Section 3.4.

The following sections describe the details of the proposed DEEP estimation framework. To simplify exposition, we assume that the physical plan consists only of HRJN* and NRJN operators, i.e., we exclude selections for now. We extend our framework to selections in Section 3.4. We also restrict the plan to binary join operators, as the extension to multi-way operators is straightforward.

3.1 Modeling Data Statistics

Consider a join of $m \geq 1$ relations $R_1 \bowtie \cdots \bowtie R_m$ where each relation R_i has a ranking criterion T_i. We use $T_i[R_i]$ to denote the set of base scores that T_i assigns to tuples in R_i. Let $\mathbf{b} = (b_1, \ldots, b_m)$ be a combination of base scores such that $b_i \in T_i[R_i]$ for all i. We model the distribution of scores with a *frequency score tensor* F, where $F(\mathbf{b})$ is defined as the number of join results with the combination of base scores $\mathbf{b}$.

DEEP relies on the aforementioned score frequency tensors in order to compute depth estimates. This approach lends some nice properties to the estimation methodology. More concretely, observe that a frequency score tensor takes into account the joint distribution of scores and joins, and hence DEEP does not inherently assume any independence between attributes. Moreover, the use of tensors allows us to decouple the estimation methodology from the details of particular ranking criteria. In other words, DEEP can support any ranking criterion as long as the database system can

provide this information on the distribution of base scores. Finally, note that tensors describe only the base scores and join attributes; we do not assume any statistics or properties for the overall scoring function, except that it is monotonic.

In practice, we expect the system to provide these tensor-based statistics using data synopses such as histograms, wavelets, or sampling. This means that the available information about score distributions may not be totally accurate. In particular, (1) the system may only have approximations of the true frequencies given by F, and (2) the precise sets $T_i[R_i]$ of assigned scores may not be known. From this point onward, we allow for these inaccuracies whenever we refer to a frequency tensor or a set of assigned base scores.

The algorithms of DEEP use a specific interface to access frequency information. The interface comprises the following two methods, which we define using the above notation:

$getFreq(F, \mathbf{b})$ This method simply returns the frequency of $\mathbf{b}$ that is indicated by F.

$nextScore(F, \mathbf{b}, i)$ This method returns the next score below b_i in the i-th dimension of the domain of F. In other words, this is the maximum score assigned by T_i that is less than b_i.

These methods provide a simple and clean interface for the integration of DEEP on top of an existing query optimizer. We show how this is possible in Section 3.3, where we discuss the implementation of this interface using common summarization techniques. Moreover, this interface allows the system to optimize the computation of score frequency information, as it is possible to compute the requested statistics on-the-fly without materializing the tensors. For instance, a call to $getFreq(F, \mathbf{b})$ can be evaluated by combining frequency tensors of individual relations with statistical information on the distribution of join keys in the relations. This approach can lead to a very efficient estimator, since DEEP typically uses only a few cells from each tensor.

3.2 DEEP Estimation

We now introduce the details of the DEEP estimation methodology. DEEP provides a simple interface that takes a query along with relevant statistics, and produces depth estimates for the inputs of each operator. This interface is enabled by a central algorithm, ESTIMATEDEPTHS, that computes estimates of the left and right input depths for a specific join operator given the number of pull requests from the parent operator. We denote an invocation of the algorithm as ESTIMATEDEPTHS(F_L, F_R, F_O, d), where F_L and F_R are the score frequency tensors of the left and right input respectively, F_O is the frequency tensor of the output, and d the number of pull requests. DEEP derives depth estimates on all the inputs of a specific physical plan by applying ESTIMATEDEPTHS at each operator in a top-down traversal of the plan structure. We illustrate this process with the left-deep physical plan (*Hotel* $\bowtie$ *Restaurant*) $\bowtie$ *Event* of Figure 2. DEEP considers first the top operator and invokes ESTIMATEDEPTHS$(F_{HR}, F_E, F_{HRE}, k)$, where: F_{HR} is the frequency tensor of *Hotel* $\bowtie$ *Restaurant*; F_E is the frequency tensor of *Event*; F_{HRE} is the frequency tensor of the three-way join; and, K is the number of query results. This call provides estimates l_{HR} and r_E on the left and right depth respectively of the top operator. The resulting estimate l_{HR} is subsequently used as the number of pull requests for the bottom join operator, which leads to a second invocation ESTIMATEDEPTHS$(F_H, F_R, F_{HR}, l_{HR})$, where F_H and F_R are the score frequency tensors of relations *Hotel* and *Restaurant* respectively. This second call provides estimates on the depths of relations *Hotel* and *Restaurant* and thus concludes estimation. We note that this top-down approach is inherent in ranking

Procedure ESTIMATEDEPTHS(F_L,F_R,F_O,d)
Input: Frequency tensor F_L of the left input L;
Frequency tensor F_R of the right input R;
Frequency tensor F_O of the output O;
count d of pull requests from parent operator.
Output: Estimates l and r for left and right input depths.
begin
1.$\overline{S}(\omega_d)$ = TERMSCOREESTIMATE(F_O, d)
2.$[l^B, l^W]$ = LEFTDEPTHESTIMATE$(F_L, \overline{S}(\omega_d))$
3.$[r^B, r^W]$ = RIGHTDEPTHESTIMATE$(F_R, \overline{S}(\omega_d))$ /** for HRJN* **/
4.$l = (l^B + l^W)/2$
5.$r = (r^B + r^W)/2$ /** Only for HRJN* **/
6.**return** l, r
end

Figure 6: Algorithm ESTIMATEDEPTHS**.**

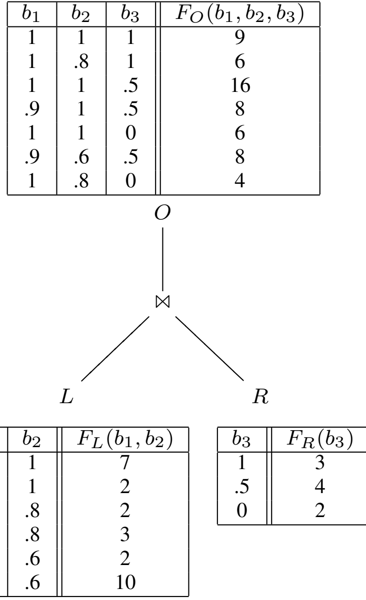
query plans, due to the close coupling of physical operators and the fact that the number of pull requests is known a-priori only for the top operator.

Before describing ESTIMATEDEPTHS in detail, we need to introduce some terminology. Consider a rank join operator with inputs L and R and output O. Recall that the operator generates its output in descending order of the upper bound $\overline{S}$. We use ω_i to denote the i-th output tuple in this ordering. Similarly, we use λ_i to denote the i-th tuple of the left input L when the latter is accessed in descending order of $\overline{S}$, and define ρ_i similarly. (Hence, the notation ρ_i applies only to HRJN*.)

Consider an invocation ESTIMATEDEPTHS(F_L, F_R, F_O, d). The main observation behind our methodology is that the depths of an HRJN* or an NRJN operator are closely tied to the score $\overline{S}(\omega_d)$ of the last result tuple. More concretely, let l^B be defined as the minimum depth in L such that $\overline{S}(\lambda_{l^B}) \leq \overline{S}(\omega_d)$, i.e., the left threshold does not exceed the termination score. Based on the definition of the two operators, it becomes obvious that both HRJN* and NRJN will pull at least up to λ_{l^B}, since it is the first tuple of L that satisfies the termination condition. Similarly, let l^W be defined as the minimum depth in L such that $\overline{S}(\lambda_{l^W}) < \overline{S}(\omega_d)$, i.e., the left threshold falls below the termination score. It can be shown that neither HRJN* nor NRJN will pull past λ_{l^W}. (The proof of this fact is straightforward, and is omitted due to space constraints.) Based on these observations, we can define $[l^B, l^W]$ as a tight range for the left depth l, and similarly define a range $[r^B, r^W]$ for the right depth r in the case of HRJN*. These bounds provide the optimizer with useful statistics on the best- and worst-case cost of an operator, and can also be used to derive an estimate of the depth. One possibility, for instance, is to set the estimates at the midpoint of the ranges, e.g., $l = (l^B + l^W)/2$, in order to capture an average case.

The previous discussion captures the main idea behind the ESTIMATEDEPTHS algorithm: first, it derives an estimate of $\overline{S}(\omega_d)$; then, it uses the estimate on $\overline{S}(\omega_d)$ to compute the ranges $[l^B, l^W]$ and $[r^B, r^W]$ for l and r respectively; finally, it computes estimates of l and r based on the ranges. The corresponding pseudo-code is depicted in Figure 6.

The following sections describe the details of estimating $\overline{S}(\omega_d)$ (algorithm TERMSCOREESTIMATE) and deriving the ranges $[l^B, l^W]$ and $[r^B, r^W]$ (algorithms LEFTDEPTHESTIMATE and RIGHTDEPTHESTIMATE respectively). In each case, we first formalize the computation of the needed information assuming that the complete tensors are known. Subsequently, we present an algorithm that implements the formalism using the selective access interface presented in Section 3.1. Figure 7 shows the running example that we will use in the presentation of our techniques.



b_1	b_2	b_3	$F_O(b_1, b_2, b_3)$
1	1	1	9
1	.8	1	6
1	1	.5	16
.9	1	.5	8
1	1	0	6
.9	.6	.5	8
1	.8	0	4

b_1	b_2	$F_L(b_1, b_2)$
1	1	7
.9	1	2
1	.8	2
.9	.8	3
.9	.6	2
.8	.6	10

b_3	$F_R(b_3)$
1	3
.5	4
0	2

Figure 7: Running example for estimation algorithms.

3.2.1 *Estimating the Termination Score*

In this section, we discuss the estimation of the termination score $\overline{\mathcal{S}}(\omega_d)$ based on the frequency output tensor F_O (line 1 in Figure 6).

Computation. We provide the intuition behind our approach with the computation of $\overline{\mathcal{S}}(\omega_d)$ on the output tensor of Figure 7. For the example, we assume that the score is a simple sum of base scores and the number of pull requests is $d = 10$. We start our computation by listing the possible combinations of base scores in descending order of their score bound:

$\mathbf{b}_i$	$\overline{\mathcal{S}}(\mathbf{b}_i)$	$F_O(\mathbf{b}_i)$
$\mathbf{b}_1 = (1, 1, 1)$	3	9
$\mathbf{b}_2 = (.9, 1, 1)$	2.9	0
$\mathbf{b}_3 = (1, .8, 1)$	2.8	6
$\mathbf{b}_4 = (.8, 1, 1)$	2.8	0
$\mathbf{b}_5 = (.9, .8, 1)$	2.7	0
...		

First, we observe that the 9 highest scores correspond to $\mathbf{b}_1$. Another way to view this is that $\mathbf{b}_1$ "generates" $F_O(\mathbf{b}_1) = 9$ answers $\omega_1, \ldots, \omega_9$ with the score $\overline{\mathcal{S}}(\mathbf{b}_1)$. In a similar manner, $\mathbf{b}_2$ generates $F_O(\mathbf{b}_2) = 0$ answers, and $\mathbf{b}_3$ generates $F_O(\mathbf{b}_3) = 6$ answers $\omega_{10}, \ldots, \omega_{15}$ with the score $\overline{\mathcal{S}}(\mathbf{b}_3)$. We see that $\overline{\mathcal{S}}(\omega_{10})$ is generated by $\mathbf{b}_3$ in this process, and hence we conclude that the termination score is equal to $\overline{\mathcal{S}}(\mathbf{b}_3)$.

The previous example illustrates the crux behind our approach. The main idea is to identify the score combination that suffices to compute at least d result tuples in nonincreasing order of $\overline{\mathcal{S}}$ values. This idea is formalized as follows. Let $\mathbf{b}_1, \ldots, \mathbf{b}_N$ denote the possible combinations of base scores in the domain of F_O, in descending order of their score bound. In other words, our ordering assumes $\overline{\mathcal{S}}(\mathbf{b}_i) \geq \overline{\mathcal{S}}(\mathbf{b}_j)$ for all $i < j$. The goal is to find $\mathbf{b}_x$ such that $\sum_{i=1}^{x-1} F_O(\mathbf{b}_i) < d \leq \sum_{i=1}^{x} F_O(\mathbf{b}_i)$. The termination score can then be computed exactly as $\overline{\mathcal{S}}(\mathbf{b}_x)$. The following theorem provides a formal statement of this methodology:

THEOREM 3.1. *Let $\mathbf{b}_1, \ldots, \mathbf{b}_N$ denote the nonincreasing ordering of base score combinations defined above, and define $x \equiv \min\{j \mid \sum_{i=1}^{j} F_O(\mathbf{b}_i) \geq d\}$. The termination score is computed as $\overline{\mathcal{S}}(\omega_d) = \overline{\mathcal{S}}(\mathbf{b}_x)$.* ∎

One realization of Theorem 3.1 is to sort the elements of F_O and then iterate over the sorted sequence $\mathbf{b}_1, \ldots, \mathbf{b}_N$ until the accumulated frequency is at least d. Clearly, the iteration terminates exactly at the desired combination $\mathbf{b}_x$. This approach, however, implies the materialization of F which can be prohibitively expensive for the stringent time constraints of estimation. Moreover, we expect only a few entries from F to actually contribute to the computation of $\overline{\mathcal{S}}(\omega_d)$, which implies that sorting the entire tensor is likely to be wasted work. To address these issues, we introduce below the TERMSCOREESTIMATE algorithm, that takes advantage of the monotonicity property of $\mathcal{S}$ in order to provide a more efficient realization of Theorem 3.1.

Algorithm TERMSCOREESTIMATE**.** The proposed algorithm uses the monotonicity property of $\mathcal{S}$ in order to efficiently iterate over $\mathbf{b}_1, \ldots, \mathbf{b}_N$ by selectively accessing the underlying tensor. Essentially, TERMSCOREESTIMATE stays true to the spirit of ranking query processing: it reads just "enough" statistics in order to compute its estimate.

Consider a base score b_i in the combination $\mathbf{b} = (b_1, \ldots, b_m)$, and let b_i^- denote the immediately lower score in $T_i[R_i]$. We define the dominated neighbor of $\mathbf{b}$ on dimension i as the score combination $\mathbf{b}'$ that results from substituting b_i with b_i^-. In other words, $\mathbf{b}' = (b'_1, \ldots, b'_m \mid b'_j = b_j \forall j \neq i \wedge b'_i = b_i^-)$. Symmetrically, we call $\mathbf{b}$ a dominant neighbor of $\mathbf{b}'$. The monotonicity of $\mathcal{S}$ guarantees that $\overline{\mathcal{S}}(\mathbf{b}) \geq \overline{\mathcal{S}}(\mathbf{b}')$ for every dominant neighbor $\mathbf{b}$ of $\mathbf{b}'$. In turn, this implies that $\mathbf{b}'$ appears in the ordering $\mathbf{b}_1, \ldots, \mathbf{b}_N$ after all of its dominant neighbors. This is the key property on which TERMSCOREESTIMATE relies in order to simulate the traversal over $\mathbf{b}_1, \ldots, \mathbf{b}_N$.

Procedure TERMSCOREESTIMATE(F_O,d)
Input: Output score frequency tensor F_O; number of pull requests d
Output: Estimate of the termination score $\overline{\mathcal{S}}(\omega_d)$
begin
1.$Q \leftarrow \emptyset$
2.$\mathbf{b} \leftarrow (1, 1, \ldots, 1)$
3.$topCount \leftarrow getFreq(F_O, \mathbf{b})$
4.$exhausted \leftarrow$ **false**
5.**while** $topCount < d$ **and** $exhausted =$ **false do**
6. **for each** $i = 1, \ldots, m$ **do** */** Visit dominated neighbors of* $\mathbf{b}$ ***/*
7. $b_i^- \leftarrow nextScore(F_O, \mathbf{b}, i)$
8. **if** $b_i^- \neq \emptyset$ **then**
9. $\mathbf{b}' \leftarrow (b'_1, \ldots, b'_m \mid b'_j = b_j \forall j \neq i \wedge \ b'_i = b_i^-)$
10. $dc[\mathbf{b}'] \leftarrow dc[\mathbf{b}'] + 1$
11. **if** $dc[\mathbf{b}'] = nb(\mathbf{b}')$ **then** $Q.push(\mathbf{b}')$ **end if**
12. **end if**
13. **end for**
14. **if** $Q \neq \emptyset$ **then** */** Retrieve next combination **/*
15. $\mathbf{b} \leftarrow Q.pop()$
16. $topCount \mathrel{+}= getFreq(F_O, \mathbf{b})$
17. **else**
18. $exhausted =$ **true**
19. **endif**
20.**end while**
21.**return** $\overline{\mathcal{S}}(\mathbf{b})$
end

Figure 8: Algorithm TERMSCOREESTIMATE**.**

The pseudo-code for TERMSCOREESTIMATE is shown in Figure 8. The algorithm maintains two data structures: a hash table dc that maps each score combination to the count of its dominant neighbors that have been processed by the algorithm, and a priority

Queue Q

$\mathbf{b}$	$\overline{\mathcal{S}}(\mathbf{b})$	$F_O(\mathbf{b})$
$(.9, 1, 1)$	2.9	0
$(1, .8, 1)$	2.8	6
$(1, 1, .5)$	2.5	16

$\mathbf{b} = \mathbf{b}_1 = (1, 1, 1)$

$topCount = 9$

(a) After processing $\mathbf{b}_1$

Queue Q

$\mathbf{b}$	$\overline{\mathcal{S}}(\mathbf{b})$	$F_O(\mathbf{b})$
$(1, .8, 1)$	2.8	6
$(1, 1, .5)$	2.5	16

$\mathbf{b} = \mathbf{b}_2 = (.9, 1, 1)$

$topCount = 9$

(b) $\mathbf{b}_2$ is popped from Q

Queue Q

$\mathbf{b}$	$\overline{\mathcal{S}}(\mathbf{b})$	$F_O(\mathbf{b})$
$(1, .8, 1)$	2.8	6
$(.8, 1, 1)$	2.8	0
$(1, 1, .5)$	2.5	16

$\mathbf{b} = \mathbf{b}_2 = (.9, 1, 1)$

$topCount = 9$

(c) After processing $\mathbf{b}_2$

Queue Q

$\mathbf{b}$	$\overline{\mathcal{S}}(\mathbf{b})$	$F_O(\mathbf{b})$
$(.8, 1, 1)$	2.8	0
$(1, 1, .5)$	2.5	16

$\mathbf{b} = \mathbf{b}_3 = (1, .8, 1)$

$topCount = 15$

(d) $\mathbf{b}_3$ is popped from Q

Figure 9: Example run of TERMSCOREESTIMATE**.**

queue Q that stores combinations $\mathbf{b}$ prioritized based on $\overline{\mathcal{S}}(\mathbf{b})$ first and $F_O(\mathbf{b})$ second. The pseudo-code also uses $nb(\mathbf{b})$ to denote the total number of dominant neighbors of $\mathbf{b}$. (We note that $nb(\mathbf{b})$ can be computed easily as the number of scores in $\mathbf{b}$ that are not equal to the top score, that is, $nb(\mathbf{b}) = |\{i \mid b_i \neq 1\}|$.) The algorithm proceeds in iterations until the condition of Theorem 3.1 is satisfied or tensor F is exhausted. At the beginning of the n-th iteration, the algorithm has already accessed combinations $\mathbf{b}_1, \ldots, \mathbf{b}_n$ and their frequencies have been aggregated in variable $topCount$. Variable $\mathbf{b}$ tracks the most recent combination $\mathbf{b}_n$. The first part of the iteration (lines 6–13) examines the current combination $\mathbf{b}_n$, and increases the count $dc[\mathbf{b}']$ for every dominated neighbor $\mathbf{b}'$ of $\mathbf{b}_n$. If $dc[\mathbf{b}']$ becomes equal to the total number of dominant neighbors $nb(\mathbf{b}')$, this implies that the prefix $\mathbf{b}_1, \ldots, \mathbf{b}_n$ contains all the combinations that dominate $\mathbf{b}'$; hence, $\mathbf{b}'$ becomes a candidate for the next score combination $\mathbf{b}_{n+1}$ and is inserted in Q. Once these steps are completed for all dominated neighbors, the top element in Q is guaranteed to be the next combination $\mathbf{b}_{n+1}$. The algorithm thus retrieves this element, updates $topCount$ accordingly, and starts a new iteration.

EXAMPLE 3.1.: *Consider the application of our algorithm on tensor F_O of Figure 7. At iteration 1, the algorithm examines $\mathbf{b}_1 = (1, 1, 1)$ and increments the dc counts of dominated neighbors $(.9, 1, 1)$, $(1, .8, 1)$, and $(1, 1, .5)$. Since these combinations have $\mathbf{b}_1$ only as their neighbor, they are inserted immediately in Q (Figure 9(a)). Subsequently, $(.9, 1, 1)$ is identified as $\mathbf{b}_2$ and topcount is increased to $F_O(\mathbf{b}_1) + F_O(\mathbf{b}_2)$ (Figure 9(b)).*

The second iteration examines $\mathbf{b}_2 = (.9, 1, 1)$ and increases the dc counts of neighbors $(.8, 1, 1)$, $(.9, .8, 1)$, and $(.9, 1, .5)$. In this case, only $(.8, 1, 1)$ is entirely covered by the prefix $\mathbf{b}_1, \mathbf{b}_2$ and is thus inserted in Q (Figure 9(c)). Combination $(.9, .8, 1)$ is dominated by $(1, .8, 1)$ which is still in Q, and hence cannot be a candidate for the next combination $\mathbf{b}_3$. The same holds for $(.9, 1., 5)$ that is dominated by $(1, 1, .5)$.

The third iteration selects $(1, .8, 1)$ as $\mathbf{b}_3$ and the total count becomes $topCount = F_O(\mathbf{b}_1) + F_O(\mathbf{b}_2) + F_O(\mathbf{b}_3) = 15 \geq 10$ (Figure 9(d)). At this point, the algorithm stops and correctly returns $\overline{\mathcal{S}}(1, .8, 1) = 2.8$ as the termination score. ∎

The running time of TERMSCOREESTIMATE depends heavily on the implementation of the underlying tensor, but each individual tensor operation is typically fast enough to be considered constant time. In addition, the dimension m of the base score vectors is normally a small constant. With these assumptions, the running time is dominated by the priority queue operations. In the worst case, each of the N score combinations in the domain of F_O needs to be pushed and popped once. Each of these operations takes logarithmic time, implying a cost of $O(N \log N)$. This worst case is essentially equivalent to the naive approach of sorting the cells of the tensor.

It is useful to reiterate that TERMSCOREESTIMATE relies on the access interface defined in Section 3.1 in order to retrieve information from tensor F_O. As mentioned earlier, this interface allows the system to compute the requested information on-the-fly, without needing to materialize the tensor F_O. We revisit this point in more detail in Section 3.3, where we discuss the implementation of DEEP on top of common summarization techniques such as histograms [15].

3.2.2 Estimating Depth Bounds

Having the means to compute $\overline{\mathcal{S}}(\omega_d)$, we now switch our attention to the computation of the ranges $[l^B, l^W]$, and $[r^B, r^W]$ for l and r respectively (lines 2–3 in Figure 6). In the interest of space, we only discuss the derivation of $[l^B, l^W]$, as the case for $[r^B, r^W]$ is completely symmetrical. Without loss of generality, we assume that the left input tensor F_L is defined on the domain $T_1[R_1] \times \cdots \times T_{m'}[R_{m'}]$ for some $m' < m$.

Computation. We first provide the intuition behind our approach with the computation of l^B and l^W over the left tensor F_L of Figure 7. Recall that our running example uses a sum of base scores as the scoring function. This means that the score bound is computed as $\overline{\mathcal{S}}(b_1, b_2) = \mathcal{S}(b_1, b_2, 1) = b_1 + b_2 + 1$. As in the previous section, the first step is to list the possible base score combinations of F_L in descending order of their score bounds:

$\mathbf{b}_i$	$\overline{\mathcal{S}}(\mathbf{b}_i)$	$F_L(\mathbf{b}_i)$
$\mathbf{b}_1 = (1, 1)$	3	7
$\mathbf{b}_2 = (.9, 1)$	2.9	2
$\mathbf{b}_3 = (1, .8)$	2.8	2
$\mathbf{b}_4 = (.9, .8)$	2.7	3
...		

Similar to our methodology in the previous section, we can view each combination as a "generator" of the tuples in input L. For instance, $\mathbf{b}_1$ generates $F_L(\mathbf{b}_1) = 7$ tuples in L with score $\overline{\mathcal{S}}(\mathbf{b}_1)$, $\mathbf{b}_2$ generates $F_L(\mathbf{b}_2) = 2$ more tuples with score $\overline{\mathcal{S}}(\mathbf{b}_2)$, and so on. We note that $\mathbf{b}_3$ is the first combination $\mathbf{b}_i$ such that $\overline{\mathcal{S}}(\mathbf{b}_i) \leq \overline{\mathcal{S}}(\omega_d) = 2.8$. Thus, the first tuple in the group generated by $\mathbf{b}_3$ corresponds to the best-case input depth l^B, i.e., $l^B = F_L(\mathbf{b}_1) + F_L(\mathbf{b}_2) + 1 = 9$. Similarly, $\mathbf{b}_4$ is the first combination $\mathbf{b}_i$ such that $\overline{\mathcal{S}}(\mathbf{b}_i) < \overline{\mathcal{S}}(\omega_d)$, and hence there are exactly $F_L(\mathbf{b}_1) + F_L(\mathbf{b}_2) + F_L(\mathbf{b}_3)$ tuples whose score is not less than $\omega(d)$. Using the same argument as before, this implies that $l^W = F_L(\mathbf{b}_1) + F_L(\mathbf{b}_2) + F_L(\mathbf{b}_3) + 1 = 12$.

As illustrated in the previous example, the main idea behind our approach is to iterate over the score combinations $\mathbf{b}_1, \ldots, \mathbf{b}_{N_L}$ in descending order of their corresponding bounds and identify

```
Procedure LEFTDEPTHESTIMATE(F_L, S_term)
Input: Score frequency tensor F_L; termination score S_term.
Output: Bounds l^B and l^W.
begin
1.Q ← ∅; exhausted ← false
2.b ← (1, 1, ..., 1)
3.l ← 0; l^B ← 0; l^W ← 0
4.if S̄(b) = S_term then l^B = 1 end if
5.while S̄(b) ≥ S_term and exhausted = false do
6.   l += getFreq(F_L, b)
7.   for each i = 1, ..., m' do    /** Visit dominated neighbors of b **/
8.      b_i^- ← nextScore(F_L, b, i)
9.      if b_i^- ≠ ∅ then
10.        b' ← (b'_1, ..., b'_{m'} | b'_j = b_j ∀j ≠ i ∧ b'_i = b_i^-)
11.        dc[b'] ← dc[b'] + 1
12.        if dc[b'] = nb(b') then Q.push(b') end if
13.     end if
14.  end for
15.  if Q ≠ ∅ then   /** Retrieve next combination **/
16.     b ← Q.pop()
17.     if S̄(b) ≤ S_term and l^B = 0 then l^B ← l + 1 end if
18.     if S̄(b) < S_term then l^W ← l + 1 end if
19.  else
20.     exhausted = true; l^W = l
21.  end if
22.end while
23.return l^B, l^W
end
```

Figure 10: Algorithm LEFTDEPTHESTIMATE

the first combinations $\mathbf{b}_y$ and $\mathbf{b}_z$ such that $\overline{\mathcal{S}}(\mathbf{b}_y) \leq \overline{\mathcal{S}}(\omega_d)$ and $\overline{\mathcal{S}}(\mathbf{b}_z) < \overline{\mathcal{S}}(\omega_d)$ respectively. The bounds l^B and l^W can then be computed in terms of the accumulated frequency up to those combinations. More formally, we have the following result:

THEOREM 3.2. *Let $\mathbf{b}_1, \ldots, \mathbf{b}_{N_L}$ be defined as previously. If $\overline{\mathcal{S}}(\mathbf{b}_{N_L})$ is less than the termination score $\overline{\mathcal{S}}(\omega_d)$, then the bounds l^B and l^W are computed as follows:*

$$l^B = \min \left\{ \sum_{1 \leq i < y} F_L(\mathbf{b}_i) \;\middle|\; 1 \leq y \leq N_L \wedge \overline{\mathcal{S}}(\mathbf{b}_y) \leq \overline{\mathcal{S}}(\omega_d) \right\} + 1$$

$$l^W = \min \left\{ \sum_{1 \leq i < z} F_L(\mathbf{b}_i) \;\middle|\; 1 \leq z \leq N_L \wedge \overline{\mathcal{S}}(\mathbf{b}_z) < \overline{\mathcal{S}}(\omega_d) \right\} + 1$$

If $\overline{\mathcal{S}}(\mathbf{b}_{N_L}) = \overline{\mathcal{S}}(\omega_d)$, then l^B remains the same, but l^W is computed as $l^W = \sum_{i=1}^{N_L} F_L(\mathbf{b}_i)$. ∎

Algorithm LEFTDEPTHESTIMATE**.** Similar to the estimation of $\overline{\mathcal{S}}(\omega_d)$, a naive implementation of the previous formalization can prove prohibitively expensive for the stringent (time and space) requirements of query optimization. To address this issue, we introduce the LEFTDEPTHESTIMATE algorithm that computes these bounds by accessing selective parts of the tensor.

Figure 10 shows the pseudo-code for LEFTDEPTHESTIMATE. The algorithm relies on the same general principle as TERMSCOREESTIMATE in order to perform the ordered traversal over $\mathbf{b}_1, \ldots, \mathbf{b}_{N_L}$. The same data structures are used, namely, a hash table dc for counts of dominant neighbors that have been processed, and a priority queue Q of score combinations in decreasing order of $\overline{\mathcal{S}}$. (Ties are broken again in decreasing order of frequency.) Also recall that $nb(\mathbf{b})$ denotes the number of dominant neighbors of $\mathbf{b}$. Each iteration (lines 6–21) proceeds in two stages. The first stage (lines 7–14) increments the dc counts for the dominated neighbors of the current combination $\mathbf{b}_n$, and updates Q with combinations whose dominant neighbors all appear in the prefix $\mathbf{b}_1, \ldots, \mathbf{b}_n$. The second stage (lines 15–21) retrieves the next element $\mathbf{b}_{n+1}$ from Q, and updates l^B and/or l^W if $\overline{\mathcal{S}}(\mathbf{b}_{n+1})$ satisfies the corresponding inequality with $\mathcal{S}_{\text{term}}$. The algorithm terminates once l^W has been set, or if the input tensor is exhausted.

Similar to TERMSCOREESTIMATE, the LEFTDEPTHESTIMATE algorithm uses the two-method interface (Section 3.1) to access the input tensor F_L, and thus does not assume the tensor is materialized. Another similarity is that the worst-case cost of the algorithm is equivalent to the cost of sorting the cells of the input tensor.

3.3 Integrating DEEP in an Optimizer

In this section, we discuss the integration of DEEP in a query optimizer, which essentially amounts to the implementation of the *getFreq*/*nextScore* interface for accessing score frequency tensors. We show that it is possible to implement this interface efficiently for a wide range of ranking criteria, using information that is commonly available from the data statistics of the optimizer. This provides strong evidence for the practicality of DEEP as a depth estimation framework.

To make the presentation more concrete, we henceforth assume that the optimizer uses multi-dimensional histograms [15] to store data statistics. The methodology extends naturally to other techniques such as wavelet summaries [2, 14] or samples [1]. Consider a base table R_i with an attribute set $\mathcal{A}_i = \{A_1, \ldots, A_{n_i}\}$ and let $Dom(A_k)$ denote the value domain of attribute $A_k \in \mathcal{A}_i$. Given a combination of values $\mathbf{v} \in Dom(A_1) \times \cdots \times Dom(A_{n_i})$, we use $Fr_i(\mathbf{v})$ to denote the frequency of tuples in R_i that match $\mathbf{v}$, so R_i comprises the combinations with non-zero frequency. A histogram H_i provides an approximation of the frequency distribution Fr_i by partitioning the domain $Dom(A_1) \times \cdots \times Dom(A_{n_i})$ in hyper-rectangles, termed buckets, and storing aggregate statistical information per bucket. We will use $H_i(\mathbf{v})$ to denote the histogram-provided frequency of combination $\mathbf{v}$ and $\mathcal{V}_i$ for the set of combinations such that $H_i(\mathbf{v}) > 0$. We extend this notation to $\mathcal{V}_i(A_k)$ for the projection of $\mathcal{V}_i$ on the attribute A_k. The set $\mathcal{V}_i$ provides an approximation of the distinct values in R_i, since H_i performs some approximation of the values present in R_i. One such approximation, for instance, is through the uniform spread assumption [15].

In what follows, we describe a mechanism that uses H_i to implement access to the frequency tensor F_i for a ranking criterion T_i. To simplify exposition, we henceforth assume that R_i contains two attributes A_1 and A_2. As the first case in our discussion, we assume that T_i computes the score of a tuple by applying a function on A_1 that is invertible and increasing or decreasing. This represents a frequent case that arises most commonly when T_i ranks tuples in the order given by a specific attribute, e.g. ordering *Restaurant* tuples by decreasing rating or increasing cost. We use the shorthand notation $T_i(v_1)$ to denote the value of T_i on any combination of values with v_1 in the first column, and $T_i^{-1}(b_i)$ to denote the value in $Dom(A_1)$ that T_i maps to b_i. Under these assumptions, we can implement access to the tensor as follows:

$$getFreq(F_i, b_i) = \sum_{v_2 \in \mathcal{V}_i(A_2)} H_i(T_i^{-1}(b_i), v_2)$$

$$nextScore(F_i, b_i, 1) = \begin{cases} T_i(\max\{v_1 \in \mathcal{V}_i(A_1) \mid v_1 < T_i^{-1}(b_i)\}) & \text{if } T_i \text{ is increasing} \\ T_i(\min\{v_1 \in \mathcal{V}_i(A_1) \mid v_1 > T_i^{-1}(b_i)\}) & \text{if } T_i \text{ is decreasing} \end{cases}$$

The *getFreq* method simply returns the approximate frequency of $T_i^{-1}(b_i)$ based on the underlying histogram. When T_i is increasing the *nextScore* method finds the maximum value $v \in \mathcal{V}_i(A_1)$ that

is less than $T_i^{-1}(b_i)$, which yields the maximum score $T_i(v_1) < b_i$ since T_i is increasing. The case when T_i is decreasing is symmetrical. We note that it is possible to take advantage of the specifics of the value-based summarization in order to quickly find the closest approximate value to $T_i^{-1}(b_i)$. For example, if the histogram uses the uniform spread assumption, then the closest value can be computed algebraically without examining the contents of $\mathcal{V}_i(A_1)$. The computation on histograms implies that the scores returned to DEEP are essentially quantized based on the value approximation $\mathcal{V}_i$. This point arises in conventional selectivity estimation as well, and the setting of the quantization is part of the physical tuning of the database.

We observe that we can extend the previous methodology to invertible ranking criteria that are not monotonic, and to ranking criteria that map a limited number of attribute values to the same base score. As an example of the latter, we consider a "distance" criterion that assigns a score based on the distance from some target constant, e.g., $T_i(\mathbf{v}) = |c - v_1|$. Note that each possible base score b_i can come from at most two values of v_1, namely $c - b_i$ and $b_i - c$. We can thus implement the tensor interface by combining the information that H_i yields about these two values, using the same concept as above.

The previously described mechanism provides adequate support for a significant class of practical ranking criteria. For more complex criteria, or for criteria that require more complex statistics than what can be stored in a histogram, we propose a catch-all mechanism that relies on the typical assumptions of uniformity and independence. More concretely, we assume that there are s distinct base scores that are uniformly distributed in $[0, 1]$, independent of attribute values. The value of s is a system parameter and it may be some fixed number, such as 100, or it may vary for different criteria. Under our assumptions, each score value occurs with frequency $\frac{|R_i|}{s}$, where $|R_i|$ denotes the cardinality of the relation, and distinct scores are separated by a gap of $\frac{1}{s-1}$. Hence the computation of the tensor can be expressed as follows: $getFreq(F_i, b_i) = \frac{|R_i|}{s}$, and $nextScore(F_i, b_i, 1) = b_i - \frac{1}{s-1}$. Of course, these assumptions may not be valid on the input data and this will introduce error in the estimation process. A more effective solution requires the development of specialized data synopses that enable the computation of score frequency tensors for complex ranking functions.

Up to this point, we have considered access to the score tensor of a single relation. Access to the tensor of a join output is implemented similarly, by combining the statistics for the single-table tensors. (We note that a similar methodology is employed in relational systems in order to estimate the frequency distribution for the result of a conventional join.) As an example, consider a relation $R_j(A_2, A_3)$ with a ranking criterion T_j that is invertible and computed on attribute A_3. Consider the join $R_i \bowtie R_j$ on attribute A_2. The score tensor F of the join can be computed as follows:

$$getFreq(F, (b_i, b_j)) = \sum_{v_2 \in \mathcal{V}_i(A_2) \cup \mathcal{V}_j(A_2)} H_i(T_i^{-1}(b_i), v_2) \cdot H_j(v_2, T_j^{-1}(b_j))$$

(The $nextScore$ methods remain the same as before.) Again, the key idea is that the optimizer does not need to materialize the intermediate tensor F, but can instead compute entries on-the-fly as they are requested by DEEP. We note that this expression can be optimized further by "aligning" the buckets of the two histograms and performing the computation on a per-bucket basis. (This optimization is similar to techniques for approximate query answering over histograms [10].) We can derive a similar set of expressions for the case where one of the criteria is not invertible (or both of them), by adopting the solution outlined earlier. We do not discuss this further as it is a straightforward extension.

3.4 Extensions

Handling Selections. Up to this point, we have assumed that the physical plan consists solely of HRJN* and NRJN operators. It is possible to apply our techniques to plans that include selection operators as well, by performing a transformation that substitutes each selection operator with an equivalent NRJN join operator. More concretely, consider a selection operator Op_σ with input L and assume that the operator applies a range predicate on an integer attribute. We define a conceptual relation R that contains a single attribute with the values in the selected range. It is straightforward to show that the results of the plan do not change if Op_σ is substituted with an NRJN operator on inputs L and R and a equi-join condition on the predicated attribute. Moreover, the left input depth of the join operator is exactly the input depth of the original operator Op_σ.

Other pulling strategies. We can extend our estimation methodology to the variant of HRJN* that reads tuples in batches, and to a pulling strategy that alternates between the two inputs in a round-robin fashion. The main idea is to compute the depths that HRJN* would pull, and translate these depths for the alternate pulling strategy. We explain how this works for the round-robin version of HRJN (the extension to batch access is simpler). Let $[l^W, l^B]$ and $[r^W, r^B]$ denote the ranges of depths for the left and right input respectively, as computed by DEEP for the HRJN* operator. The round-robin strategy must drop the score threshold down to the terminal score $\overline{\mathcal{S}}(\omega_d)$ on both the left and right. This means that it pulls at least $\max(l^B, r^B)$ from both sides. Using similar logic, we can observe that a worst-case bound on the input depths of both sides is given by $\max(l^W, r^W)$. However, to be precise, these bounds may be off by one depending on which side is pulled first.

4. EXPERIMENTAL STUDY

In this section, we present the results of an extensive experimental study that we have conducted to validate the effectiveness of our estimation framework.

4.1 Methodology

Techniques. We have used the following techniques in our study:

- DEEP. We have completed a prototype implementation of the DEEP framework that we introduce in this paper. Our prototype implements score frequency tensors using TuG synopses [16] as the underlying data summarization technique (see Section 3.3). We have chosen this particular method as it yields reasonably accurate approximations, but, as described in Section 3.3, DEEP is readily portable to other methods as well. The overall space for the data statistics was set to 150KB.

- Probabilistic Estimator. We have implemented the probabilistic estimation methodology introduced by Ilyas et al. [8]. In order to make a fair comparison, we assume that the estimator uses the same underlying 150KB TuG synopses as our DEEP prototype in order to retrieve estimates on join selectivity. The original formulation of the methodology assumes that all relations have the same size and is thus not applicable on the data sets in our study. To overcome this restriction, we have developed a modification of the estimator that lifts this assumption but works only on single-join queries. Extending our modifications to more than one join is non-trivial and beyond the scope of our paper.

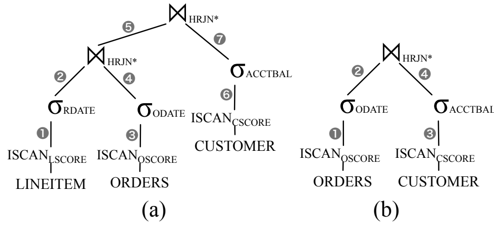


Figure 11: Plan templates for workload generation.

- Sample-Based Estimator. We have implemented the sample-based estimation technique introduced by Li et al. [11]. Each table-sample is created with a 5% sampling rate, resulting in a total of 4.6MB for the samples of all tables. We note that this is significantly larger than the 150KB of statistics that we use for our DEEP implementation. We resorted to this relatively large sample size, since lower sampling rates caused the estimator to severely underestimate join selectivities. To ensure some statistical robustness, we derive a depth estimate by averaging the estimates over 5 such independently created samples.

Data sets. Our data sets are based on a scale 1.0 (1GB), skewed version of the TPC-H data set. We have augmented each relation with a score attribute that serves the role of a ranking criterion on the relation. The values in score attributes follow a Zipfian distribution with the same skew z for all relations. In our experiments, we vary z as 0, 0.5, 1.0, and 1.5, creating data sets that range from uniform score distributions to relatively skewed scores.

Workloads. We generate a workload of physical plans by creating several instantiations of a plan template. In this paper, we report experiments using the two templates shown in Figure 11. (The numbers on the inputs serve as identifiers for easy reference in the experiments.) The templates generate single- and two-join physical plans with selection criteria on all the inputs, and thus model physical plans of varying complexity. We only include HRJN* in the plans, as it is the most complex of the two operators that we consider in our work. We use the sum of attribute scores as the scoring function $\mathcal{S}$. We note that we have performed experiments with other physical plans and have obtained similar results to the previous two templates.

A workload corresponds to a specific template and value of K, and contains 250 matching plans with randomly generated selection criteria. To create meaningful ranking queries, we ensure that each generated plan produces at least 10,000 output tuples if the top-K selection is removed. We vary K as 10, 100, and 1000, thus resulting in six workloads overall.

Evaluation Metric. We use the average absolute relative error of estimation to quantify the accuracy of each technique. More concretely, let d and $\widehat{d}$ denote the true and estimated depth respectively for an input depth. We compute the absolute relative error of the estimate as $ARE(d, \widehat{d}) = |d - \widehat{d}|/max(d, sn)$, where sn is a sanity bound that avoids an inordinately high percentage when d is small. Given a workload, we set sn to the 10th percentile of actual depths[2] and we report the average estimation error per input in the corresponding templates (see Figure 11).

In addition to the average, we employ the Cumulative Distribution Frequency (CDF) of the ARE as a more detailed metric of accuracy. The CDF consists of pairs (x, y) that are interpreted as follows: y% of estimates have an ARE that is less than or equal to x%. Thus, when comparing two techniques, a dominating curve denotes higher estimation accuracy for a larger part of the workload.

[2]Hence, 90% of the considered true depths are above the sanity bound.

4.2 Experimental Results

In this section, we present a subset of the most representative results of our study. Unless otherwise noted, we use the following parameters in our experiments: $K = 10$ and $z = 1.5$.

The Effect of Skew. The goal of this experiment is to evaluate the effectiveness of the three techniques relative to the skew in the base scores. We vary parameter z as 0, 0.5, 1.0, and 1.5, and measure the estimation error against a workload generated by the single-join template. We report the error on input number 1, as the results for other inputs are similar.

Figure 12(a) shows the estimation error of the three techniques as a function of skew in the base scores. (In all plots, we use a logarithmic scale for the y-axis and also print the magnitude of the error above each bar.) As the results demonstrate, DEEP yields accurate estimates for all values of z and consistently outperforms the other techniques. For $z = 1.5$, for instance, DEEP has a low estimation error of 2%, compared to 44% for the probabilistic method and 342% for the sample-based estimate. This difference in error can be explained as follows. DEEP takes directly into account the distribution of scores in the base tables, and thus remains accurate as long as the optimization statistics provide a good model of the score distribution. Conversely, the other two techniques fail to capture accurately the distribution of scores and this inevitably leads to high errors. More concretely, the sample-based estimator tends to underestimate the selectivity of the join, and in turn overestimates the number of tuples that need to be joined in order to generate the top K results. (We note that this type of underestimation is common for sample-based estimators.) The probabilistic estimator is based on an analysis that assumes uniformly distributed scores, and thus its accuracy quickly deteriorates as the actual score distribution deviates from uniform scores. This is shown more clearly in Figure 12(b), which charts the CDF of the estimation error of the probabilistic estimator for $z = 0$ and $z = 1.0$. For comparison, we also plot the estimation error for DEEP. (Recall that DEEP and the probabilistic estimator utilize the same underlying data statistics in all cases.) While the two techniques have essentially the same accuracy for $z = 0$, the performance of the probabilistic estimator drops significantly when the scores are not uniform ($z = 1.0$). DEEP, on the other hand, actually improves in accuracy over $z = 0$, since it does not make a-priori assumptions on the distribution of scores.

We henceforth focus our analysis on data sets with $z > 0$, since skewed distributions are common in real-life data. As a result, we exclude the probabilistic estimator from the remaining experiments, as it is well suited only for uniform score distributions.

Estimation Error Across Inputs. In this set of experiments, we evaluate the accuracy of the techniques at the different points in each template plan shown in Figure 11.

Figure 13(a) shows the average estimation error for DEEP and the sample-based estimator at different points in the template plan for a single-join workload. DEEP continues to perform well, consistently yielding estimation errors of at most 5% for all inputs. Given the diversity of the workload, which consists of single-join queries with randomly generated selection predicates over the ranked inputs, this demonstrates again that DEEP manages to model accurately depth computation with respect to the dependencies among join relationships, score values, and attribute values. In contrast, the error of the sampled-based estimator remains high on all in-

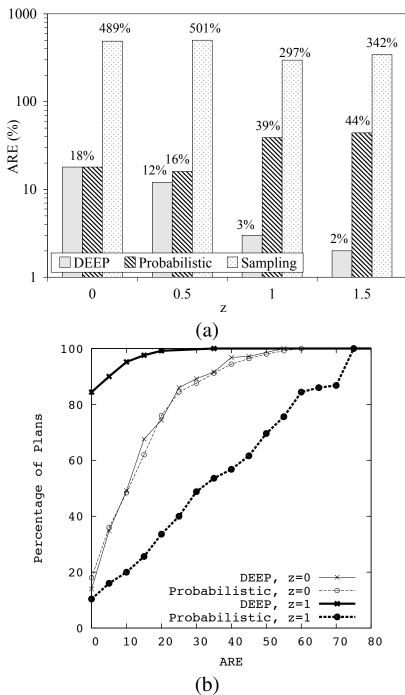


Figure 12: Varied skew [k=10, Input 1, single-join workload]: (a) average estimation error; (b) error CDF for DEEP and the probabilistic estimator.

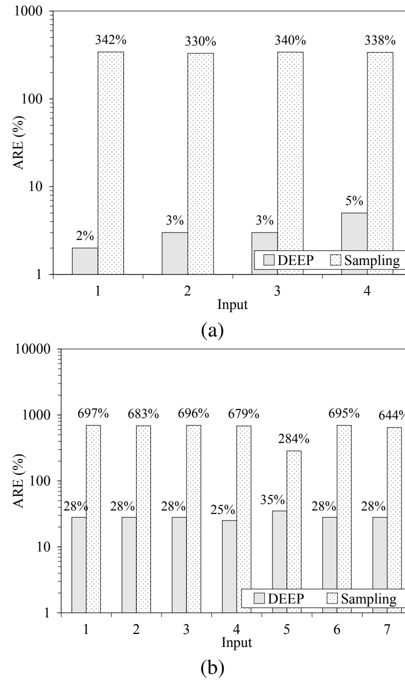


Figure 13: Average estimation error for different plan inputs [k=10, z=1.5]: (a) single-join workload; (b) two-join workload.

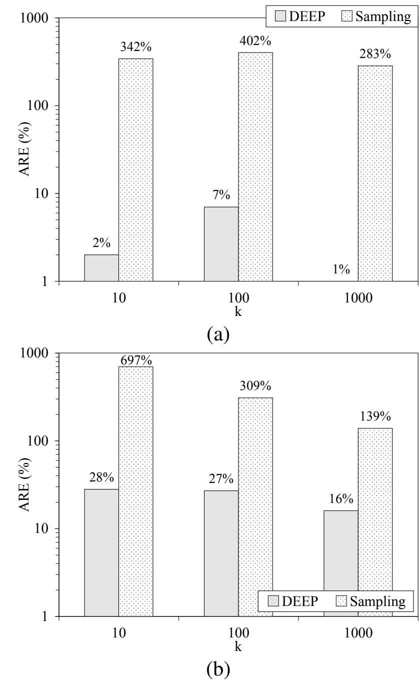


Figure 14: Average estimation error for different K [z=1.5, Input 1]: (a) single-join workload; (b) two-join workload.

puts, due again to the inability of independent samples to capture accurately the key/foreign key joins.

Figure 13(b) shows the results of the same experiment on a two-join workload. The estimation error is increased for both techniques, since the plans become more complex and the input depths depend on the correlations of three inputs (vs. two in the previous case). The increase in error for complex plans is analogous to results in selectivity estimation [9]. Nonetheless, DEEP continues to outperform the sample-based estimator by a wide margin, yielding an average error of less than 30% in most cases.

Estimation Error With Respect to k. The final set of experiments evaluates the effectiveness of the estimation techniques as we vary the number of top results from the plan. We set K equal to 10, 100, and 1000, and examine the estimation error at input 1 of each template. (The errors at other inputs were comparable.)

Figure 14(a) shows the average estimation error of DEEP and the sample-based estimator as a function of K, for the single-join workload. The results show that DEEP enables accurate depth estimates across a wide range of K values. The error of the sample-based estimator, on the other hand, remains consistently high and at about two orders of magnitude higher than DEEP.

Figure 14(b) shows the results of the same experiment for the two-join workload. Similar to the previous set of experiments, the higher complexity of the plans leads to increased estimation errors. DEEP, however, remains the most accurate technique, yielding errors that are reasonable in the context of query optimization.

5. ANALYSIS OF HRJN*

As mentioned in the previous sections, the estimation framework that we introduce is centered around the HRJN* variant of the HRJN operator. In this section, we present a theoretical analysis that shows the optimality of HRJN* and thus justifies its selection as the default implementation of HRJN.

The basic thresholding scheme used in HRJN* has been introduced by Fagin et al. [4] and its performance has been analyzed assuming that the join is 1-1, every input has a single ranking criterion, and every tuple in some input generates at least one result (total join participation). Our analysis considers a more extended scenario: the join is many-to-many, inputs may carry more than one ranking criterion, and total participation is not guaranteed. Another study [6] has examined the optimality of the generic HRJN algorithm within the class of deterministic rank join algorithms, and has shown that the round-robin pulling strategy ensures instance optimality.[3] That analysis, however, assumed that both L and R are simple relations, and does not extend to the case where one of the inputs comes from a query sub-plan (which is common in complex query plans). To the best of our knowledge, this is the first theoretical analysis of HRJN* in a broader context, and is thus of general interest beyond the scope of depth estimation.

We define an instance of the rank join problem as $I = (L, R, d)$, where L and R are the ranked inputs to be joined and d is the number of desired results. Given a rank join algorithm A, we define $TotalDepths(A, I)$ as the sum of the left and right depths when the algorithm runs on an input instance I. We note that our results extend to the model where inputs are read in "blocks" of tuples, and the cost of an algorithm is thus defined as the total number of accessed blocks. In what follows, we examine the $TotalDepths$

[3] Algorithm A is called instance-optimal within a class $\mathcal{A}$ of algorithms for a class $\mathcal{I}$ of inputs and cost function $cost$, if there are constants c and c' such that for every algorithm $B \in \mathcal{A}$ and input $I \in \mathcal{I}$ it holds that $cost(A, I) \leq c \cdot cost(B, I) + c'$. The constant c is called the optimality ratio of A.

cost of HRJN* within the class $\mathcal{H}$ of HRJN realizations. We first analyze the performance of HRJN* for a restricted class of inputs, and then extend our analysis to consider all possible inputs. In the interest of space, we present only the main results of our analysis. The complete results, along with detailed proofs, can be found in the full version of the paper [17].

A restricted class of inputs. Let $\mathcal{I}$ denote the class of possible instances of the rank join problem. We structure our analysis based on two different kinds of ties that may occur between scores in an instance. There can be *intra-input ties* where either $\overline{S}(\lambda_i) = \overline{S}(\lambda_j)$ for some $\lambda_i \neq \lambda_j$ or $\overline{S}(\rho_i) = \overline{S}(\rho_j)$ for some $\rho_i \neq \rho_j$. We also consider *inter-input ties*, where $\overline{S}(\lambda_i) = \overline{S}(\rho_j)$ for some λ_i, ρ_j. We define $\mathcal{I}^{\text{ties}} \subset \mathcal{I}$ as the class of instances with *both* inter- and intra-input ties. The remaining instances $\mathcal{I} - \mathcal{I}^{\text{ties}}$ may contain inter- or intra-input ties, but not both. The main intuition is that $\mathcal{I}^{\text{ties}}$ contains the most complex problem instances, while $\mathcal{I} - \mathcal{I}^{\text{ties}}$ represents cases of "controlled ambiguity" in the input scores, where HRJN* exhibits strong optimality properties.

Our first result states the optimality of HRJN* within $\mathcal{I} - \mathcal{I}^{\text{ties}}$.

THEOREM 5.1. *Let $A \in \mathcal{H}$ be an HRJN algorithm that terminates at depths l^A and r^A for an input instance $I \in \mathcal{I} - \mathcal{I}^{\text{ties}}$. Let l^* and r^* be the termination depths of* HRJN* *on the same instance. It holds that $l^* \leq l^A$ and $r^* \leq r^A$.* ∎

The proof considers the cases of no intra-input ties and no inter-input ties separately. In each case, we assume for contradiction that $l^* > l^A$ and then show that HRJN* must halt before reading $l^A + 1$ tuples from L. Our result shows that HRJN* is optimal in terms of each depth individually, and is therefore optimal with respect to the combined $TotalDepths$ metric that takes both depths into account. Thus, for the specific class of algorithms $\mathcal{H}$ and inputs $\mathcal{I} - \mathcal{I}^{\text{ties}}$, HRJN* will compute a solution with the least input access overall.

The general class of inputs. Next, we consider the class $\mathcal{I}$ of all inputs, where tie scores are allowed to simultaneously occur within and between input relations. The following theorem formalizes the optimality properties of HRJN* in this class.

THEOREM 5.2. HRJN* *is instance-optimal within $\mathcal{H}$ for the class $\mathcal{I}$ of inputs and the cost metric $TotalDepths$, with an instance optimality ratio of 2.* ∎

The proof works by showing that the maximum of the left and right depth for HRJN* is no larger than any other algorithm in $\mathcal{H}$. Overall, our analysis demonstrates the nice properties of HRJN*: for $\mathcal{I} - \mathcal{I}^{\text{ties}}$ it is the optimal algorithm for each depth individually, while for $\mathcal{I}^{\text{ties}}$ it performs at most twice as many accesses as the optimal algorithm for each specific instance. This provides strong justification for choosing HRJN* as the default algorithm in $\mathcal{H}$.

6. CONCLUSIONS

Accurate depth estimation is a key requirement for the effective incorporation of ranking queries in a relational database system. In this paper, we introduce the DEEP estimation framework for approximating the input depths of a physical plan with rank join operators. DEEP enables a systematic estimation methodology that takes directly into account the distribution of scores and values in the underlying data. We develop efficient estimation algorithms that implement the formalism of DEEP and validate their performance experimentally on data sets of diverse characteristics. The results verify the effectiveness of DEEP as an accurate estimation methodology, and demonstrate its several advantages over previously proposed techniques.

Acknowledgments. This work was partially supported by the National Science Foundation under Grant No. IIS-0447966p, and by an IBM Faculty Development Award.

7. REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join Synopses for Approximate Query Answering. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, 1999.

[2] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate Query Processing Using Wavelets. In *Proceedings of the 26th Intl. Conf. on Very Large Data Bases*, pages 111–122, 2000.

[3] Surajit Chaudhuri, Nilesh Dalvi, and Raghav Kaushik. Robust cardinality and cost estimation for skyline operator. *Proceedings of the 22nd Intl. Conf. on Data Engineering*, 0:64, 2006.

[4] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 102–113, 2001.

[5] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[6] F. Ilyas, G. Aref, and K. Elmagarmid. Supporting top-k join queries in relational databases. *International Journal on Very Large Databases*, 13(3):207–221, 2004.

[7] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid, Hicham Elmongui, Rahul Shah, and Jeffrey S. Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, December 2006.

[8] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 203–214, 2004.

[9] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, 1991.

[10] Yannis E. Ioannidis and Viswanath Poosala. Histogram-Based Approximation of Set-Valued Query Answers. In *Proceedings of the 25th Intl. Conf. on Very Large Data Bases*, pages 174–185, 1999.

[11] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 131–142, 2005.

[12] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116(1 & 2):195–226, 1993.

[13] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 426–435, New York, NY, USA, 1998. ACM Press.

[14] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-Based Histograms for Selectivity Estimation. In *Proceedings of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 448–459, 1998.

[15] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 294 – 305, 1996.

[16] Joshua Spiegel and Neoklis Polyzotis. Graph-based synopses for relational selectivity estimation. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 205–216, 2006.

[17] Karl Schnaitter, Joshua Spiegel, and Neoklis Polyzotis. Depth estimation for ranking query optimization. Technical report UCSC-CRL-07-02, UC Santa Cruz, 2007.