Proactive Re-optimization with Rio

Shivnath Babu[†]
Stanford University
shivnath@cs.stanford.edu

Pedro Bizarro[‡]
University of Wisconsin – Madison
pedro@cs.wisc.edu

David DeWitt[‡]
University of Wisconsin – Madison
dewitt@cs.wisc.edu

ABSTRACT

Traditional query optimizers rely on the accuracy of estimated statistics of intermediate subexpressions to choose good query execution plans. This design often leads to suboptimal plan choices for complex queries since errors in estimates grow exponentially in the presence of skewed and correlated data distributions. We propose to demonstrate the *Rio* prototype database system that uses *proactive re-optimization* to address the problems with traditional optimizers. Rio supports three new techniques:

1. Intervals of uncertainty are considered around estimates of statistics during plan enumeration and costing
2. These intervals are used to pick execution plans that are robust to deviations of actual values of statistics from estimated values, or to defer the choice of execution plan until the uncertainty in estimates can be resolved
3. Statistics of intermediate subexpressions are collected quickly, accurately, and efficiently during query execution

These three features are fully functional in the current Rio prototype which is built using the *Predator* open-source DBMS [5]. In this proposal, we first describe the novel features of Rio, then we use an example query to illustrate the main aspects of our demonstration.

1. INTRODUCTION

Data sizes, query complexity, and the hardware resources to manage databases have grown dramatically in the last two decades. Unfortunately, query optimizers have not kept pace with the ability of database systems to execute complex queries over very large databases. Most current optimizers use a *plan-first execute-next* approach: the optimizer enumerates plans, estimates the cost of each plan, and executes the plan with lowest estimated cost. This approach, designed more than twenty years ago, relies heavily on the accuracy of estimated statistics of intermediate sub-expressions to choose good plans. It is well known that errors in estimates of intermediate statistics grow exponentially in the presence of skewed and correlated data distributions. Such estimation errors can result in suboptimal query plans that degrade the performance of complex queries by orders of magnitude [4].

[†] Supported by NSF grants IIS-0118173 and IIS-0324431.

[‡] Supported by NSF grant IIS-0086002.

Re-optimization is one promising technique to cope with optimizer mistakes. Re-optimization interleaves the optimization and execution stages of processing a query *Q*, possibly multiple times, over the running time of *Q*. Current re-optimizers take a *reactive* approach: they use a traditional optimizer to first generate a plan, then respond to suboptimalities detected in this plan during execution [3, 4]. This approach is handicapped by its use of an optimizer that is completely blind to issues affecting re-optimization. For example, consider the join of relations R and S. An *indexed nested-loop join (INLJ)* with R as the outer may outperform a *hybrid hash join (HHJ)* if R is small, S is large, and S has an (unclustered) index on the join attribute. As the size of R increases, the performance of HHJ quickly becomes better than that of INLJ. Suppose the size of R is unknown. Then, it may be better to use the *safe* HHJ instead of the *risky* INLJ because the INLJ would require the system to track statistics on R during execution, and in the worst case, an expensive re-optimization followed by a change of plan if R is much larger than estimated. Furthermore, the ability of reactive re-optimizers to collect statistics quickly and accurately during query execution is limited. Consequently, when re-optimization is triggered, the optimizer may make new mistakes, leading potentially to thrashing [1].

2. PROACTIVE RE-OPTIMIZATION

We have developed *Rio*, a prototype database system that uses a new approach called *proactive re-optimization* to process queries. Figure 1 shows the architecture of a proactive re-optimizer. Sections 2.1—2.3 discuss the three core components of a proactive re-optimizer. These components are described thoroughly in [1].

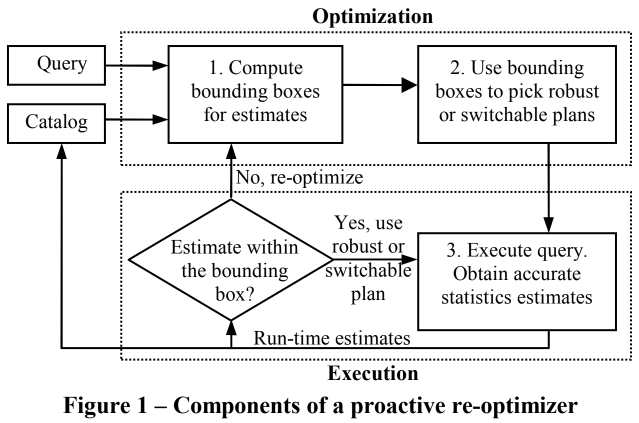

Optimization
Query
Catalog
1. Compute bounding boxes for estimates
2. Use bounding boxes to pick robust or switchable plans
No, re-optimize
Estimate within the bounding box?
Yes, use robust or switchable plan
3. Execute query. Obtain accurate statistics estimates
Run-time estimates
Execution

Figure 1 – Components of a proactive re-optimizer

2.1 Representing Uncertainty in Statistics

Current re-optimizers compute a single-point estimate for each statistic needed to cost plans. Rio accounts for possible errors in these estimates by considering intervals, or *bounding boxes*, around the estimates. Intuitively, the width of the bounding box is proportional to the uncertainty in the corresponding estimate. The uncertainty in each estimate is determined based on the way the estimate is derived. This approach was adapted from [3] and is illustrated by Example 1.

Example 1: Consider the query σ(R) ⋈ S where σ represents a set of selection predicates on relation R. Suppose an estimate |S| = 160MB is available in the catalog. However, in the absence of a multidimensional histogram on R, |σ(R)| is estimated to be 150MB based on estimates of the selectivities of the predicates in σ and from an assumption of independence among these predicates. Thus, the estimate of |σ(R)| is more uncertain than that of |S|. Figure 2 shows an example bounding box around the single-point estimate (150, 160) of |σ(R)| and |S|.

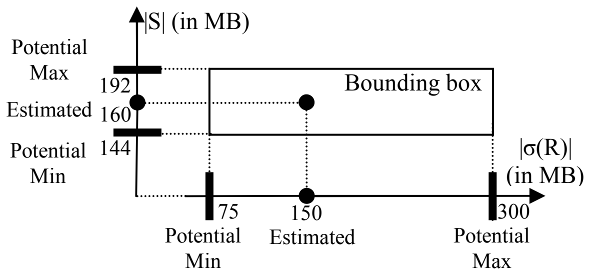


Figure 2 – Bounding box around estimates of |σ(R)| and |S| ∎

2.2 Robust and Switchable Plans

Since current re-optimizers consider single-point estimates only, their plan choices may lead to extra re-optimization steps and loss of work when actual statistics differ from their estimates. Rio uses bounding boxes during optimization to address this problem. During plan enumeration and costing, Rio tries to find plans that are *robust* or *switchable* based on the bounding boxes around uncertain estimates. A robust plan in a bounding box *B* is a single plan whose cost is close to optimal at all points within *B*. Intuitively, a switchable plan in *B* is a set *S* of plans with the following properties: a) for each point *pt* in *B*, there is a plan *p* in *S* whose cost at *pt* is close to that of the optimal plan at *pt*; b) the decision of which plan in *S* to use can be postponed until accurate estimates of uncertain statistics are available during query execution; and c) if the actual statistics lie within *B*, we can pick and run the appropriate member plan from *S* without losing any significant fraction of the execution work done so far.

Example 2: Consider Example 1. Let the buffer-cache size be Memory = 200MB. Suppose the size of S is known accurately from the catalog to be 160MB. Figure 3 shows the cost functions of two plans P1 and P2 in a bounding box *B* = [75MB, 300MB] for |σ(R)|. Plan P1 is a HHJ that builds on σ(R) and probes with S, and Plan P2 is a HHJ that builds on S and probes with σ(R). As shown in Figure 3, P1 is optimal for the estimated |σ(R)| = 150MB < |S| = 160MB. But, P1 is not optimal at all points in the bounding box. Plan P2 is not optimal for the estimated size of σ(R). However, P2's cost is very close to optimal at all points in *B*, since |S| = 160MB < Memory, so P2 will always finish in one pass over R and S. On the other hand, the hash table on σ(R) in Plan P1 will spill to disk when |σ(R)| > Memory = 200 MB, as indicated by the cost function for P1 in Figure 3. Therefore, Plan P2 is a robust plan in *B*.

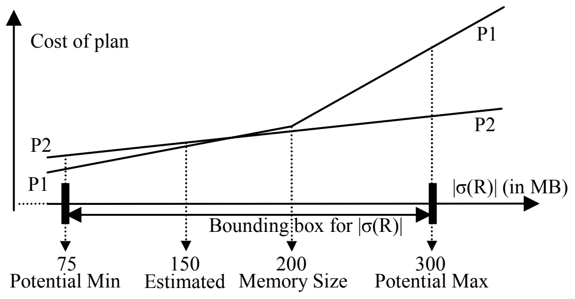


Figure 3 – Robust and switchable plans

Picking P2 would be a safe option. However, P1 and P2, which are HHJs with their builds and probes reversed, are switchable plans in Rio [1]. Thus, it is better to pick the switchable plan *S* = {P1, P2} instead of the robust plan P2: *S* is guaranteed to run the optimal plan as long as |σ(R)| lies within *B*. ∎

2.3 Re-Optimization-Aware Query Execution

2.3.1 Buffers and Switch Operators

A switchable plan chosen by Rio during optimization is executed at run-time using a combination of a *switch operator* and a *buffer operator* [1]. Figure 4 shows these operators for the switchable plan *S*={P1, P2} from Example 2. The buffer operator is placed above the common subplan for the uncertain input to the join, σ(R), and the switch operator is placed above the buffer operator. During query execution, the buffer operator buffers tuples from its input subplan until it gets an *end-of-sample punctuation eos(f)*. (Generation of such punctuations is described in Section 2.3.2.) Punctuation *eos(f)* signals that the set of tuples buffered so far is an *f* % random sample of the complete input. Therefore, if *n* is the number of buffered records, then 100*n/f* is a fairly accurate estimate of the input cardinality. The switch operator uses this estimate to pick the appropriate member plan *P* from {P1, P2}, and execution continues with *P*.

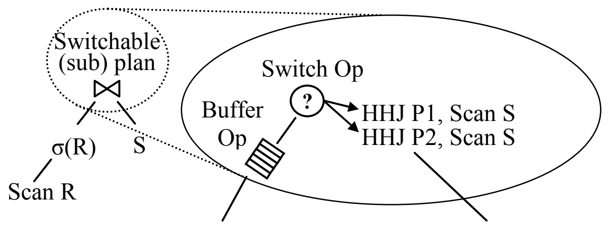


Figure 4 – Switch and buffer operators

2.3.2 Run-time Random-Sample Processing

To generate *eos(f)* punctuations required by buffer operators, we altered the regular processing of some of Predator's operators so that, with minimal overhead, they can prefix their output with a random sample of their entire output. Each such operator *O* first outputs an *f* % random sample of its entire output. (*f* is a user-defined parameter.) Next, *O* generates an end-of-sample punctuation *eos*(*f*) to signal the end of the sample. Finally, *O* sends its remaining output tuples. As shown in Figure 5, tuples output as part of the random sample are not generated again.

Rio supports a variety of techniques to generate and process random samples as part of query execution, without adding any significant run-time overhead. Details are given in [1]. As one example, Rio may store the tuples in a table R in random order on disk so that a sequential scan over R will produce these tuples in random order.

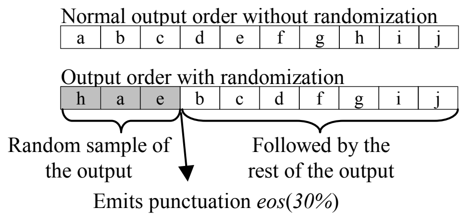

Normal output order without randomization
a b c d e f g h i j
Output order with randomization
h a e b c d f g i j
Random sample of the output
Followed by the rest of the output
Emits punctuation *eos*(*30%*)

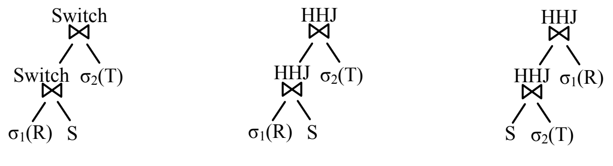
Figure 5 - Operator output with a random-sample prefix

3. DEMONSTRATION OF RIO

We propose to demonstrate each of Rio's features described in Section 2 using queries and data drawn from a real-life OLAP workload. We have developed a graphical interface for purposes of demonstration by extending an existing visualizer for adaptive processing [2]. In the rest of this paper, we describe a scenario considered in [1] that we will use as part of our demonstration of Rio.

Consider the query $\sigma1(R) \bowtie S \bowtie \sigma2(T)$ over relations R, S, and T. Suppose there are no histograms on R, so a default estimate of the selectivity of $\sigma1$ is used to estimate $|\sigma1(R)|$. However, $|\sigma2(T)|$ is estimated accurately from a histogram on T. The optimal plan for this query for low values of $|\sigma1(R)|$ is Plan P6b shown in Figure 6. For higher values of $|\sigma1(R)|$, Plan P6c in Figure 6 is optimal. Furthermore, Plan P6b is the optimal plan for the single-point estimates of all input sizes relevant to $\sigma1(R) \bowtie S \bowtie \sigma2(T)$. Hence, a traditional optimizer (TRAD) will pick Plan P6b.

Rio starts with Plan P6a in Figure 6 which contains two switch operators. The first switch operator has two member plans: (i) HHJ with $\sigma1(R)$ as build and S as probe, and (ii) HHJ with S as build and $\sigma1(R)$ as probe. This switch operator will choose between these two member plans based on the run-time estimate of $|\sigma1(R)|$, obtained from processing a 1% random sample of R. The two member plans in the second switch operator are (i) HHJ with $\sigma1(R) \bowtie S$ as build and $\sigma2(T)$ as probe, and (ii) HHJ with $\sigma2(T)$ as build and $\sigma1(R) \bowtie S$ as probe. The choice between these two plans will be made based on an estimate of $|\sigma1(R) \bowtie S|$ obtained from processing a 1% random sample of $\sigma1(R) \bowtie S$.



Plan P6a **Plan P6b** **Plan P6c**

Figure 6 – Plans for $\sigma1(R) \bowtie S \bowtie \sigma2(T)$

Figure 7 shows the performance of TRAD, Rio, and a reactive re-optimizer RRO as we vary the error in estimating $|\sigma1(R)|$ by varying the selectivity of $\sigma1$.

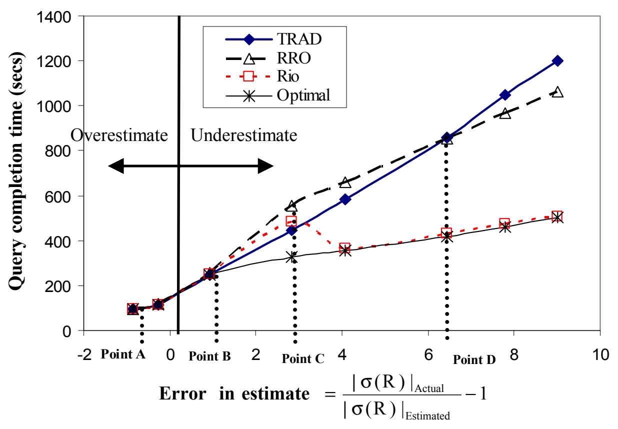


Figure 7 – Performance of different optimizers

Figure 7 also shows the performance of the optimal plan at all points, which we determined manually in each case. As part of our demonstration, we plan to reproduce this setting using a real-life workload and illustrate how:

1. Switchable plans and switch operators enable Rio to find and execute the optimal plan in the presence of uncertain statistics (e.g., at Point B in Figure 7).
2. Quick, efficient, and accurate collection of statistics during query execution enables Rio to converge quickly to efficient plans even in the presence of highly uncertain statistics (e.g., point D in Figure 7).
3. A reactive re-optimizer can get stuck in highly suboptimal plans, and the techniques Rio uses to address this problem (e.g., point D in Figure 7).
4. Rio's performance is affected when input statistics lie in regions where one plan ceases to be optimal with respect to another plan (e.g., point C in Figure 7).

4. REFERENCES

[1] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, June 2005.

[2] S. Babu and J. Widom. StreaMon: An Adaptive Engine for Stream Query Processing (Demonstration proposal). In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 931-932, June 2004.

[3] N. Kabra and D. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data,* pages 106-117, June 1998.

[4] V. Markl et al. Robust Query Processing through Progressive Optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659-670, June 2004.

[5] P. Seshadri. Predator: A Resource for Database Research. *SIGMOD Record* 27(1): 16-20, 1998.