

Robust and Efficient Algorithms for Rank Join Evaluation

Jonathan Finger^{*}
Univ. of California Santa Cruz
jfinger@soe.ucsc.edu

Neoklis Polyzotis
Univ. of California Santa Cruz
alkis@ucsc.edu

ABSTRACT

In the rank join problem we are given a relational join $R_1 \bowtie R_2$ and a function that assigns numeric scores to the join tuples, and the goal is to return the tuples with the highest score. This problem lies at the core of processing top-k SQL queries, and recent studies have introduced specialized operators that solve the rank join problem by accessing only a subset of the input tuples. A desirable property for such operators is instance-optimality, i.e., their I/O cost should remain within a factor of the optimal for different inputs. However, a recent theoretical study has shown that existing rank join operators are not instance-optimal even though they have been shown to perform well empirically. The same study proposed the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator that was proved to be instance-optimal, but its performance was not tested empirically and in fact it was hinted that its complexity can be high. Thus, the following important question is raised: Is it possible to design a rank join operator that is both instance-optimal and computationally efficient?

In this paper we provide an answer to this challenging question. We perform an empirical study of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ and show that its computational cost can offset the benefits of instance-optimality. Using the insights gained by the study, we develop the novel FRPA operator that addresses the efficiency bottlenecks of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. We prove that FRPA is instance-optimal in general and more specifically that it never performs more I/O than $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. FRPA is the first operator that possesses these properties and is thus of interest in the theoretical study of rank join operators. We further identify cases where the overhead of FRPA becomes significant, and propose the a-FRPA operator that automatically adapts its overhead to the characteristics of the input. An extensive experimental study validates the effectiveness of the new operators and demonstrates that they offer significant performance improvements (up to an order of magnitude) over the state-of-the-art.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*Query Processing*

^{*}Author's current address: HistoRx, Inc., 300 George Street, New Haven, CT 06511.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...\$5.00.

General Terms

Algorithms, Performance.

Keywords

ranking queries, rank join, feasible region bound, adaptive pulling.

1. INTRODUCTION

Consider a database similar to `yelp.com` that records information on theaters, hotels, bars, etc. for different cities around the world. Imagine a query that returns pairs of theaters and restaurants that are located in Paris, ranked by a combination of the restaurant's rating and price, and the proximity of the theater to a specific location. Now, imagine the same query but assume that its results are returned unranked.

The previous example illustrates the concept of a *rank join* and its importance in the interactive exploration of query results. Loosely speaking, in the rank join problem we are given a relational join $R_1 \bowtie R_2$ and a function \mathcal{S} that assigns numerical scores to the join results, and the goal is to retrieve the K results with the highest scores. The rank join problem forms the basis for the evaluation of ranking SQL queries (also referred to as top-k queries) in relational DBMSs. The following is a sample ranking query that builds on our previous example:

```
SELECT h.name, b.name, t.name
FROM Hotels h, Bars b, Theaters t
WHERE h.city = b.city AND b.city = t.city AND h.city = 'Paris'
RANK BY 0.4*h.rating+0.1*b.rating+0.5/dist(t,currentLocation)
```

The importance of rank join evaluation has led to the development of a host of specialized rank join operators [1, 4, 8, 12, 11, 13]. These operators can generate the top join results by accessing only a subset of each input, provided that two common assumptions hold: the scoring function \mathcal{S} is monotonic, and the join operator can access input tuples in order of their potential to generate high scoring results. Recent studies [8, 10] have also shown that a physical plan for a ranking join query can be formed by pipelining several rank join operators. Returning to the previous example, one example pipeline would evaluate first the rank join over $\text{Hotels} \bowtie \text{Bars}$ and then feed the results to the rank join $(\text{Hotels} \bowtie \text{Bars}) \bowtie \text{Theaters}$. This plan will generate the top answers by scanning only a subset of each input relation, and may thus be far more efficient than the naive method of generating and scoring every join result. Overall, the ability to terminate before scanning the complete input is a key difference between rank join operators and conventional join operators.

Clearly, the I/O cost of a rank join operator is driven by the amount of input that it accesses before termination. This cost com-

ponent is significant when the input relations are large or when access to the input tuples is expensive, e.g., when the relations are streamed over the network. A natural question is whether there exists a rank join operator that generates the correct top results by accessing the least number of input tuples. Unfortunately, a result shown by Fagin et al. [4] yields a negative answer, i.e., no rank join operator is I/O optimal over all possible inputs. Since optimality is not possible, the study of rank join operators has used *instance-optimality* to characterize I/O efficiency. In a nutshell, a rank join operator is instance-optimal if its I/O cost for any input is within a constant factor of the cost of any other rank join operator on the same input. Instance optimality can thus be viewed as a property of *robustness*, since it bounds the difference in performance between an instance-optimal rank join operator and the optimal operator on any input.

Instance optimality has been shown to hold for existing rank join operators under specific assumptions [1, 11, 8], but a recent theoretical study [13] proved the following interesting result: None of the existing operators is instance-optimal in a setting that matches closely the execution environment of a database system. To this end, the study introduced the new $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator that was shown to be instance-optimal in this setting, and, to the best of our knowledge, it is the only deterministic rank join operator to have this property. However, the overall cost of a rank join operator involves both I/O and CPU cost, and instance-optimality characterizes solely the former. Given that there exist no empirical results on the performance of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$, it is unclear whether $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ is more efficient overall than existing algorithms (e.g., HRJN^* [8]) that are not instance-optimal within the same class of inputs as $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. If the answer to the previous questions is indeed negative, then it is natural to inquire whether we can design a rank join operator that is both computationally efficient and instance-optimal. The existence of such an operator will have clear implications for the design of efficient and robust ranking query processors.

Our Contributions. Motivated by the previous observations, we investigate the existence of rank join operators that are computationally efficient and robust (i.e., instance-optimal) in terms of I/O cost. We begin with an empirical study of the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator against the well known HRJN^* operator and demonstrate that the computational overhead of the former essentially cancels its instance-optimality and leads to bad performance overall. The experimental results point to two main sources of inefficiency: the computation of bounds on the scores of unseen join results, and unnecessary I/O which results from a blind round-robin access to the two inputs. This is the first empirical study of the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator and the results are thus of general interest.

Having demonstrated the inefficiency of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$, we embark on the design of a rank join operator that is computationally efficient and instance-optimal in terms of I/O. To this end, we develop techniques that directly address the bottlenecks of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. We first introduce the FR^* scheme that allows the efficient computation of score bounds on unseen join results. We subsequently couple FR^* with a strategy that prioritizes the I/O requests of the rank join operator based on the potential of each input to generate results with high scores. We combine these two techniques in the novel FRPA rank join operator. We show analytically that FRPA is instance-optimal, and more specifically that it provably outperforms $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ in terms of I/O cost. To the best of our knowledge, this is the first rank join operator to have these properties. Moreover, the optimized FR^* scheme ensures that the computational overhead of FRPA is low for a large class of inputs.

To address the cases where the overhead of FRPA becomes prohibitively expensive, we introduce the adaptive a-FRPA rank join

operator. The new operator employs a novel adaptive scheme to regulate the overhead of computing score bounds on unseen join results. An interesting property of a-FRPA is that it works precisely like FRPA on the inputs where the latter is efficient, and it gradually morphs its behavior towards the computationally efficient (yet, not instance-optimal) HRJN^* operator for the inputs on which FRPA does not work well. Thus, the key advantage of a-FRPA is that it can adaptively explore the trade-off between instance-optimality and computational efficiency.

Finally, we present an extensive experimental study that evaluates the performance of the new operators under different operating parameters. The results demonstrate that FRPA and a-FRPA outperform the state-of-the-art operators by a significant margin (in some cases, by an order of magnitude) in terms of computational efficiency and I/O cost. Moreover, we observe that the adaptive a-FRPA operator provides a “best-of-both-worlds” hybrid, having low computation overhead and an I/O cost that is either equal or very close to the instance-optimal cost of the FRPA operator. Overall, our study validates the effectiveness of the proposed techniques and demonstrates their numerous advantages over existing rank join operators.

2. PRELIMINARIES

In this section, we formally define the rank join problem and present a formalism for modeling rank join operators.

2.1 The Rank Join Problem

We consider the natural join of two relations R_1 and R_2 , where each tuple $\tau_i \in R_i$ is composed of *attribute values* and *base scores*. The base scores are denoted as a vector $\mathbf{b}(\tau_i) \in [0, 1]^{e_i}$ for some $e_i \geq 0$, and signify the importance of the tuple according to criteria specified by the query. The score vector of a join result $\tau = \tau_1 \bowtie \tau_2$ is defined as the concatenation of $\mathbf{b}(\tau_1)$ and $\mathbf{b}(\tau_2)$. Base scores are aggregated using a *scoring function* \mathcal{S} that computes the score of τ as $\mathcal{S}(\mathbf{b}(\tau))$. We may also use $\mathcal{S}(\tau)$ as shorthand for the score of τ . Following common practice, we assume that \mathcal{S} is monotonic, i.e., $\mathcal{S}(x_1, \dots, x_e) \leq \mathcal{S}(y_1, \dots, y_e)$ if $x_i \leq y_i$ for all i .

Given a tuple $\tau_1 \in R_1$, we define $\overline{\mathcal{S}}(\tau_1)$ to be the value of \mathcal{S} using the base scores of τ_1 and substituting 1 for the missing scores. We call $\overline{\mathcal{S}}(\tau_1)$ the *score bound* of τ_1 , since monotonicity implies that $\overline{\mathcal{S}}(\tau_1) \geq \mathcal{S}(\tau)$ for any join tuple $\tau = \tau_1 \bowtie \tau_2$ that is derived from τ_1 . We define the score bound $\overline{\mathcal{S}}(\tau_2)$ in a similar fashion for a tuple $\tau_2 \in R_2$.

The objective of the rank join problem that we consider in this paper is to find K tuples with the highest scores from the natural join $R_1 \bowtie R_2$. Formally:

DEFINITION 2.1. *An instance I of the rank join problem is a 4-tuple $(R_1, R_2, \mathcal{S}, K)$ such that: (a) relations R_1 and R_2 are accessed sequentially in decreasing order of $\overline{\mathcal{S}}$; (b) \mathcal{S} is a monotonic scoring function; and (c), $0 < K \leq |R_1 \bowtie R_2|$.*

This definition requires that at least K join results exist, which guarantees that it is possible to fulfill a request for the top K results. We do not place any restrictions on the input relations except that each R_i is accessed sequentially and in decreasing order of $\overline{\mathcal{S}}$. Formally, we use $R_i[p]$ to denote the p -th tuple in R_i and assume that $\overline{\mathcal{S}}(R_i[p]) \geq \overline{\mathcal{S}}(R_i[q])$ for $q \geq p$. This particular access model is a common assumption in the study of rank join evaluation in database systems [8, 12], as it enables the development of efficient rank join operators (which we discuss later). When the inputs are base tables, this type of access is typically provided through index structures. For instance, going back to the example ranking query

of the previous section, the ordered access to relation Bars can be provided by an index on Bars.rating.

A solution to a problem instance I is an ordered relation O comprising the top K results of $R_1 \bowtie R_2$ ordered by \mathcal{S} . There may be more than one possible solution O for a particular instance I if there are tied scores in the output, but the sequence of scores in O is completely determined by I . We henceforth use $\mathcal{S}^{\text{term}}$ to denote the least score in any solution O .

On a final note, it is possible to generalize the definition of the rank join problem to the natural join of n relations R_1, \dots, R_n . We focus on the binary rank join problem because existing database systems implement binary physical operators, and also because the restriction to two inputs enables interesting optimizations. (Note that some of our techniques extend naturally to the n -ary case.) However, the n -ary problem is also interesting, since multi-way rank join operators have been shown to be instance-optimal compared to plans of binary rank join operators [13]. We believe that a full investigation of the n -ary rank join variant is an interesting direction for future work.

2.2 Rank Join Operators

A rank join operator is a deterministic algorithm that solves the aforementioned rank join problem. We are interested in the evaluation of rank joins in a database system, so we assume that a rank join operator works incrementally and supports a *getNext()* method which returns the next result in its output. (This is typically referred to as the *iterator* interface [6].) Thus, a solution O is obtained by invoking *getNext()* K times.

Given a rank join operator and a problem instance I , we define its left and right *depths* as the number of input tuples from R_1 and R_2 , respectively, that the operator accesses in order to satisfy the K *getNext* requests. Clearly, the input depths reflect the amount of I/O performed by the algorithm and thus affect heavily the cost of rank join evaluation. In what follows, we use $\text{depth}(A, I, i)$ to denote the depth of algorithm A on relation R_i of some problem instance I , and $\text{sumDepths}(A, I) = \text{depth}(A, I, 1) + \text{depth}(A, I, 2)$ for the total number of accessed tuples.

Previous studies on rank join operators employ the notion of *instance-optimality* to characterize the cost of an algorithm with respect to the *sumDepths* metric. Formally, given a class of algorithms \mathcal{A} and a class of rank join instances \mathcal{I} , an algorithm $A \in \mathcal{A}$ is instance-optimal within \mathcal{A} and \mathcal{I} if there exist constants c_0 and c_1 such that $\text{cost}(A, I) \leq c_0 \min\{\text{cost}(B, I) \mid B \in \mathcal{A}\} + c_1$ for any instance $I \in \mathcal{I}$. Constant c_0 is called the optimality ratio of A . In a nutshell, instance-optimality implies that the operator cannot perform much more I/O than any other operator on any rank join instance. We often refer to this property as *robustness* and accordingly refer to an instance-optimal operator as *robust*.

The PBRJ template. We adopt the formalism of the *Pull Bound Rank Join* template [13] (or, PBRJ for short) to describe rank join operators. The pseudo-code for PBRJ is shown in Figure 1. PBRJ is an algorithm template that is instantiated with two deterministic components, namely, a pulling strategy P , and a bounding scheme B . On each loop iteration (lines 2–7), the pulling strategy P chooses a relation R_i to read, and the new tuple ρ_i is stored in an input buffer HR_i (typically a hash table). New join results are generated by joining ρ_i with the tuples in the other input buffer and they are pushed to an ordered output buffer O . After each tuple is processed, it is given to the bounding scheme B via the method *updateBound*. The return value t has the following semantics: for any join tuple $\tau = \tau_1 \bowtie \tau_2$ such that $\tau_1 \in R_1 - HR_1 \vee \tau_2 \in R_2 - HR_2$, it holds that $\mathcal{S}(\tau) \leq t$. In other words, t provides a *bound on the score of unseen join results*. A call to *getNext* returns

Function PBRJ.getNext()

Output: Next tuple of $R_1 \bowtie R_2$ ordered by \mathcal{S} .

Data: Input buffers HR_1 and HR_2 initialized to empty; sorted output buffer O initialized to empty; bound t initialized to ∞ .

```

1 while ( $O = \emptyset \vee \mathcal{S}(O.\text{top}()) < t$ )  $\wedge$  inputs not exhausted do
2    $i \leftarrow P.\text{chooseInput}()$ ;
3    $\rho_i \leftarrow$  next tuple of  $R_i$ ;
4    $R \leftarrow \rho_i \bowtie HR_j$  for  $j \neq i$ ;
5   Add each member of  $R$  to  $O$ ;
6   Add  $\rho_i$  to  $HR_i$ ;
7    $t \leftarrow B.\text{updateBound}(\rho_i)$ ;
8 if  $O \neq \emptyset$  then return  $O.\text{pop}()$  else return EndOfOutput;
```

Figure 1: PBRJ template. The operator is instantiated with a pulling strategy P and a bounding scheme B .

the top tuple in O provided its score is not smaller than the bound t , since this indicates that the buffered results cannot be improved by reading more tuples.

PBRJ provides a convenient method to analyze the performance of deterministic rank join operators. More formally, the following “equivalence” result holds [13]: Given a rank join operator A , there exists an instantiation F_A of PBRJ such that, for any instance I , it holds that $\text{depth}(A, I, i) = \text{depth}(F_A, I, i)$ for $i \in \{1, 2\}$. In this paper we derive theoretical results for specific instantiations of PBRJ, which extend to the set of deterministic rank join operators by virtue of this equivalence.

3. LIMITATIONS OF STATE OF THE ART

In this paper we investigate the following question: Is it possible to design a rank join operator that is both instance-optimal (or, robust) and computationally efficient? To provide some background, we first review the state of the art in rank join operators, and then provide a brief overview of an experimental study that we conducted to evaluate the only known operator to be instance-optimal in a general setting. These results validate the motivation behind our work and also provide valuable insights that we use in the development of our novel rank join operators.

3.1 State of the Art

In what follows, we briefly review previous studies on rank join evaluation that consider the same target domain as our work, i.e., rank joins with an equi-join condition and several score attributes per input. It is important to note that rank join evaluation is related to the problem of ranked list aggregation, and indeed previous studies adapt several ideas from the seminal work of Fagin et al [4]. We also note the existence of studies that consider the generation of ranked output in a different query model [15, 7, 2], and are therefore omitted from our review.

The HRJN* operator of Ilyas et al. [8] is an instantiation of the PBRJ template with the *corner bounding* scheme and the *threshold-adaptive pulling* strategy. The corner bound maintains a per-input threshold $\text{thr}_i = \overline{\mathcal{S}}(\rho_i)$, where ρ_i is the last accessed tuple from the same input, and returns $\max(\text{thr}_1, \text{thr}_2)$ as the bound value. In turn, the threshold-adaptive strategy pulls from the input with the highest value for thr_i . HRJN* has been shown to perform well in practice, but a recent study [13] proved that it is not instance-optimal for the variant of the rank join problem that we consider.

The PBRJ_{FR}^{RR} [13] rank join operator instantiates the PBRJ template with the FR bounding scheme and a round-robin pulling strategy. (We examine this operator in more detail in Section 3.2.) The original study showed PBRJ_{FR}^{RR} to be instance-optimal within the

class of algorithms and instances that we consider in this paper. To the best of our knowledge, this is the only known operator to have this property. However, the original study did not provide an empirical evaluation of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$, and hence it is not known whether it performs efficiently in practice.

The recent work of Agrawal and Widom [1] introduced a rank join operator in the context of uncertain databases. The main novelty of their algorithm is that it operates with limited memory, but the assumption is that it is able to rescan the inputs at different offsets. This makes it unsuitable as an intermediate operator in pipelined physical plans. Our work targets the scenario where the inputs can be accessed only in a single-pass fashion, which matches the model of physical execution plans for ranking queries [10, 14]

Finally, a recent study introduced the LARA-J algorithm [11] which uses ideas similar to the earlier J^* [12] operator. Both operators, however, are defined for problem instances where each relation has a single score attribute. We target the more general variant of several score attributes per input, which arises frequently in the pipelined evaluation of several rank join operators [10].

3.2 $\text{PBRJ}_{\text{FR}}^{\text{RR}}$: Is it Efficient?

As indicated in the previous discussion, $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ [13] is the only rank join operator known to be instance-optimal in the general setting that we consider. Given that there exist no published results on the empirical performance of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$, we conducted an experimental study to evaluate its efficiency. Here we review the results from one representative experiment that lead to some interesting observations. The details of the experimental methodology and a detailed review of the results appear in Section 6.

We first review the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator. The operator instantiates the PBRJ template with the following bounding scheme and pulling strategy, respectively:

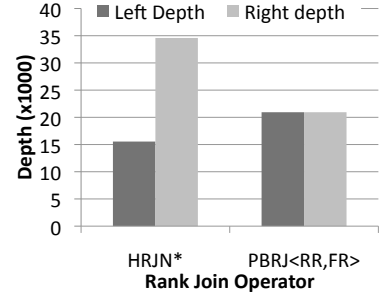
Feasible Region Bound (FR) The FR bound maintains a *cover* CR_i for each input R_i that bounds tightly the base scores of tuples in $R_i - HR_i$. An example cover is shown in Figure 4(a). Using this information, FR is able to compute a tight bound for the score of an unseen result tuple τ . We elaborate on the details of these covers and the bound computation in Section 4.1.

Round Robin Pulling Strategy (RR) The RR pulling strategy simply alternates between the two inputs.

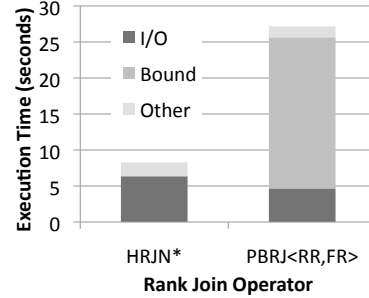
As mentioned above, the FR bound is *tight*, which means that it is indeed possible for the unseen tuples in $R_1 - HR_1$ and $R_2 - HR_2$ to generate a join result whose score is equal to the bound. This tightness property is key in proving that $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ is instance-optimal within the class of deterministic rank join operators with an optimality ratio of 2.

We now discuss the results of our study. To provide some context for comparison, we pitted $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ against the HRJN^* operator of Ilyas et al. [8]. HRJN^* makes for an interesting competitor because it was shown to have good performance in practice, yet a recent study proved that it is not instance-optimal. We note that this is the first empirical comparison between the two algorithms, and the results are thus of general interest.

Figure 2(a) shows the performance of the two algorithms in terms of the *sumDepths* metric. $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ outperforms HRJN^* by a significant margin of 15K tuples, which agrees with the theoretical results of [13]. Another interesting observation is that HRJN^* is able to stop considerably earlier on the left input compared to $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. Essentially, the HRJN^* operator uses an adaptive pulling strategy that allows it to focus its accesses on the input with most potential.



(a)



(b)

Figure 2: A comparison of HRJN^* and $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ on a problem instance (R_1, R_2, S, K) such that $K = 100$ and each input has 3 base scores ($e_1 = e_2 = 3$). The scores in the two relations are distributed according to a Zipfian distribution with skew 0.5 and using a score cut of .75. (The methodology for generating scores is described in Section 6.1.) Part (a) shows the input depths of the algorithms, and part (b) shows the total execution time and its breakdown in terms of three components: I/O time, time spent in calling the bounding scheme, and time spent in other tasks.

On the other hand, $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ follows a blind round-robin pulling pattern that may lead to unnecessary accesses from a specific input.

Figure 2(b) depicts the overall execution time of the two operators for the same experiment. The chart also shows the breakdown in terms of three components: time spent doing I/O, computation of the bound on unseen join results, and other computations. The overall execution time indicates that, in spite of the significant savings in I/O, $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ actually performs worse. The breakdown reveals that the computation of the FR bound dominates in terms of cost and essentially outweighs the savings in I/O.

We observed similar trends using several other problem instances. To summarize, the state-of-the-art rank join operators seem to occupy two “corners” in the two-dimensional plane that indicates efficiency and instance-optimality: either they are instance-optimal but not efficient, or they are efficient but not instance-optimal. The techniques that we develop in subsequent sections are motivated by this observation and aim to populate the corner point of efficiency and instance-optimality.

4. THE FRPA RANK JOIN OPERATOR

Motivated by the observations in the previous section, we embark on the design of a rank join operator that is both instance-

optimal and computationally efficient. Our initial approach is to address the inefficiencies of the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator. To this end, we first develop the *Fast Feasible Region* bound (denoted as FR^*), which guarantees the tightness property of the FR bound but it is more efficient to maintain. We couple the new bound with a novel pulling strategy termed *Potential Adaptive* (denoted as PA) that prioritizes its selections based on a potential metric for each input. These two components give rise to the novel FRPA operator, which has several attractive properties: (a) it provably outperforms $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ in terms of I/O cost, (b) it is instance-optimal, and (c) the experimental results suggest that it outperforms both HRJN* and $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ by a wide margin for a large class of inputs.

Before detailing the design of the new operator, we introduce some necessary notation and terminology. Given e -dimensional points $x = (x_1, \dots, x_e)$ and $y = (y_1, \dots, y_e)$, we define the binary relations \preceq , \prec , and \ll as follows: $x \preceq y$ if $x_i \leq y_i$ for all i ; $x \prec y$ if $x \preceq y$ and $x \neq y$; and, $x \ll y$ if $x_i < y_i$ for all i . We say that a set of points C is a cover for a set of points X if for every $x \in X$ there exists $c \in C$ such that $x \preceq c$. A skyline [3, 9] is a special case of a minimal cover where the covering points come from X . More specifically, the skyline of X , denoted as $SL(X)$, is the set $C \subseteq X$ such that C covers X and for all $c, c' \in C$ it holds that $c \not\ll c'$. Finally, we use $x[i \mapsto \alpha]$ to denote the point that results by substituting the i -th coordinate of x with α .

4.1 Overview of the FR Bound

We begin with a short overview of the FR bound used by the $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ operator [13]. Our presentation focuses on the components that are relevant to the development of our techniques. We refer the reader to the original study for the complete details.

The pseudo-code for the FR bound is shown in Figure 3. The main component is the $\text{FR}::\text{UpdateBound}$ function that is invoked with every newly accessed tuple ρ_i and returns the updated bound value. We first discuss the global variables used by the function, namely CR_i , G_i , and g_i . Each CR_i stores a cover of $\mathbf{b}[R_i - HR_i]$ and thus delineates a region that contains the base scores of unseen tuples. (See also Figure 4(a).) Variable g_i is the score bound of the last accessed tuple from R_i , and G_i comprises all observed tuples from R_i with a score bound of g_i . This effectively divides R_i into contiguous *groups* of tuples with equal score bounds. Thus, G_i acts as a buffer for the accessed tuples from the current group and g_i indicates their score.

The cover CR_i is updated with the score vectors in G_i when a new group is detected (line 2 in function UpdateBound). The idea is the following. Let τ_i be an unseen tuple and τ'_i be a tuple in G_i . Given that g_i has changed and that R_i is accessed in decreasing order of \bar{S} , it holds that $\bar{S}(\tau_i) \leq \bar{S}(\rho_i) < \bar{S}(\tau'_i)$. The monotonicity of \mathcal{S} implies that $\mathbf{b}(\tau_i)$ cannot dominate $\mathbf{b}(\tau'_i)$, and thus it follows that the region dominated by $\mathbf{b}(\tau'_i)$ can be removed from CR_i . This update is performed using function UpdateCR . We do not go into the details in the interest of space, but we illustrate this update in Figure 4(b).

Once the global data structures are updated, UpdateBound computes and returns the updated value of the bound (line 7). The computation is done by function ResultBound . Let $\tau \equiv \tau_1 \bowtie \tau_2$ be a result tuple that the rank join operator has not discovered yet, i.e., $\tau_1 \in R_1 - HR_1 \vee \tau_2 \in R_2 - HR_2$. Function ResultBound returns a bound on $\mathcal{S}(\tau)$, computed as the maximum of three bounds t_1 , t_2 , and t_{both} . Each individual bound corresponds to the different possibilities for τ_1 and τ_2 . Let us consider first t_2 , which represents the case $\tau_1 \in HR_1 \wedge \tau_2 \in R_2 - HR_2$, i.e., the join tuple is formed by an unseen tuple of R_2 and a seen tuple of R_1 . Clearly, $\bar{S}(\tau_2)$ provides a correct upper bound for

```

Function  $\text{FR}::\text{UpdateBound}(\rho_i)$ 
Input: Tuple  $\rho_i$  read from input  $i$ ,  $i \in \{1, 2\}$ .
Output: The updated value of the feasible region bound.
Data: Covers  $CR_1$  and  $CR_2$  for  $\mathbf{b}[R_1 - HR_1]$  and  $\mathbf{b}[R_2 - HR_2]$ ,
    respectively, initialized to  $\{(1, 1, \dots, 1)\}$ ; current groups  $G_1$ 
    and  $G_2$  initialized to empty; current bounds  $g_1$  and  $g_2$ 
    initialized to  $\infty$ 
1 if  $\bar{S}(\rho_i) < g_i$  then
2    $CR_i \leftarrow \text{FR}::\text{UpdateCR}(CR_i, \mathbf{b}[G_i])$ ;
3    $g_i \leftarrow \bar{S}(\rho_i)$ ;
4    $G_i \leftarrow \{\rho_i\}$ ;
5 else
6    $G_i \leftarrow G_i \cup \{\rho_i\}$ ;
7 return  $\text{FR}::\text{ResultBound}()$ ;

Function  $\text{FR}::\text{UpdateCR}(C, Y)$ 
Input: Current cover  $C$ ; Set of new score vectors  $Y$ 
Output: Updated cover
1 if  $Y = \emptyset$  then return  $C$ ; // Base case for recursion
2  $y \leftarrow$  some element of  $Y$ ;
3  $S \leftarrow \text{FR}::\text{UpdateCR}(C, Y - \{y\})$ ; // Recursive Call
4  $S^- \leftarrow \{s \in S \mid y \preceq s\}$ ; // Removed points
5  $S^+ \leftarrow \bigcup_{i=1}^e \{s^-[i \mapsto y[i]] \mid s^- \in S^-\}$ ; // New points
6 return  $(S - S^-) \cup (S^+ \cap (0, 1]^e)$ ;

Function  $\text{FR}::\text{ResultBound}()$ 
Output: The maximum score of a join result  $\tau_1 \bowtie \tau_2$  such that
     $\tau_1 \in R_1 - HR_1 \vee \tau_2 \in R_2 - HR_2$ .
// Case (i):  $\tau_1 \in HR_1 \wedge \tau_2 \in R_2 - HR_2$ 
1  $t_2^{\text{cover}} \leftarrow \max\{\mathcal{S}(\mathbf{b}[\tau_1]c_2) \mid \tau_1 \in HR_1 \wedge c_2 \in CR_2\}$ ;
2  $t_2^{\text{order}} \leftarrow g_2$ ;
3  $t_2 \leftarrow \min\{t_2^{\text{cover}}, t_2^{\text{order}}\}$ ;
// Case (ii):  $\tau_1 \in R_1 - HR_1 \wedge \tau_2 \in HR_2$ 
4  $t_1^{\text{cover}} \leftarrow \max\{\mathcal{S}(c_1 \mathbf{b}[\tau_2]) \mid c_1 \in CR_1 \wedge \tau_2 \in HR_2\}$ ;
5  $t_1^{\text{order}} \leftarrow g_1$ ;
6  $t_1 \leftarrow \min\{t_1^{\text{cover}}, t_1^{\text{order}}\}$ ;
// Case (iii):  $\tau_1 \in R_1 - HR_1 \wedge \tau_2 \in R_2 - HR_2$ 
7  $t_{\text{both}}^{\text{cover}} \leftarrow \max\{\mathcal{S}(c_1 c_2) \mid c_1 \in CR_1 \wedge c_2 \in CR_2\}$ ;
8  $t_{\text{both}}^{\text{order}} \leftarrow \min\{g_1, g_2\}$ ;
9  $t_{\text{both}} \leftarrow \min\{t_{\text{both}}^{\text{cover}}, t_{\text{both}}^{\text{order}}\}$ ;
// Final bound
10 return  $\max\{t_1, t_2, t_{\text{both}}\}$ ;

```

Figure 3: The FR bound.

$\mathcal{S}(\tau)$, and consequently so does $g_2 \geq \bar{S}(\tau_2)$. This is termed the *order bound* and is denoted as t_2^{order} . A second bound is derived using the cover CR_2 . The semantics of the cover imply that $\mathbf{b}(\tau_2)$ is dominated by at least one point c_2 in CR_2 , and consequently $\mathcal{S}(\tau_1 \bowtie \tau_2) \leq \mathcal{S}(\text{base}(\tau_1)c_2)$. By taking the maximum score value over all possible choices of τ_1 and c_2 , we obtain a correct bound t_2^{cover} (line 1 in Function ResultBound) termed the *cover bound*. The final bound t_2 is derived as the minimum of the two correct bounds.

The two remaining bounds t_1 and t_{both} are defined similarly. More concretely, t_1 corresponds to the case where only τ_1 comes from the unseen part of R_1 and $\tau_2 \in HR_2$. The third case is where both τ_1 and τ_2 are unseen, i.e., $\tau_1 \in R_1 - HR_1 \wedge \tau_2 \in R_2 - HR_2$. The threshold value of the feasible region bound is computed as the maximum over the three possible choices, since each choice represents a different case for the result tuple $\tau \equiv \tau_1 \bowtie \tau_2$.

Efficiency of the FR Bound. At this point, it is interesting to discuss the computational efficiency of the FR bound. It is straightforward

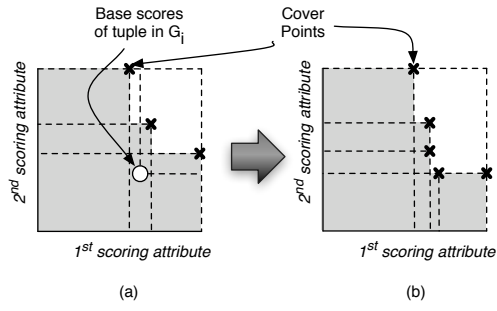


Figure 4: Part (a) shows an example cover CR_i assuming two-dimensional score vectors. The cover consists of a set of points. The gray region dominated by the cover points contains the base scores of tuples in $R_i - HR_i$. Part (b) shows an update of the cover using a score vector y corresponding to a tuple in G_i . The vector identifies a region, namely the vectors that dominate y , that must be removed from the feasible region. Any cover points that exist in that region are projected along the boundaries of the removed region.

ward to show that $FR::UpdateBound$ is in PTIME under data complexity, but the exponent of the polynomial and the hidden constants contribute to a high overhead. One major source of inefficiency is the combinatorial complexity of computing the cover bounds. More concretely, each cover bound requires the computation of a cross product, whose size grows as the rank join operator accesses more tuples. Moreover, each invocation of $resultBound$ requires the computation of three such cross products.

4.2 Operator Definition

The FRPA operator is an instantiation of the PBRJ template (Figure 1) using the new FR^* bounding scheme and the new PA pulling strategy. (Thus, following the notation established in [13] we can denote the new rank join operator as $PBRJ_{FR^*}^{PA}$.) In what follows, we describe these two components and then analyze the performance of FRPA.

4.2.1 The FR^* Bound

The FR^* bound represents an optimized version of the FR bound. This implies that it too is a tight bounding scheme, which is crucial for the optimality properties of FRPA.

Reducing the complexity of cover bounds. The first observation behind FR^* is that it is possible to speed-up the computation of cover bounds by taking into account the monotonicity of the scoring function \mathcal{S} . More concretely, let us consider the computation of t_1^{cover} (line 4 of function $FR::ResultBound$). Consider a point $c \in CR_1$ such that $c \notin SL(CR_1)$. The monotonicity of \mathcal{S} guarantees that $\max\{\mathcal{S}(cb[\tau_2]) \mid \tau_2 \in HR_2\} \leq \max\{\mathcal{S}(c'b[\tau_2]) \mid c' \in SL(CR_1), \tau_2 \in HR_2\}$. In other words, the value of t_1^{cover} can be computed using solely the points in $SL(CR_1)$. Using a similar reasoning, we can prove the same result for the skyline $SL(b[HR_2])$. This gives rise to the following definition of t_1^{cover} .

$$t_1^{cover} = \max\{\mathcal{S}(c_1s_2) \mid c_1 \in SL(CR_1), s_2 \in SL(b[HR_2])\}$$

Similarly, we can redefine the other two cover bounds as shown below.

$$t_2^{cover} = \max\{\mathcal{S}(s_1c_2) \mid s_1 \in SL(b[HR_1]), c_2 \in SL(CR_2)\}$$

$$t_{both}^{cover} = \max\{\mathcal{S}(c_1c_2) \mid c_1 \in SL(CR_1), c_2 \in SL(CR_2)\}$$

	ρ_i a skyline point for HR_i ?	
	Yes	No
$\bar{\mathcal{S}}(\rho_i) = g_i$	t_i^{cover}	–
$\bar{\mathcal{S}}(\rho_i) < g_i$	all t_i^{order} but	all t_i^{order} but and t_i^{cover}

Table 1: Decision matrix on which bounds need to be recomputed when $resultBound$ is invoked after the access of a tuple ρ_i . The symbol \bar{i} denotes the opposite input, i.e., $\bar{i} = 1$ if $i = 2$ and vice versa.

The revised expressions have clearly lower complexity, assuming of course that the skylines can be maintained efficiently. (We show later that this is indeed possible.) Another interesting property is that the skyline $SL(b[HR_i])$ is likely to “freeze” relatively early, since R_i is accessed in decreasing order of $\bar{\mathcal{S}}$ and this in turn implies that the dominating points are accessed first. The early freeze property limits the complexity of computing the cover bounds even if HR_i continues to grow in size.

Avoiding redundant computation. The second observation is that some of the computation performed by $FR::ResultBound$ may be cached for subsequent invocations of the function. By identifying such opportunities we can reduce significantly the overhead of the bounding scheme.

We illustrate this point with an example. Suppose that $FR::UpdateBound$ is invoked with a tuple $\rho_1 \in R_1$ such that $\bar{\mathcal{S}}(\rho_1) = g_1$, i.e., the current group G_1 is not reset. An immediate observation is that CR_1 is not modified and the same holds for g_1 . Also, since the pull happened on input R_1 , neither CR_2 , HR_2 , nor g_2 change. These observations allow us to assert that, out of the six bounds computed in $ResultBound$, only t_2^{cover} may change in value and thus affect the returned bound. Furthermore, a closer examination shows that t_2^{cover} may change in value only if $b(\rho_1)$ causes a change of $SL(b[HR_1])$. Hence, we can assert the following general implication: If $\bar{\mathcal{S}}(\rho_i) = g_i$ then the value of $FR::ResultBound$ can change only if ρ_i is in $SL(HR_i)$.

Let us now consider the converse case where $\bar{\mathcal{S}}(\rho_1) < g_1$. This causes an update on CR_1 , which consequently affects t_1^{cover} and t_{both}^{cover} . Also, t_1^{order} and t_{both}^{order} are affected, since g_1 changes in value. However, we observe that t_2^{order} remains unchanged, and in addition t_2^{cover} remains unmodified if ρ_1 is not a new skyline point. Hence, we may still be able to avoid some (potentially significant) computation by tracking the skyline of HR_1 .

Definition of the FR^* Bound. The FR^* bound incorporates the previous optimizations in the computation of the feasible region bound. Figure 5 shows the pseudo-code. FR^* maintains a set SHR_i that stores the skyline of $base[HR_i]$ and is updated with every new tuple ρ_i . The complexity of this update is linear to $|SHR_i|$, and in practice we expect it to be low due to the early freeze property. FR^* also ensures that each cover CR_i forms a skyline, by modifying slightly function $UpdateCR$. The trick is to skyline the set $S^+ \cap (0, 1]^e$, which holds the new points of the cover, prior to performing the union in line 6 of $UpdateCR$. This slight modification ensures that $FR^*::UpdateCR$ always returns a skyline of points.

Function $ResultBound$ is modified to use the alternative cover bound definitions and also to selectively recompute the different bound components. The computation of the bound components is guided by the decision matrix shown in Table 1. The return value of t is derived using the recomputed components and the cached values for the remaining components. Our experimental results show

Function $\text{FR}^*::\text{UpdateBound}(\rho_i)$
Input: Tuple ρ_i read from input $i, i \in \{1, 2\}$.
Output: The updated value of the feasible region bound.
Data: Sets CR_i initialized to empty; current groups G_i initialized to empty; current bounds g_i initialized to ∞ ; Sets SHR_i .

- 1 $SHR_i \leftarrow SL(SHR_i \cup \{\rho_i\})$;
- 2 **if** $\bar{S}(\rho_i) < g_i$ **then**
- 3 $CR_i \leftarrow \text{FR}^*.\text{UpdateCRCR}_i, \mathbf{b}[G_i]$;
- 4 $g_i \leftarrow \bar{S}(\rho_i)$;
- 5 $G_i \leftarrow \{\rho_i\}$;
- 6 **else**
- 7 $G_i \leftarrow G_i \cup \{\rho_i\}$;
- 8 **return** $\text{FR}^*::\text{ResultBound}()$;

Function $\text{FR}^*::\text{UpdateCR}(C, Y)$
Input: Current cover C ; Set of new score vectors Y
Output: Updated cover
// Same as $\text{FR}.\text{UpdateCR}$ except that the return statement is the following

- 6 **return** $(S - S^-) \cup SL(S^+ \cap (0, 1]^e)$;

Function $\text{FR}^*::\text{ResultBound}()$
Output: The maximum score of a join result $\tau_1 \bowtie \tau_2$ such that $\tau_1 \in R_1 - HR_1 \vee \tau_2 \in R_2 - HR_2$.
// Same as $\text{FR}.\text{ResultBound}$ except: (i) cover bounds employ SHR_i instead of HR_i , and (ii) only the bounds shown in Table 1 are recomputed

Figure 5: The FR^* bound.

that this approach yields significant savings in execution time. In several cases we avoid recomputing some (or even all) of the cover bounds, which, as mentioned earlier, have combinatorial complexity.

As a concluding remark, we state the following tightness theorem that follows directly from the equivalence between FR^* and the original FR bound. The tightness property is crucial for the optimality results that we state later.

THEOREM 4.1. *Let I be a rank join instance and consider the execution of PBRJ on I using the FR^* bound. Then, after each accessed tuple ρ_i , $\text{FR}^*::\text{UpdateBound}(\rho_i)$ returns a tight upper bound for $\mathcal{S}(\tau)$ where $\tau = \tau_1 \bowtie \tau_2$ and $\tau_1 \in R_1 - HR_1 \vee \tau_2 \in R_2 - HR_2$.*

4.2.2 The PA Pulling Strategy

The motivation behind the PA strategy comes from the experimental results shown in Figure 2(a). We observe that the “blind” round-robin strategy accesses both inputs equally, although there is evidence from the adaptive strategy of HRJN^* that the left input is less important for computing a solution. Thus, the goal is to design an adaptive strategy that works with the FR^* bound and yields strong guarantees on the performance of the rank join operator.

At a high level, PA accesses tuples from the input relations based on a metric that quantifies the potential of each relation to generate a high scoring result. More concretely, we define the “potential” metric $\text{pot}_i = \max\{t_i, t_{\text{both}}\}$ that measures the maximum score of a result tuple using a tuple in $R_i - HR_i$. The PA pulling strategy is then defined as follows: Choose the input that has the maximal potential, breaking ties in favor of the input with the least depth, then the input with the least index.

Intuitively, PA can be viewed as a generalization of the HRJN^* pulling strategy to the feasible-region bound. However, PA utilizes

different metrics to measure the potential of each input, and thus we expect the two strategies to make different choices. Moreover, we are able to prove strong optimality properties for the performance of PA, whereas no such results are known for the HRJN^* strategy.

4.3 Analysis of FRPA

In this section, we analyze the performance of the proposed FRPA operator in terms of the sumDepths metric and overall complexity.

Recall that the adaptive strategy of FRPA aims to avoid the unnecessary accesses of $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. Thus, we expect intuitively that the new operator will be instance-optimal given that $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ is instance-optimal. Indeed, we show that FRPA is instance-optimal with a ratio of 2 within the same class of inputs and algorithms as $\text{PBRJ}_{\text{FR}}^{\text{RR}}$. Moreover, we are able to prove a much stronger result: FRPA can never access more tuples than $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ on any of the two inputs. This implies that the new operator dominates $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ in terms of I/O cost. These theoretical guarantees provide compelling evidence in favor of FRPA. The results are of general interest as well, since, to the best of our knowledge, FRPA is the first rank join operator with an adaptive pulling strategy that is instance-optimal within the same class as $\text{PBRJ}_{\text{FR}}^{\text{RR}}$.

The following theorems formalize our results. We omit some of the details in the proofs in the interest of space. The complete proofs can be found in the full version of this paper [5].

THEOREM 4.2. *Let $I = (R_1, R_2, \mathcal{S}, K)$ be a rank join instance. Then $\text{depth}(I, \text{FRPA}, i) \leq \text{depth}(I, \text{PBRJ}_{\text{FR}}^{\text{RR}}, i)$ for $1 \leq i \leq 2$.*

PROOF. (Sketch) By contradiction. Assume that there exists an input k such that $\text{depth}(I, \text{PBRJ}^*, k) > \text{depth}(I, \text{PBRJ}_{\text{FR}}^{\text{RR}}, k)$. Let \bar{k} be the opposing input. For each i , we define $r_i = \text{depth}(I, \text{PBRJ}_{\text{FR}}^{\text{RR}}, i)$, and p_i as the depth of FRPA on input R_i right before it pulls tuple $r_k + 1$ on input k . Hence, $p_k = r_k$. Let t be the bound of FRPA before the next pull which is computed as $\max\{t_k, t_{\bar{k}}, t_{\text{both}}\}$. By definition, $\text{pot}_k = \max\{t_k, t_{\text{both}}\}$ and $\text{pot}_{\bar{k}} = \{t_{\bar{k}}, t_{\text{both}}\}$.

The proof proceeds by showing two claims: $t_k \leq \mathcal{S}^{\text{term}}$, and $\text{pot}_k > \text{pot}_{\bar{k}}$. We also know that $\text{pot}_k \geq \mathcal{S}^{\text{term}}$ otherwise the algorithm would not pull from input k . We distinguish the following cases:

$\text{pot}_k = t_k$. It follows that $\text{pot}_k = \mathcal{S}^{\text{term}}$ and $\text{pot}_{\bar{k}} < \mathcal{S}^{\text{term}}$. The latter implies that no top results exist that use $HR_{\bar{k}}^{\text{RR}} - HR_{\bar{k}}$. At this point, FRPA has already computed the same solution as $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ and the bound indicates that no better results exist from either of the two inputs. Hence, the algorithm should not pull further than p_k which is a contradiction.

$\text{pot}_k = t_{\text{both}}$. Given that $\text{pot}_k > \text{pot}_{\bar{k}}$, it has to be that $t_{\text{both}} > \max\{t_{\text{both}}, t_{\bar{k}}\}$ which is a contradiction. \square

THEOREM 4.3. *FRPA is instance-optimal within the same class of algorithms and inputs as $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ with the same optimality ratio, assuming that the cost of an algorithm is measured with the sumDepths metric.*

PROOF. Let \mathcal{A} and \mathcal{I} be the classes of algorithms and inputs, respectively, for which $\text{PBRJ}_{\text{FR}}^{\text{RR}}$ is instance-optimal. Let $I \in \mathcal{I}$ be a problem instance and let $A \in \mathcal{A}$ be the optimal algorithm for solving I . Instance-optimality implies that $\text{sumDepths}(\text{PBRJ}_{\text{FR}}^{\text{RR}}, I) \leq 2 \cdot \text{sumDepths}(A, I) + c$ for a constant c that is independent of I and A . It follows from Theorem 4.2 that $\text{sumDepths}(\text{FRPA}, I) \leq \text{sumDepths}(\text{PBRJ}_{\text{FR}}^{\text{RR}}, I)$, and hence $\text{sumDepths}(\text{FRPA}, I) \leq 2 \cdot \text{sumDepths}(A, I) + c$. Since I was chosen arbitrarily, we can conclude that FRPA is instance-optimal with a ratio of 2. \square

We examine next the computational complexity of the new rank join operator. Our discussion focuses on the FR^* bound, since the complexity of the adaptive pulling strategy is constant. It is straightforward to show that FR^* is in PTIME under data complexity, similarly to FR. The analysis, however, indicates that FR^* inherits the same worst case. More concretely, the complexity of FR^* involves an exponential dependency to the maximum number of score attributes in the two inputs, which essentially stems from the maximum theoretical size of the covers CR_1 and CR_2 . Even though this theoretical maximum is loose, we have observed empirically that the sizes of the covers can grow very quickly with the number of score attributes. As a concrete example, for one specific experiment we observed that $|CR_1| + |CR_2|$ grew by an order of magnitude when the number of score attributes increased by 1. This growth has a negative effect on the overall execution time of the operator and in certain cases cancels the benefits of reduced pulling.

5. THE ADAPTIVE a-FRPA OPERATOR

As mentioned in the concluding remarks of the previous section, the increased complexity of the FR^* bound can affect negatively the performance of the rank join operator. It is interesting to ponder the development of a tight bounding scheme with low computational complexity, but the hardness results presented in [13] make this an unpromising option. Hence, we adopt a different approach and introduce a heuristic that attempts to control the complexity of the FR^* bound at the potential expense of tightness.

More concretely, we develop a new bounding scheme, termed the Adaptive Feasible Region bound and denoted as aFR, that maintains covers CR_1 and CR_2 of bounded size. The new bounding scheme is compatible with the adaptive pulling strategy of the previous section and thus gives rise to the a-FRPA operator. The basic idea behind aFR is to associate each CR_i to a *resolution* that controls its size. The resolution is initially high and is decreased accordingly as the cover grows in size. The tradeoff is that the resulting CR_i may not enclose tightly the space of unseen tuple scores, which in turn makes the aFR bound loose.

The adaptation achieved by the a-FRPA operator has the following interesting property: a-FRPA behaves exactly like FRPA when both covers are small, and exactly like the HRJN* operator when both covers grow very large. Thus, a-FRPA represents an interesting hybrid that can adapt its behavior gradually from a provably instance-optimal operator to a computationally efficient operator, depending on the characteristics of the input.

5.1 The aFR Bound

This section describes the adaptive aFR bound. We first present the main intuitions behind the design of aFR and then discuss its details.

5.1.1 Design Overview

We begin our presentation with two alternative bounding schemes that address the same problem – constraining the size of CR_i . Our goal is to present the shortcomings of these straightforward solutions and also to develop the intuition behind the aFR bound.

One solution is to simply stop updating CR_i in $FR^*::UpdateCR$ if $|CR_i|$ exceeds a threshold. The obvious shortcoming is that the feasible region “freezes” past some point, and thus fails to capture accurately the score vectors in $b[R_i - HR_i]$ as HR_i grows.

Another solution is to maintain the cover over a grid that quantizes the space of score vectors. More concretely, each dimension is quantized in a fixed number of intervals and all cover computations inside $FR^*::UpdateCR$ occur at the corners of the grid. The

resolution of the grid limits the total number of possible score vectors and thus constrains $|CR_i|$. However, this mechanism requires a very coarse grid resolution in order to guarantee that $|CR_i|$ always remains below a specific size. For instance, to limit $|CR_i|$ below 500 for 3 score attributes, each dimension must be quantized to 8 intervals. A coarse resolution provides a loose approximation of the actual feasible region, which in turn affects the usefulness of cover bounds. Moreover, this coarse resolution is used even if the precise CR_i would be small in size.

The proposed aFR bound essentially combines the aforementioned solutions in an attempt to overcome their shortcomings. It employs a grid to quantize the space of score vectors and thus limit the size of CR_i . The difference is that the resolution of the grid is not fixed, but it is adapted dynamically so that the size of CR_i is always bounded by some system-defined threshold. Thus, CR_i continues to evolve as HR_i is updated, and there is an effort to keep CR_i as “detailed” as possible given the size threshold and the characteristics of the input.

5.1.2 The Grid Tree Structure

The maintenance of an adaptive cover is done with the *grid tree* data structure that we discuss in the following paragraphs. Our presentation assumes that the grid tree is used to maintain a cover for some set X of points that reside in the unit hyper-cube of e dimensions. In the context of a-FRPA, X will represent the score vectors $b[R_i - HR_i]$ and e the number of base scores in R_i .

A grid tree is a quad-tree over the unit hyper-rectangle¹ comprising L_0 levels, where L_0 is a parameter of the structure. We use u^{root} to denote the root node of the tree and assume that it resides at level 0. The quad-tree organization allows us to view the nodes at each level l as a uniform grid $2^l \times \dots \times 2^l$. We call l the resolution of this particular grid. Thus, the grid tree represents several grids with resolution $0, \dots, L_0 - 1$. Only one resolution is active at each point in time and it is denoted as L , $0 \leq L < L_0$. Initially, $L = L_0 - 1$.

A node u at level L is either marked or unmarked. A marked u contributes a cover point in CR_i whose coordinates are denoted as $b(u)$ and are equal to the upper-right corner of u . An internal node at level $l < L$ is implicitly marked if it has a descendant at level L that is marked. We use $coverPoints(u^{\text{root}}, l)$ to denote the set of cover points inferred by the marked nodes at level l .

We denote the level of a node u as $u.level$. Each node also records an integer counter $u.covered$ defined as the number of adjacent cells v such that $u \preceq v \wedge u \not\prec v \wedge (v \text{ is marked} \vee v \text{ is dominated by a marked node})$. In other words, $u.covered$ records the number of directly adjacent cells that are either covered or dominated by a covered cell. The grid tree always satisfies the following invariant for the nodes at level L :

Grid Tree Invariant If a node u is marked then $u.covered = 0$.

This invariant ensures that no point $c \in coverPoints(u^{\text{root}}, L)$ dominates any other point c' in the same set. Thus, the cover is a skyline of points, which is beneficial for the computation of cover bounds as explained in Section 4.2.1.

The grid tree can be initialized by marking nodes at level L such that the invariant is satisfied. Subsequently, function `UpdateGridCR` (shown in Figure 7) can be invoked to update the set of marked nodes according to a score vector s such that $\exists x \in X : s \preceq x$. In the context of a rank join operator, $s = b(\tau)$ for a tuple $\tau \in G_i$, and thus `UpdateGridCR` has a similar role as $FR^*::UpdateCR$. The function uses the quad-tree structure to efficiently identify the

¹Each node in the quad-tree has a fan-out of 2^e .

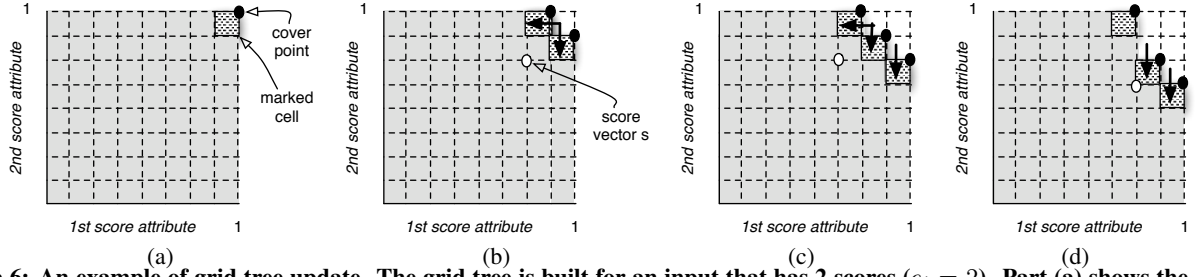


Figure 6: An example of grid tree update. The grid tree is built for an input that has 2 scores ($e_i = 2$). Part (a) shows the original tree where the only marked cell corresponds to the cover point $(1, 1)$ and assuming that the resolution is $L = 3$. Parts (b–d) show the state of the grid tree after it is updated with three score vectors. Each part shows the updated marked nodes and the implied cover points, as well as the change from the previous marked nodes. In all figures, the gray region depicts the feasible region for the scores of unseen tuples.

```

Procedure aFR::updateGridCR( $s, u, L$ )
Input: Score vector  $s$ . Node  $u$  of the quad-tree index. On the first
call,  $u$  should be the root of the quad-tree. The depth  $L$  of the
quad-tree index.
1 if  $p \not\prec b[u]$  then return ;
2 if  $u.level < L$  then //  $u$  is an internal node
3   if  $u$  is unmarked then return ;
4   foreach child  $v$  of  $u$  in dominance order do
5     aFR::UpdateGridCR( $s, v, L$ );
6 else if  $u.level = L$  then //  $u$  is a grid cell
7   if  $u$  is unmarked then return ;
8   unmark  $u$ ;
9   foreach  $v : v \prec u \wedge v \not\prec s$  do
10    decrease  $v.covered$  ;
11    if  $v.covered = 0$  then mark  $v$ ;

```

Figure 7: Maintenance of a grid cover.

marked nodes at level L that dominate s . These nodes are unmarked, and their dominated cells are then marked if the invariant is not violated. This has the effect of “sliding” the induced cover points, as shown in Figure 6. The complexity of processing an update is linear to the number of marked cells that dominate the input score vector s , which is controlled by the current resolution $L \leq L_0$. Overall, this results in an efficient update mechanism with controlled space and time complexity, but the feasible region implied by $coverPoints(u^{root}, L)$ is not guaranteed to be tight.

We now prove some important properties on the correctness of the update mechanism. The first property states that we obtain a correct cover after the update.

THEOREM 5.1. *Let X be a set of score vectors and assume that $coverPoints(u^{root}, L)$ provides a cover for X . Let s be a score vector such that $\nexists x \in X : s \preceq x$. Then, after the completion of $UpdateGridCR(s, u^{root}, L)$, $coverPoints(u^{root}, L)$ remains a cover for X .*

PROOF. (Sketch) The proof works by contradiction. Let C' denote the set $coverPoints(u^{root}, L)$ before the update call, and C denote the set after the call respectively. Assume that there exists a vector $x \in X$ that is not covered by any point in C . Since C' is a correct cover, then there was at least one vector $c \in C'$ such that $x \preceq c$. Since x is not covered, this implies that c was unmarked by the update call, which implies that $s \not\prec c$ from line 1. It follows from the assumptions of the theorem that $x \not\prec c$. Also, because s is quantized on the grid, it follows that there exists at least one

dominated neighbor c' of c such that $x \preceq c'$. Since c was originally marked, it implies that c' is covered, i.e., $c'.covered > 0$. It is possible to show that either $c'.covered > 0$ or c' is marked after the call, which implies that x is dominated by at least one marked cell in C' and this contradicts our original assumption. \square

The second property follows directly from the definition of the algorithm and states that the grid tree invariant continues to hold. This ensures that $coverPoints(u^{root}, L)$ forms a skyline which is desirable for the efficient computation of cover bounds.

LEMMA 5.1. *The grid tree invariant continues to hold on u^{root} after the completion of $UpdateGridCR(s, u^{root}, L)$.*

5.1.3 Definition of the aFR Bound

We are now ready to define the aFR bound. As hinted earlier, aFR uses a separate grid tree for each input R_i to maintain an adaptive cover of $b(R_i - HR_i)$. The main idea is to start with a grid tree of high resolution and then to gradually reduce it in order to maintain $|CR_i|$ below a given threshold.

The functions comprising the bounding scheme are shown in Figure 8. The main $aFR::UpdateBound$ function follows closely the logic and data structures of $FR^*::UpdateBound$. The difference lies in function $aFR::UpdateCR$ that maintains the adaptive cover. Initially, the cover is maintained as in $FR^*::UpdateCR$ which ensures that CR_i encloses tightly the feasible region of $b(R_i - HR_i)$. If $|CR_i|$ exceeds a predefined threshold $maxCRSize$, then the cover points are transferred to a grid tree u_i^{root} which maintains an adaptive cover of $b[R_i - HR_i]$ from that point onward. If the adaptive cover also exceeds the size threshold, then the resolution of the grid tree is reduced and the grid tree is reinitialized. We note that FR^* employs a separate copy of this mechanism for each input, which makes it possible to maintain a precise cover for one input while using an adaptive cover on the other.

At the limit, the resolution of the grid-tree can be reduced down to the minimum value 0. In this case, $coverPoints(u_i^{root}, 0)$ contains just the point $(1, \dots, 1)$ and thus the aFR bound becomes the same as the corner bound of the HRJN* algorithm. Thus, an interesting property of the aFR bound is that it can adapt its computation from the tight feasible region bound to the loose and straightforward corner bound.

The definition of the bound employs two more functions, namely $aFR::InitializeGridCR$ and $aFR::ResultBound$. The former traverses the grid tree after a set of points is marked and ensures that the grid tree invariant is enforced. The second function computes the final value of the bound and its definition is the same as

```

Function aFR::UpdateBound( $\rho_i$ )
Input: Newly accessed tuple  $\rho_i$ 
Output: A bound on the score of unseen join results
Data: Variables  $G_i, g_i, CR_i$ , and  $SHR_i$  defined the same as for
FR*::UpdateBound; A positive integer  $maxCRSize$ .
// Same as FR*::UpdateBound in Figure 5 except
for the following lines
3  $CR_i \leftarrow$  aFR::UpdateCR( $CR_i, b[G_i], maxCRSize$ );
8 return aFR::ResultBound()

Function aFR::UpdateCR( $C_i, Y, maxCRSize$ )
Input: A cover  $C_i$  for the unseen portion of  $R_i$ ; A set of score vectors
 $Y$ ; A positive integer  $maxCRSize$ .
Output: An updated cover for  $R_i$  that contains at most  $maxCRSize$ 
points.
Data: Current resolution  $L_i$  for the grid cover, initialized to  $\infty$ . A
grid tree  $u_i^{root}$  of maximum resolution  $L^0$ .
1 if  $L_i = \infty$  then
2    $C \leftarrow$  FR*::UpdateCR( $C_i, Y$ );
3   if  $|C| > maxCRSize$  then
4      $L_i \leftarrow L^0$ ;
5     Mark the nodes at level  $L_i$  corresponding to  $C$ ;
6     aFR::InitializeGridCR( $u_i^{root}, L_i$ );
7      $C \leftarrow coverPoints(u_i^{root}, L_i)$ ;
8 else
9   foreach  $s \in Y$  do aFR::UpdateGridCR( $s, u_i^{root}, L_i$ );
10   $C \leftarrow coverPoints(u_i^{root}, L_i)$ ;
11 while  $|C| > maxCRSize$  do
12  Mark the nodes at level  $L_i - 1$  of  $u_i^{root}$  that contain a marked
node at level  $L_i$ ;
13  aFR::InitializeGridCR( $u_i^{root}, L_i - 1$ );
14   $L_i \leftarrow L_i - 1$ ;
15   $C \leftarrow coverPoints(u_i^{root}, L_i)$ ;
16 return  $C$ 

Procedure aFR::InitializeGridCR( $u, L$ )
Input: Node  $u$  of the grid tree. On the first call,  $u$  should be the root
of the grid tree. Resolution  $L$  of the grid tree.
1 if  $u.level = L$  then
2    $u.covered \leftarrow |\{v \mid u \prec v \wedge u \not\prec v\}|$ ;
3    $v \wedge (v \text{ is marked} \vee v.covered > 0)$ ;
4   if  $u$  is marked and  $v.covered > 0$  then unmark  $u$ ;
5 else
6   foreach  $child\ v$  of  $u$  in dominance order do
7     aFR::InitializeGridCR( $v, L$ );

Function aFR::ResultBound()
Output: The value of the aFR bound based on  $HR_{\{1,2\}}$  and  $CR_{\{1,2\}}$ 
// Same as FR*::ResultBound

```

Figure 8: The aFR bound.

FR*::ResultBound. The difference is that the contents of CR_i may come from the marked nodes in the grid tree structure.

5.2 The a-FRPA Operator

The a-FRPA operator is the instantiation of the PBRJ template using the aFR bounding scheme and the PA pulling strategy. (Thus, it can be denoted as PBRJ_{aFR}^{PA} in the notation of [13].) We note that it is straightforward to adapt the PA strategy to the new aFR bound by simply defining each potential metric pot_i in terms of the bounds computed in aFR::resultBound.

Parameter	Possible Values (Default in bold)
e : Number of score attributes	1, 2 , 3, 4
c : Score cut	.25, .5 , .75, 1
z : Skew of score distribution	0, .5 , 1
K : Number of results	1, 10 , 100, 1000

Table 2: Parameters used in the experimental study. Default values are shown in bold.

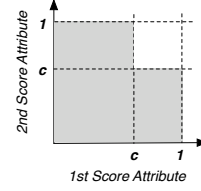


Figure 9: Illustration of the effect of parameter c for $e = 2$. The gray region contains the randomly generated score vectors that appear in the base data.

Based on the definition of the aFR bound, it is clear that a-FRPA behaves exactly the same as FRPA as long as the size of each CR_i is below the threshold parameter $maxCRSize$. Under these conditions, FRPA inherits the same nice theoretical properties and is thus instance-optimal. Once the size threshold is exceeded, aFR is not guaranteed to be tight and hence FRPA is not instance-optimal. However, a-FRPA is able to compute its bound much faster given that it employs a fast cover update mechanism and the size of each CR_i is always constrained.

As mentioned above, a-FRPA behaves exactly like the instance-optimal FRPA operator if neither of the two covers are reduced to a grid-tree. Another interesting observation is that a-FRPA behaves precisely like the HRJN* operator if the resolution of both adaptive covers is reduced to 0. Thus, a-FRPA can adapt its behavior from an instance-optimal operator to a computationally efficient operator depending on the characteristics of the input.

6. EXPERIMENTAL STUDY

In this section, we present the results of an experimental study that we conducted in order to validate the effectiveness of the presented rank join operators. Our study addresses the following high-level questions: is the adaptive operator a-FRPA more efficient than FRPA (Section 6.2.1); do the new operators compare favorably to the existing operators PBRJ_{FR}^{RR} and HRJN* (Section 6.2.2); and, how well do the new operators perform when used in pipelined physical plans (Section 6.2.3).

In what follows, we first describe the experimental methodology and then present the results of the study.

6.1 Methodology

We describe the data sets, queries, techniques, and evaluation metrics that we use in our experimental study. The parameters of our methodology are summarized in Table 2.

Data. The experimental study employs the well known TPC-H data set. We generate several random instances using Vivek Narasayya's data generator² that injects skew in the distribution of values and joins. The data scale factor to is set to 1.

²ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew

We further extend the relations in the TPC-H schema with several score attributes and assign to them random values for each tuple. The generation of score values is controlled by three parameters termed e , z , and c . Parameter e controls the number of score attributes per relation. For a specific e , we generate the score vector of each tuple by drawing its score values independently at random from a Zipfian distribution of skew z . The only constraint is that we do not generate any score vectors that dominate the point (c, c, \dots, c) specified by the third parameter. Thus, the generated score vectors lie in the hyper-cube of e dimensions except that a hyper-rectangle of volume c^e is removed from the upper-right corner. Figure 9 illustrates this idea for $e = 2$. As shown, the resulting score vectors match the intuition for real-life data sets: Certain score vectors may attain the maximum score of 1 for some coordinates, but there is a trade-off between the score attributes that precludes the ideal point $(1, 1, \dots, 1)$ from appearing.

We note that the size of the data set is not a parameter of our study. This happens because a rank join operator accesses only a prefix of each relation, and thus its performance is affected primarily by the number of top results (parameter K) and the distribution of scores (parameters e , z , and c).

Rank Joins. Our experiments employ instances of the rank join problem on the aforementioned data. We set R_1 and R_2 to the Lineitem and Orders relations respectively, because they are the largest tables and thus have enough variability to create interesting problem instances. The \mathcal{S} function simply sums up the score attributes from the two inputs. (Recall that each input has e score attributes as described previously.) The number K of results is a parameter that we vary depending on the experiment.

Techniques. The evaluation study compares four different rank join operators: the FRPA and a-FRPA operators that we introduce in this paper, and the existing operators PBRJ_{FR}^{RR} [13] and HRJN* [8]. As we saw earlier, PBRJ_{FR}^{RR} is the only instance-optimal operator in the existing literature, and HRJN* has been shown to perform well in practice even though it is not instance-optimal.

The above operators access each input in sorted order of \bar{S} by scanning a clustered index of the respective relation. (We build a separate index for each value of e .) This setting represents a best-case scenario for the cost of I/O and gives an advantage to the competitor algorithms as we explain later. In practice we expect that access to each R_i will be costlier, e.g., it may come through an unclustered index, or it may be streamed over the network similar to the middleware setting described by Fagin et al. [4]

The operators are implemented in C++ and compiled with g++ and -O2 optimization flag. We use the following experimental platform: Ubuntu Linux version 8.04, Intel Quad-Core 2.5GHz, 4GB of RAM, and a single 500GB 7200RPM (SATA) disk.

Evaluation Metrics. We measure the performance of the above operators using two metrics, namely, *sumDepths* and wall clock execution time. The metrics are measured after the completion of the K -th getNext call to the rank join operator. The *sumDepths* metric measures the total number of tuple pulls and thus provides a system-independent metric for I/O. The wall clock execution time is measured with a cold cache for each operator and it indicates the efficiency of the operator on our experimental platform.

To attain some statistical robustness and to guard against outliers in the generation of score values, we repeat each experiment using five different random data instances. All five instances conform to the same experimental parameters but employ a different seed for the generation of score values. For each experiment, we report the average of each metric over the corresponding five instances.

6.2 Results

6.2.1 Performance of Adaptive Feasible Region Bound

This set of experiments evaluates the performance of a-FRPA that employs the novel adaptive feasible region bound. The first goal is to assess the sensitivity of the operator to its parameters, namely the initial resolution L_0 and the size threshold *maxCRSize*. We also compare the operator against FRPA, in order to examine the performance implications of the loose adaptive bound. We set $e = 3$ in the experiments that follow, since we want to stress test the ability of a-FRPA to maintain an adaptive cover for each input.

Figure 10 shows the performance of a-FRPA as we vary the parameter *maxCRSize*. The initial resolution is kept fixed at $L_0 = 64$. As expected, the *sumDepths* metric decreases as the threshold increases, since a high threshold allows a-FRPA to maintain a more detailed cover for each input. The execution time graph, however, reveals that the overhead of maintaining the detailed cover outweighs the benefits of reduced I/O, thus increasing execution time. These results suggest that smaller thresholds are likely to be more effective.

We also observe that a-FRPA tracks closely the instance-optimal depth of FRPA as long as the threshold value is sufficiently large (in this experiment, greater than 500). The increased depth, however, is countered with the far more efficient computation of the adaptive cover bound. As a result, the total execution time of a-FRPA is far lower compared to FRPA. As an example, for *maxCRSize* = 500, a-FRPA improves on the execution time of FRPA by 48%, even though it performs 23% more pulls on both inputs.

Figure 11 shows the performance of a-FRPA as we vary the initial resolution L_0 . The size threshold is kept fixed at 500. The *sumDepths* metric is rather insensitive to L_0 , since the final resolution of the adaptive cover bound is determined by the size threshold which is fixed at 500. This behavior implies that a lower initial resolution is likely to be more efficient, as there is less adaptation to the final resolution. This intuition is verified by the measured execution times.

Compared to FRPA, we observe a similar trend as the previous experiment. More concretely, a-FRPA has a higher *sumDepths* metric, but the efficiency of the aFR bound offsets the additional I/O cost and results in a lower total execution time.

We conducted more experiments with different values for L_0 and *maxCRSize* and obtained similar results. We also tested a-FRPA against the naive solutions mentioned in Section 5, namely, maintaining a precise CR_i up to a specific size, and using a grid of a fixed resolution. In all cases, the adaptive cover of a-FRPA performed better in both of our evaluation metrics.

Overall, the results demonstrate that a-FRPA provides an effective hybrid between the instance-optimal I/O of FRPA and computational efficiency. Based on the measurements, we henceforth instantiate a-FRPA using *maxCRSize* = 500 and $L_0 = 64$. We note that these were not the optimal settings for the specific experiments, but we want to avoid over-fitting the operator to the parameters of the empirical study.

6.2.2 Comparative Evaluation

The next experiments compare the FRPA and a-FRPA operators introduced in this paper against the state-of-the-art operators HRJN* and PBRJ_{FR}^{RR}. The evaluation is done relative to the parameters shown in Table 2.

Effect of score cut (c). Figure 12 shows the performance of the four rank join operators as a function of the score cut c . (See Figure 9 for an explanation of the parameter.) We begin our discussion with the results for the *sumDepths* metric. Compared to HRJN*,

the new operators incur similar I/O for $c = 1$, but the difference increases with lower values of c and reaches one order of magnitude for $c = 0.5$. The inefficiency of HRJN* is due to its bounding scheme, which makes the often unrealistic assumption that the ideal score vector $(1, 1, \dots, 1)$ is present in the two inputs. This assumption causes the bound to drop at a slower rate, which in turn makes HRJN* “reach” deeper in each input. The new operators, on the other hand, employ the feasible region bound, which adapts its computation automatically to the scores actually present in the input. This adaptation yields a tighter bound, which in turn allows FRPA and a-FRPA to terminate much earlier.

The two new operators also outperform the existing PBRJ_{FR}^{RR} operator in all experiments. PBRJ_{FR}^{RR} uses the same bound as FRPA and a-FRPA, but its round-robin pulling strategy leads to more I/O compared to the adaptive strategy of the new operators. The difference is significant and ranges from 7K to 100K tuples. These results validate the effectiveness of the new strategy in avoiding the “useless” pulls of PBRJ_{FR}^{RR}. We note that a-FRPA has exactly the same I/O performance as FRPA in this experiment, since the cover of each input remains below the threshold of 500 points.

The examination of the execution times in Figure 12 yields similar observations. HRJN* incurs the highest execution time due to its I/O overhead, followed by PBRJ_{FR}^{RR} which has a high computational overhead for the FR bound. The new operators are the most efficient, since they incur less I/O and they have a low computational overhead. As a concrete example, for $c = .75$, FRPA is 53% more efficient compared to PBRJ_{FR}^{RR} and more than 3× more efficient compared to HRJN*. (a-FRPA behaves the same as FRPA and thus has the same efficiency.) We note that the difference in execution time would be amplified if the tuple accesses were costlier—recall that we employ clustered indices which represent a best-case scenario for the cost of I/O.

Effect of number of base scores per input (e). Figure 13 shows the performance of the four rank join operators as a function of e . The results for *sumDepths* demonstrate that the new operators improve on both HRJN* and PBRJ_{FR}^{RR} by a significant margin for $1 \leq e \leq 3$. For instance, the improvement is one order of magnitude and 75% respectively when $e = 1$. The inefficiency of HRJN* is due again to its corner bounding scheme, which assumes that the maximum score vector $(1, 1, \dots, 1)$ is present in both inputs. PBRJ_{FR}^{RR} is able to track the input scores more accurately through the feasible region bound, but its round-robin pulling strategy leads to more I/O than necessary. The new operators address both shortcomings: they employ the feasible region bound to track scores accurately, and they use adaptive pulling to avoid useless pulls. The same trend appears in the total execution time of the operators.

The results for $e = 4$ reveal a different picture. We note that PBRJ_{FR}^{RR} and FRPA are omitted from the graph because they required more than 10 hours to complete. Essentially, the covers CR_i explode in size and their maintenance becomes prohibitively expensive. a-FRPA is able to handle this explosion through its adaptive bound, whose space complexity is constrained by the *maxCRSize* parameter (500). In this case, however, the 4-dimensional cover does not translate in savings in terms of the *sumDepths* metric, which in turn makes the total execution time of a-FRPA similar to that of HRJN*. This trend suggests that the latter is the operator of choice when both inputs have a high number of score attributes (although the frequency of this setting in practice is not clear).

Effect of number of results (K). Figure 14 shows the performance of the four operators as we vary K , the number of retrieved results. The results indicate that FRPA and a-FRPA yield substantial improvements both in terms of I/O and total execution time,

thus validating the trends that we observed in the previous experiments. As a concrete data point, a-FRPA improves the I/O of HRJN* and PBRJ_{FR}^{RR} by 3.5× and 64% respectively, and execution time by 3× and 66% respectively. The FRPA operator offers similar improvements.

Effect of score skew (z). Our experiments with the skew z of the score distribution showed qualitatively the same results as for the default setting $z = 0.5$. We omit them in the interest of space.

6.2.3 Evaluation of Pipelined Plans

The final experiments evaluate the performance of the different operators when they are composed in a pipelined physical plan. More concretely, we consider three ranking queries over the following natural joins: $L \bowtie O$, $L \bowtie O \bowtie C$, and $L \bowtie O \bowtie C \bowtie P$ (table symbols correspond to the initial letter of Lineitem, Orders, Customers, and Part respectively). Each relation has exactly one score attribute ($e = 1$) whose values are generated using the same process as the previous experiments with $z = 0.5$, and $c = 0.5$. The ranking function \mathcal{S} aggregates base scores across all relations. For each rank join problem, we create a physical plan that pipelines several binary rank join operators of the same type. The K results are obtained by invoking the top operator’s *getNext* method an equal number of times.

The experiments that follow employ solely HRJN* and a-FRPA as the rank join operators for the binary plans. We omit FRPA and PBRJ_{FR}^{RR}, since the previous experiments showed that a-FRPA is more efficient in practice.

Figure 15 shows the performance of the HRJN* and a-FRPA physical plans for the three ranking queries that we consider. We fix $K = 10$. The results demonstrate that a-FRPA remains far more efficient than HRJN* when used in a pipelined plan, both in terms of I/O and total execution time. As an example, the new operator improves I/O by 5× and execution time by 6.7× respectively for the 3-way rank join. The reason is similar as in the previous experiments: Effective tracking of the input scores through the feasible regions of each input, which in turn allows the operator to terminate early. Moreover, the grid tree mechanism allows a-FRPA to track each feasible region efficiently.

We note that we performed experiments varying other parameters of interest and observed similar trends as the previous results.

7. CONCLUSIONS

In this paper, we examined rank join operators by considering both I/O cost and total execution time. Our empirical study of existing state-of-the-art operators showed that they are not effective when evaluated under both metrics. This led to the creation of the FRPA operator which is instance-optimal and far more efficient than the currently known instance-optimal operator. We also introduced the a-FRPA operator that can adapt its behavior automatically between the instance-optimal FRPA operator and the empirically efficient HRJN* operator depending on the input. Our experimental results validated the effectiveness of the new operators and demonstrated that they offer significant performance improvements compared to existing operators.

Acknowledgments. We wish to thank Serge Abiteboul and Karl Schnaitter for their useful feedback in the development of this work. This work was partially supported by the National Science Foundation under Grant No. IIS-0447966p, and by an IBM Faculty Development Award.

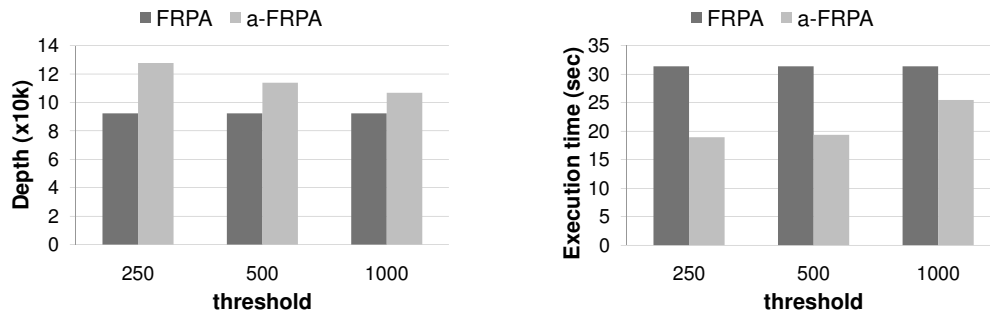


Figure 10: Performance of FRPA for different values of $maxCRSize$. L_0 is kept fixed at 64.

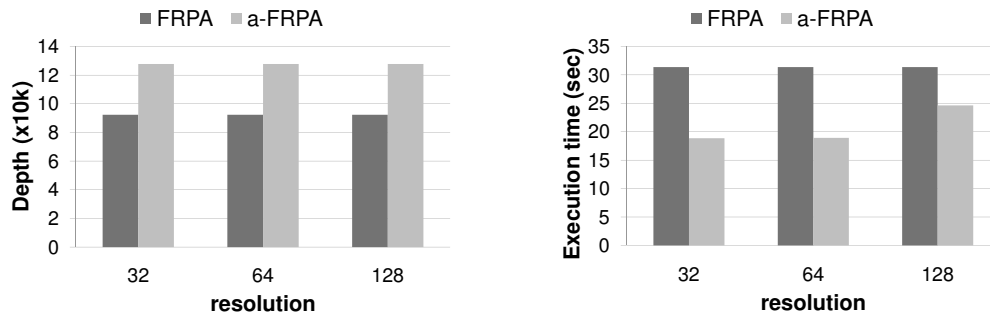


Figure 11: Performance of FRPA for different values of L_0 . Parameter $maxCRSize$ is kept fixed at 500.

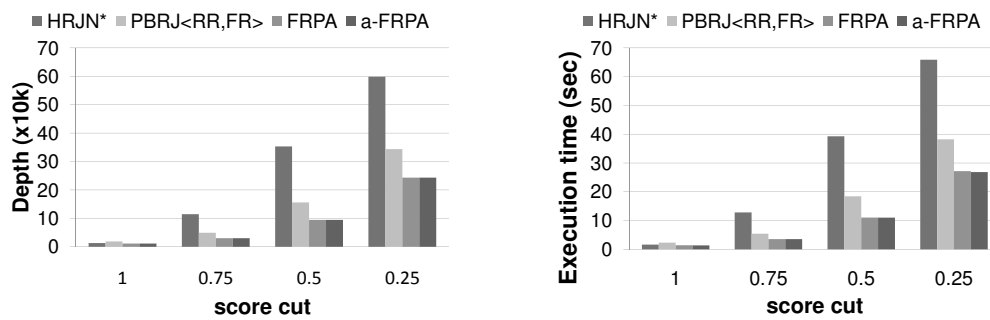


Figure 12: Effect of score cut c on performance of rank join operators. ($K = 10, z = .5, e = 2$.)

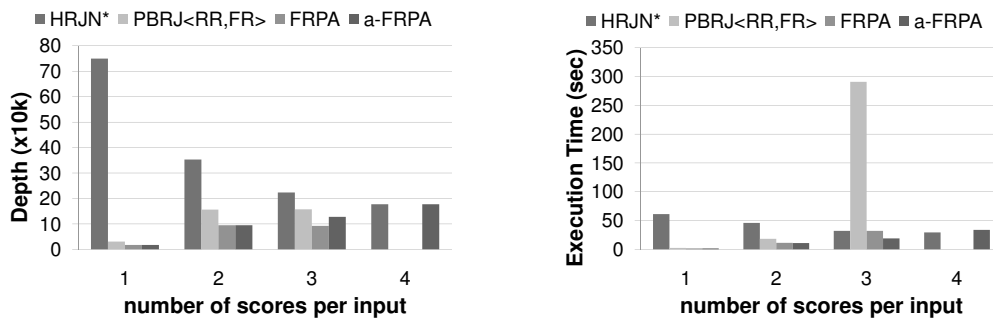


Figure 13: Effect of parameter e on performance of rank join operators ($K = 10, c = .5, z = .5$).

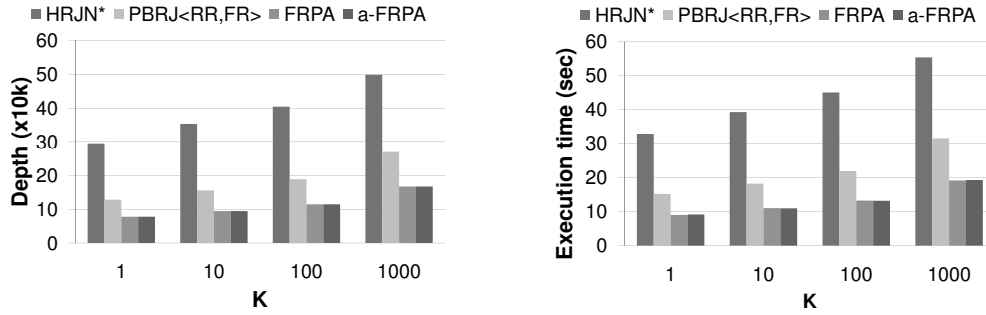


Figure 14: Effect of K on the performance of rank join operators ($z = .5, e = 2, c = .5$).

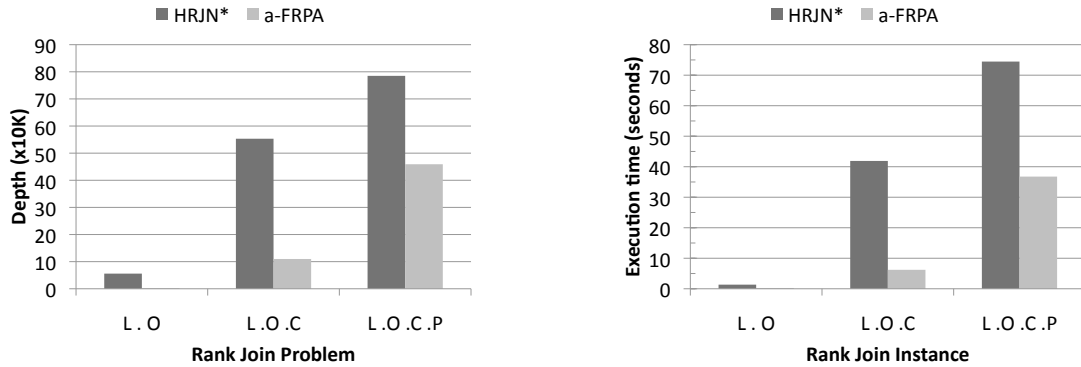


Figure 15: Performance of pipelined plans for different instances of the rank join problem.

8. REFERENCES

- [1] P. Agrawal and J. Widom. Confidence-aware joins in large uncertain databases. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, pages 628–639, 2009.
- [2] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 914–925, 2007.
- [3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, 2001.
- [4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th Symposium on Principles of Database Systems*, pages 102–113, 2001.
- [5] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. Technical Report UCSC-SOE-09-01, University of California Santa Cruz, 2009.
- [6] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.
- [7] M. Hua, J. Pei, A. W.-C. Fu, X. Lin, and H.-F. Leung. Efficiently answering top-k typicality queries on large databases. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 890–901, 2007.
- [8] I. Ilyas, W. Aref, and K. Elmagarmid. Supporting top-k join queries in relational databases. *International Journal on Very Large Databases*, 13(3):207–221, 2004.
- [9] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 275–286, 2002.
- [10] C. Li, K. C.-C. Chang, I. Ilyas, and S. Song. RankSQL: query algebra and optimization for relational top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 131–142, 2005.
- [11] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems*, 32(3):19, 2007.
- [12] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 281–290, 2001.
- [13] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *Proceedings of the 27th Symposium on Principles of Database Systems*, pages 43–52, 2008.
- [14] K. Schnaitter, J. Spiegel, and N. Polyzotis. Depth estimation for ranking query optimization. In *Proceedings of the 33rd International Conference on Very Large Databases*, pages 902–913, 2007.
- [15] M.A. Soliman, I.F. Ilyas, and K.C.-C. Chang. Probabilistic top-k and ranking-aggregate queries. *ACM Trans. Database Syst.*, 33(3):1–54, 2008.