Proactive Re-Optimization

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 to choose good execution plans. This design often leads to suboptimal plan choices for complex queries, since errors in estimates for intermediate subexpressions grow exponentially in the presence of skewed and correlated data distributions. Re-optimization is a promising technique to cope with such mistakes. Current re-optimizers first use a traditional optimizer to pick a plan, and then react to estimation errors and resulting suboptimalities detected in the plan during execution. The effectiveness of this approach is limited because traditional optimizers choose plans unaware of issues affecting re-optimization. We address this problem using *proactive re-optimization*, a new approach that incorporates three techniques: i) the uncertainty in estimates of statistics is computed in the form of bounding boxes around these estimates, ii) these bounding boxes are used to pick plans that are robust to deviations of actual values from their estimates, and iii) accurate measurements of statistics are collected quickly and efficiently during query execution. We present an extensive evaluation of these techniques using a prototype proactive re-optimizer named *Rio*. In our experiments Rio outperforms current re-optimizers by up to a factor of three.

1. INTRODUCTION

Most query optimizers use a *plan-first execute-next* approach–the optimizer enumerates plans, computes the cost of each plan, and picks the plan with lowest cost [23]. This approach relies heavily on the accuracy of estimated statistics of intermediate subexpressions to choose good plans. It is a well-known problem that errors in estimation propagate exponentially in the presence of skewed and correlated data distributions [8, 14]. Such errors, and the consequent suboptimal plan choices, were not a critical problem when databases were smaller, queries had few joins and simple predicates, and hardware resources were limited. In the last two decades, data sizes, query complexity, and the hardware resources to manage databases have grown dramatically. Query optimizers have not kept pace with the ability of database systems to execute complex queries over very large data sets.

Several techniques have been proposed to improve traditional query optimization. These techniques include better statistics [22], new algorithms for optimization [9, 13, 15], and adaptive architectures for execution [2]. A very promising technique in this direction is *re-optimization*, where the optimization and the execution stages of processing a query are interleaved, possibly multiple times, over the running time of the query [17, 18, 20, 26]. Reference [20] shows that re-optimization can improve the performance of complex queries by an order of magnitude.

Current re-optimizers take a *reactive* approach to re-optimization: they first use a traditional optimizer to generate a plan, and then track statistics and respond to estimation errors and resulting suboptimalities detected in the plan during execution. Reactive re-optimization is limited by its use of an optimizer that does not incorporate issues affecting re-optimization, and suffers from at least three shortcomings:

- The optimizer may pick plans whose performance depends heavily on uncertain statistics, making re-optimization very likely.
- The partial work done in a pipelined plan is lost when re-optimization is triggered and the plan is changed.
- The ability 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.

In this paper we propose *proactive re-optimization* to address these shortcomings. We have implemented a prototype proactive re-optimizer called *Rio* that incorporates three new techniques:

- *Bounding boxes* are computed around estimates of statistics to represent the uncertainty in these estimates.
- The bounding boxes are used during optimization to generate *robust* and *switchable* plans that minimize the need for re-optimization and the loss of pipelined work.
- Random-sample processing is merged with regular query execution to collect statistics quickly, accurately, and efficiently at run-time.

Our experimental results demonstrate that proactive re-optimization can provide up to three times improvement over a strictly reactive re-optimizer. The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 uses a series of examples to illustrate the problems with reactive re-optimization, and Section 4 shows how proactive re-optimization addresses these problems. Section 5 describes the Rio implementation and Section 6 presents an experimental evaluation. We outline future work in Section 7.

2. RELATED WORK

Reference [4] classifies adaptive query processing systems into three families: plan-based, routing-based, and continuous-query-based. In this paper we focus on plan-based systems, the more closely related to Rio being ReOpt [18] and POP [20]. Other related projects include Ginga [21], Tukwila [16], query scrambling [26], and corrective query processing [17]. ReOpt and POP use a traditional

[†] Supported by NSF grants IIS-0118173 and IIS-0324431.
[‡] Supported by NSF grant IIS-0086002.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

optimizer to pick plans based on single-point estimates of statistics. These reactive re-optimizers augment the chosen plan with *checks* that are verified at run-time. The query is re-optimized if a check is violated.

The use of intervals instead of single-point estimates for statistics has been considered by least-expected-cost optimization (LEC) [9], error-aware optimization (EAO) [27], and parametric optimization [13, 15, 21]. LEC treats statistics estimates as random variables to compute the expected cost of each plan. Unlike LEC, Rio does not assume knowledge about the underlying distribution of statistics. Instead, Rio computes the uncertainty in these estimates based on how they were derived. Like Rio, EAO considers intervals of estimates and proposes heuristics to identify robust plans. However, the techniques in EAO assume a single uncertain statistic (memory size) and a single join. Furthermore, LEC and EAO do not consider re-optimization or the collection of statistics during query execution. Therefore, these techniques use execution plans that were picked before the uncertainty in statistics is resolved. Parametric optimization identifies several execution plans during optimization, each of which is optimal for some range of values of run-time parameters. Parametric optimization, along with the *choose-plan operator* [11], enables the optimizer to defer the choice of plan to run-time. Switchable plans and switch operators in Rio are similar. However, unlike choose-plan operators, switch operators may occur within pipelines. Furthermore, parametric optimization does not consider uncertainty in estimates, collection of statistics during execution, robust plans, or re-optimization.

Rio combines the processing of random samples of tuples with regular query processing to obtain quick and accurate estimates of statistics during execution. This approach differs from previous uses of random samples, e.g., providing continuously-refined answers in an online manner [12], computing approximate query results [1, 6], or building base relation statistics from samples [8]. Robust cardinality estimation (RCE) uses random samples for cardinality estimation, to deal with uncertainty, and to explore performance-predictability tradeoffs [3]. However, RCE does not consider re-optimization. Furthermore, RCE does not consider techniques such as merging random-sample processing with regular query execution, or propagating random samples through joins.

3. PROBLEMS WITH REACTIVE RE-OPTIMIZATION

In this section we present a series of examples to highlight the problems with current approaches to query re-optimization. One known problem with traditional optimizers, e.g. [23], is that they rely frequently on outdated statistics or invalid assumptions such as independence among attributes. Consequently, they may choose suboptimal query plans that degrade performance by orders of magnitude [8, 20]. Example 1 illustrates this problem.

Example 1: Consider the query "select * from R, S where R.a=S.a and R.b>K1 and R.c>K2". Assume the database buffer-cache size is 200MB, |R|=500MB, |S|=160MB, and |σ(R)|=300MB, where σ(R) represents the result of the "R.b>K1 and R.c>K2" selection on R. However, because of skew and correlations in the data distributions of R.b and R.c, the optimizer underestimates |σ(R)| to be 150MB. With this incorrect estimate, the optimizer would pick Plan P1a for this query (Figure 1). P1a is a hash join with σ(R) as the build input and S as the probe. (Throughout this paper we use the convention that the left input of a hash join is the build and the right input is the probe.) However, since |σ(R)| is actually 300MB, Plan P1a's hash join requires two passes over R and S. P1a is suboptimal because Plan P1b, which builds on S, finishes in one pass over R and S.

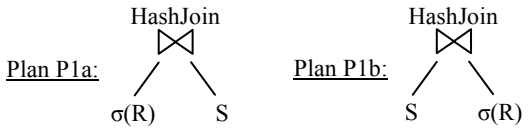


Figure 1 – Two plans for the σ(R) ⋈ S query ■

Re-optimization can avoid problems similar to the one in Example 1. Current systems that use re-optimization first use a traditional optimizer to pick the best plan, and then add *check operators* to the chosen plan. The check operators detect sub-optimality during execution, and trigger re-optimization if required. For example, the check-placement algorithm used by POP computes a *validity range* for each plan [20]. Let P be a left-deep plan. The root operator of P is a binary join operator with subtree D and base relation R as inputs. Let |D| denote the result size of D. POP defines the validity range of P as the range of values of |D| for which P has the lowest cost among all plans P', where P' is logically equivalent to P, P' is rooted at an operator with the same inputs D and R, and P' gives the same interesting orders as P.

During execution, each check operator collects statistics on its inputs. If these statistics satisfy the validity ranges for the plan picked by the optimizer, then execution proceeds as usual. Otherwise, re-optimization is invoked to choose the best plan based on the statistics collected. The reuse of intermediate results that were materialized completely in a previous execution step is considered during re-optimization. Example 2 illustrates the overall technique.

Example 2: Consider the scenario from Example 1. A re-optimizer like POP will choose the same plan (P1a) as a traditional optimizer. Additionally, POP will compute validity ranges for the chosen plan. For example, a validity range for P1a is 100KB≤|σ(R)|≤160MB. If |σ(R)|<100KB, then it is preferable to use an index nested-loops join with tuples in σ(R) probing a covering index on S. If |σ(R)|>160MB, then Plan P1b is optimal. In this example, the check |σ(R)|≤160MB will fail during execution, invoking re-optimization. ■

3.1 Limitations of Single-point Estimates

Although re-optimization preempts the execution of the suboptimal Plan P1a in Example 1 when |σ(R)|>160MB, it incurs the overhead of calling the optimizer more than once and the cost of repeating work. For example, the (partial) scan of R in Plan P1a until re-optimization is lost and must be repeated in P1b. The optimizer may be better off picking Plan P1b from the start because P1b is a *robust plan* with respect to the uncertainty in |σ(R)|; see Figure 2.

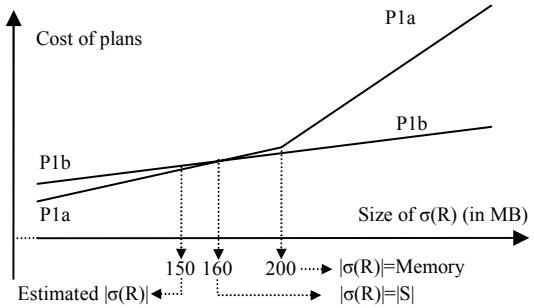


Figure 2 – Cost of plans P1a and P1b as |σ(R)| varies

When |σ(R)|≤Memory, both plans finish in one pass and involve the same amount of IO. However, when |σ(R)|>Memory, only P1b finishes in one pass.

Current re-optimizers do not account for robustness of plans since they consider *single-point estimates* for all statistics needed to cost plans. (To arrive at these single-point estimates, optimizers are often forced to make assumptions like uniformity and independence [23].) Non-robust plans may lead to extra optimizer invocations and wasted work, as we will show in Section 3.3.

3.2 Limited Information for Re-Optimization

Current re-optimizers make limited effort to collect statistics quickly and accurately during execution. For instance, the validity check in Example 2 will fail when |σ(R)|=160MB, and re-optimization will be invoked. However, the optimizer does not know |σ(R)| accurately at this point–it only knows that |σ(R)|≥160MB–which may cause it to chose a suboptimal plan again. Example 3 illustrates an extreme instance of the thrashing that can result.

Example 3: Consider the query "select * from R, S, T where R.a=S.a and S.b=T.b and R.c>K1 and R.d=K2". Assume that the sizes of the tables are known accurately to be |R|=200MB, |S|=50MB, and |T|=60MB. Further assume that |σ(R)|=80MB, but that the optimizer underestimates it significantly as 40KB.[1] Based on these statistics, the optimizer chooses Plan P3a.

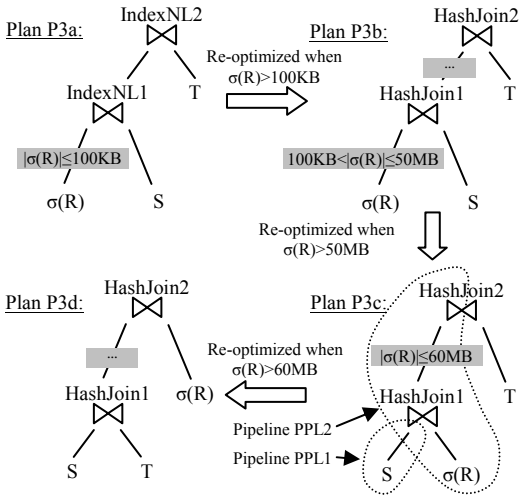


Figure 3 – Thrashing with reactive re-optimization

A reactive re-optimizer may compute validity ranges for Plan P3a as shown by the gray boxes in this plan. For example, the validity range for the index nested-loops join between σ(R) and S in P3a is |σ(R)|≤100KB. This validity-range check will fail at run-time, triggering re-optimization. Plan P3b will be picked next with a validity range as shown in Figure 3. This check will fail and re-optimization will be triggered again, and so on until the optimal Plan P3d is chosen finally. ■

3.3 Losing Partial Work in a Pipeline

In addition to the multiple re-optimization steps as illustrated in Example 3, current re-optimizers also lose the partial work done by a pipeline in execution when re-optimization is triggered. For example, Plan P3c in Figure 3 has a pipeline PPL2 (enclosed with dotted lines) that scans R, probes S in HashJoin1, and builds joining tuples into HashJoin2. The validity-range check before HashJoin2 will fail before pipeline PPL2 finishes, and the partial work done by this pipeline will be lost. On the other hand, work done by completed pipelines, like PPL1–scanning and building S–can be reused. However, in this example, the build of S in Plan P3c cannot be reused in Plan P3d because the hash tables are built on different join attributes.

4. PROACTIVE RE-OPTIMIZATION

This paper proposes *proactive re-optimization*, a new paradigm for query re-optimization. Proactive re-optimization addresses the problems with current reactive approaches that were illustrated in Section 3. A proactive re-optimizer incorporates three new techniques:

1. Computing *bounding boxes*–intervals around estimates–as a representation of the uncertainty in estimates of statistics.
2. Using bounding boxes during optimization to generate *robust plans* and *switchable plans* that avoid re-optimization and loss of pipelined work.
3. Using randomization to collect statistics quickly, accurately, and efficiently as part of query execution.

Figure 4 shows the architecture of a proactive re-optimizer. In Section 5 we introduce Rio, our specific implementation of a proactive re-optimizer.

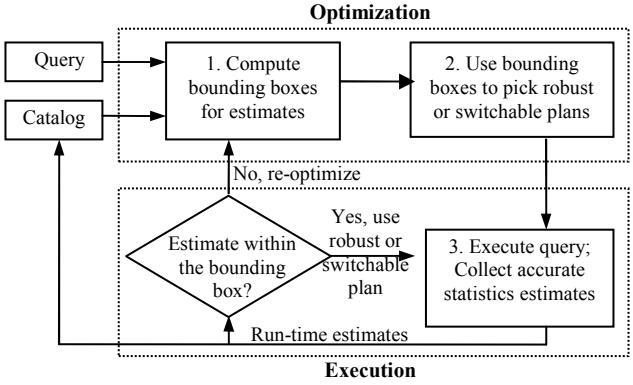


Figure 4 – Proactive re-optimization

4.1 Representing Uncertainty in Statistics

Current re-optimizers compute a single-point estimate for any statistic needed to cost plans. One way to account for possible errors in estimates is to consider intervals, or bounding boxes, around the estimates. If the optimizer is very certain of the quality of an estimate, then its bounding box should be narrow. If the optimizer is uncertain of the estimate's quality, then the bounding box should be wider. There are different ways of computing bounding boxes, e.g., using strict upper and lower bounds [7] or by characterizing uncertainty in estimates using discrete buckets that depend on the way the estimate was derived [18]. Our implementation uses the latter approach as described in Section 5.2.

Example 4: Consider the scenario from Example 1. The costs of plans P1a and P1b depend mainly on |σ(R)| and |S|. Suppose a recent estimate of |S|=160MB is available in the catalog. However, in the absence of a multidimensional histogram on R, |σ(R)| must be estimated from the estimated selectivities of

[1] A recent paper from IBM reports cardinality estimation errors on real datasets that exceed six orders of magnitude [20].

R.b>K1 and R.c>K2 and an assumption of independence between these predicates. This estimate of |σ(R)|=150MB is thus very uncertain. In this case, Figure 5 shows an example bounding box around the single-point estimate (|σ(R)|=150MB, |S|=160MB).

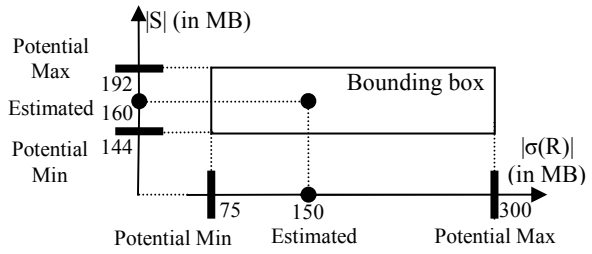


Figure 5 – Bounding box around estimates of |σ(R)| and |S| ■

4.2 Using Bounding Boxes During Optimization

Since current re-optimizers consider single-point estimates only, their plan choices may lead to extra re-optimization steps and to the loss of partial pipelined work if actual statistics differ from their estimates. Bounding boxes can be used during optimization to address this problem. While there is always one plan that is optimal for a single-point estimate, one of the following four cases can occur with a bounding box B:

(C.i) *Single optimal plan*. A single plan is optimal at all points within B.

(C.ii) *Single robust plan*. There is a single plan whose cost is very close to optimal at all points within B.

(C.iii) *A switchable plan*. Intuitively, a switchable plan in *B* is a set *S* of plans with the following properties: a) At 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 deferred until accurate estimates of uncertain statistics are available at query execution time; and c) If the actual statistics lie within *B*, an appropriate plan from *S* can be picked and run without losing any significant fraction of the execution work done so far.

(C.iv) *None of the above*. Different plans are optimal at different points in B, but no switchable plan is available.

A proactive re-optimizer identifies which of the above four cases *B* falls into. Note that a single optimal plan is also robust, and a robust plan is a singleton switchable plan.

Example 5 illustrates how a proactive re-optimizer can exploit robust plans and switchable plans. Details of how to enumerate and choose robust and switchable plans are given in Section 0.

Example 5: Consider the scenario from Example 1. Figure 6 is the same as Figure 2 except that it considers the bounding box B=[75MB, 300MB] for |σ(R)|.

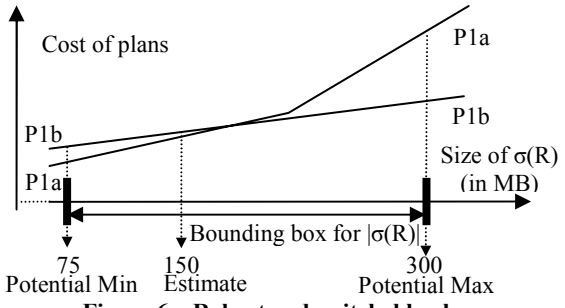


Figure 6 – Robust and switchable plans

As seen, Plan P1a is optimal for the estimated |σ(R)|=150MB, but not in the entire bounding box. While Plan P1b is not optimal for the estimated |σ(R)|, P1b is robust because its cost is very close to optimal at all points in B. Therefore, picking Plan P1b would be a safe option. However, as we will see in Section 0, P1a and P1b (which are hybrid hash joins with build and probe reversed) are switchable. It is preferable to pick the switchable plan P={P1a, P1b} instead of the robust P1b because P is guaranteed to run the optimal plan as long as |σ(R)| lies within B. ■

4.3 Accurate Run-Time Statistics Collection

As seen in Example 3, the lack of accurate run-time statistics collection can lead to thrashing during re-optimization. In general, accurate run-time estimates are needed to pick the right plan from a switchable set, to detect when to trigger re-optimization, and to pick a better plan in the next optimization step.

For efficiency, we hide the cost of collecting accurate statistics by combining statistics collection with regular query execution. Furthermore, for early detection of the need to re-optimize, the run-time estimates must be computed both quickly and accurately. We achieve these goals by using a new technique of merging the processing of random samples of tuples along with regular query execution. Example 6 illustrates this approach. Implementation details are given in Section 5.4.

Example 6: Consider Example 3. Assume that the optimizer had picked the suboptimal Plan P3a which contains a pair of index nested-loops joins with σ(R) as the outer input. Suppose tuples in R are physically laid out in random order on disk. Then, once 5% of the R tuples have been scanned and processed, a fairly accurate estimate of the selectivity of σ is available. Thus, |σ(R)| can be estimated reliably. This estimate enables a proactive re-optimizer to detect quickly that P3d is the optimal plan, thereby avoiding the thrashing problem in reactive re-optimizers. ■

5. PROACTIVE RE-OPTIMIZATION WITH RIO

Section 4 presented an overview of proactive re-optimization without providing specifics about the implementation. We now describe our prototype proactive re-optimizer Rio.

5.1 Building Rio

Rio was built using the Predator DBMS [24] by extending it as follows:

- Equi-height and end-biased histograms were added [22].
- Predator has a traditional cost-based dynamic-programming optimizer [23] which we refer to as *TRAD*. We added:
  - A *Validity-Ranges Optimizer* (*VRO)*, our implementation of the algorithms used by POP [20].
  - *Rio*, our proactive re-optimizer.
  - Uncertainty buckets and rules from [18] to generate and propagate uncertainty buckets during query optimization.
- The following operators were added:
  - A hybrid hash join operator [19] that processes tuples from two input subtrees. At most one of the subtrees is a *deep subtree* and at least one is a subtree with one *base relation*. Either subtree can be the build input of the hash join. Thus, this operator enables us to consider arbitrary linear plan shapes, e.g., right-deep join trees like Plan P10c in Figure 10. Recall our convention that the left

input to the hash join is the build and the right input is the probe.
    - A switch operator to implement switchable plans.
    - Operators to read random samples from base relations and to generate random samples of joins as part of query execution.
    - Buffer operators to buffer tuples and delay processing in a pipeline until the statistics necessary to choose among the set of plans in a switch operator have been collected.
    - Operators to scan previously materialized expressions for reuse after re-optimization. Materialized expressions that may be reused include completed builds of hash joins and the sorted temporary files created by a sort operator.
    - The original validity-ranges algorithm [20] uses checks on buffers to trigger re-optimization when the buffers overflow or underflow. In our VRO implementation, validity ranges are checked by buffer operators placed appropriately in the plan which buffer and count incoming tuples. The buffer operators trigger re-optimization if any validity range is violated.
- *Execution engine:*
    - The ability to stop query execution midway, re-optimize, and restart execution.
    - An in-memory catalog to track statistics collected at run-time as well as expressions materialized as part of query execution. The optimizer consults this catalog during re-optimization.
    - An inter-operator communication mechanism based on *punctuations* [25] that, e.g., allows an operator C to signal to its parent operator that C has generated a 1% random sample of its output.

5.2 Computing Bounding Boxes

Recall that a proactive re-optimizer uses bounding boxes instead of single-point estimates for statistics needed to cost plans. Currently, Rio restricts the computation of bounding boxes to size and selectivity estimates. For each such estimate *E*, a bounding box *B* is computed using a two-step process:

- An uncertainty bucket *U* is assigned to the estimate *E*
- The bounding box is computed from the (*E*, *U*) pair

To compute *U*, we adopted a technique from [18] that uses a set of rules to compute uncertainty. (We plan to try other techniques in the future, e.g., stochastic intervals as in [3].) In the original approach [18], the value of *U* belongs to a three-valued domain {*small*, *medium*, *large*} that characterizes the uncertainty in the estimate E. The value of *U* is computed based on the way *E* is derived. For example, if an accurate value of *E* is available in the catalog, then *U* takes the value *small* that denotes low uncertainty. In Rio, we augmented the domain of *U* to an integer domain with values from 0 (no uncertainty) to 6 (very high uncertainty).

A bounding box *B* of an estimated value *E* is an interval [*lo*, *hi*] that contains *E*. The uncertainty value *U* is used to compute the values *lo* and *hi* as shown in Figure 7. Example 7 illustrates the computation of uncertainty buckets and bounding boxes for our running example.

Example 7: Consider the scenario from Example 1. The optimizer needs to cost plans P1a and P1b which depend on $|\sigma(R)|$ and $|S|$. Recall that σ represents R.b>K1 and R.c>K2. The single-point estimates for $|S|$ and $|\sigma(R)|$ are E_S=160MB and E_R=150MB respectively. Assume that E_S was obtained from the catalog. Therefore, our rules adapted from [18] for derivation of uncertainty set U_S=1 (low uncertainty in E_S). From Figure 7, the bounding box for E_S is B_S=[144, 192]. On the other hand, assume that the estimate E_R was computed from the estimated selectivities of R.b>K1 and R.c>K2 based on the assumption that these predicates are independent (no multidimensional histogram was available). Thus, the uncertainty in E_R is high. Accordingly, our rules for derivation of uncertainty set U_R=5. From Figure 7, the bounding box for E_R is B_R=[75, 300]. ■

```
ComputeBoundingBox(Inputs: estimate E, uncertainty U
                   Outputs: lo, hi) {
   Δ+ = 0.2; // increment step
   Δ- = 0.1; // decrement step
   hi = E * (1 + Δ+ * U);
   lo = E * (1 - Δ- * U);
}
```

Figure 7 – Computing bounding boxes for an (E, U) pair

5.3 Optimizing with Bounding Boxes

The TRAD optimizer enumerates and groups plans based on their join subset (JS) and interesting orders (IO) [23]. For each distinct (JS, IO) pair enumerated, TRAD prunes away all plans except the plan with the lowest cost, denoted *BestPlan*. The cost of each plan is computed based on estimated statistics.

VRO takes the same steps as TRAD initially, so VRO will find the same optimal plan (*BestPlan*) for each (JS, IO) pair. However, VRO then adds validity ranges on the inputs to the join operators in *BestPlan* [20]. Consider a join operator *O* with inputs R_D and R_B, where R_D is the deep subtree input and R_B is the base relation input. The validity range of *O* is the range of values of $|R_D|$ where operator *O* has the lowest cost among all join operators with the same inputs R_D and R_B, and giving the same set of interesting orders as *O*. The validity range of *O* is computed by varying $|R_D|$ up (and down) until the cost of *O* is higher than that of some other join operator with the same inputs R_D and R_B and giving the same set of interesting orders as *O*. The *Newton-Raphson method* can be applied to the join cost-functions to compute validity ranges more efficiently than linear search; see [20].

Unlike TRAD and VRO, Rio computes bounding boxes for all input sizes used to cost plans. Then it tries to compute a switchable plan (which may also be a single robust plan or a single optimal plan) for each distinct (JS, IO) pair based on the bounding boxes on the inputs to the plan. If Rio fails to find a switchable plan for a (JS, IO) pair, then it picks the optimal plan for (JS, IO) based on the single-point estimates of input sizes (*BestPlan*), and adds validity ranges like VRO.

Rio computes switchable plans in two steps. First, it finds three seed plans for each (JS, IO) pair. Then, it creates the switchable plan from the seed plans as described next.

5.3.1 Generating the Seed Plans

In traditional enumeration, plan cost is computed using single-point estimates of statistics. In Rio, the enumeration considers three different costs for each plan, C_{Low}, C_{Est}, and C_{High}. Cost C_{Est} is computed using the single-point estimate of statistics exactly like in traditional enumeration. Cost C_{Low} (C_{High}) is computed at

the lower left corner (upper right corner) of a bounding box as illustrated in Figure 8.

Rio augments the (JS, IO) pair used during traditional enumeration with an extra *cost bucket* CB that takes values *Low*, *Estimated*, or *High*. Like the interesting order concept, the cost bucket defines which plans and costs are comparable during cost-based pruning, e.g., a Plan P for (JS, IO, CB=*Low*) is pruned if and only if there exists a Plan P' for (JS, IO, CB=*Low*) with a lower cost C_{Low} than P. For each distinct (JS, IO) pair, Rio enumerates and prunes plans for the three triples (JS, IO, CB=*Low*), (JS, IO, CB=*Estimated*), and (JS, IO, CB=*High*). The plans that remain after pruning are the three plans corresponding to the minimum C_{Low}, C_{Est}, and C_{High} for (JS, IO).

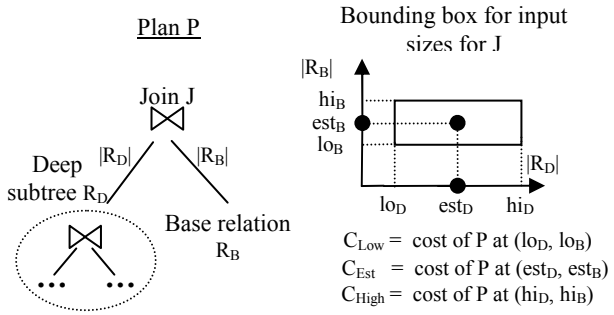


Figure 8 – Computing plan costs

Note that the best plan for (JS, IO, CB=*Estimated*) is the same plan (*BestPlan*) as computed by TRAD for (JS, IO). Also, the addition of the extra cost bucket guarantees that the optimal plan for the estimated statistics will not prune away plans that are optimal at the upper right or lower left corners of the bounding boxes for input sizes. For each (JS, IO) pair, we end up with three seed plans from which a switchable plan will be created:

- *BestPlanLow*, the plan with minimum cost C_{Low}
- *BestPlanEst*, the plan with minimum cost C_{Est}
- *BestPlanHigh*, the plan with minimum cost C_{High}

5.3.2 Generating the Switchable Plan

Given the seeds *BestPlanLow*, *BestPlanEst*, and *BestPlanHigh*, one of four cases arises:

(C.i) The seeds are all the same plan.

(C.ii) The seeds are not all the same plan, but one of them is a robust plan.

(C.iii) The seeds are not all the same plan, and none of them is robust, but a switchable plan can be created from the seeds.

(C.iv) We cannot find a single optimal plan, a single robust plan, or a switchable plan from the seeds.

In Case (C.i), the single optimal plan is the switchable plan. (Recall that an optimal plan is also robust and a robust plan is a singleton switchable plan.) In Case (C.ii), the optimizer checks if any of the seeds is a robust plan. A necessary test to determine whether *BestPlanLow* is robust is to check whether (i) cost C_{Est} of *BestPlanLow* is close to (e.g., within 20% of) C_{Est} of *BestPlanEst*, and (ii) cost C_{High} of *BestPlanLow* is close to C_{High} of *BestPlanHigh*. Intuitively, we are testing whether *BestPlanLow* has performance close to optimal at the estimated point and at the upper corner of the bounding box as well. While this test is not sufficient to guarantee robustness–because we do not check all points in the bounding box–Rio currently labels a plan as robust if it passes this *plan-robustness test*. If one of the seeds passes this test, then Rio uses that seed as a singleton switchable plan.

Example 8: Consider the scenario from Example 1. As seen in Figure 9, *BestPlanLow* = *BestPlanEst* = P1a and *BestPlanHigh* = P1b. The cost of P1a is not within 20% of the cost of P1b at the upper corner of the bounding box (|σ(R)|=300MB). Thus, P1a is not a robust plan within the bounding box. On the other hand, P1b is within 20% of the cost of P1a both at the estimated point (|σ(R)|=150MB) and at the lower corner of the bounding box (|σ(R)|=75MB). Therefore, P1b passes the plan-robustness test.

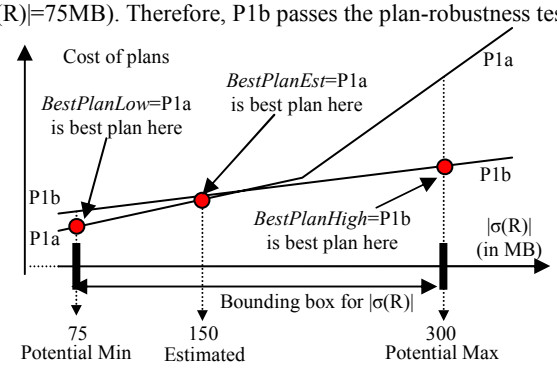


Figure 9 – Finding a robust plan in |σ(R)|'s bounding box ■

If none of the seeds is a single optimal plan or a single robust plan (Case (C.iii)), then the optimizer tries to find a switchable plan. A switchable plan for a (JS, IO) pair is a set of plans *S* where:

(i) All plans in *S* have a different join operator as the root operator. (Hybrid hash joins with the build and probe reversed are treated as different operators.)

(ii) All plans in *S* have the same subplan for the deep subtree input to the root operator.

(iii) All plans in S have the same base table, but not necessarily the same access path, as the other input to the root operator.

Figure 10 contains an example of a switchable plan with three member plans for (JS={R,S,T}, IO=∅). Any two members of a switchable plan are said to be *switchable with each other*. In Section 5.4 we illustrate how the switchable plan chooses one of its members at execution time.

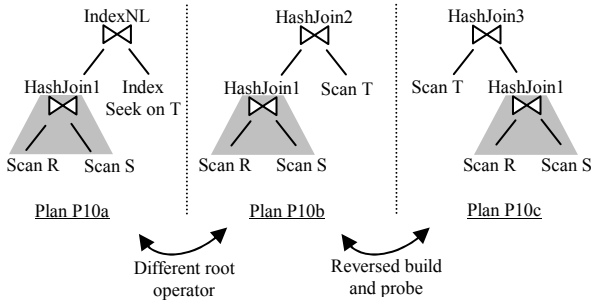
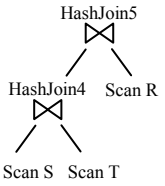


Figure 10 – Possible members of a switchable plan

If the seed plans for a (JS, IO) pair have the same subplan for the deep subtree, then the seeds themselves constitute a switchable plan. If these subplans are different, then Rio picks one of the

seed plans, say *BestPlanLow*, and enumerates the set SW_Low of all plans that are switchable with *BestPlanLow* based on Conditions (i)—(iii) of switchable plans above. Then, among the plans in SW_Low, Rio finds the plan, *planMinEst*, with minimum cost at the estimated statistics point, and the plan, *planMinHigh*, with minimum cost at the upper right corner of the bounding box. If C_{Est} of *planMinEst* is close to (e.g., within 20%) C_{Est} of *BestPlanEst*, and C_{High} of *planMinHigh* is close to C_{High} of *BestPlanHigh*, then {*BestPlanLow*, *planMinEst*, *planMinHigh*} is a switchable plan. If not, Rio tries the same procedure with the two other seed plans.

Example 9: Suppose *BestPlanLow* = Plan P10a, *BestPlanEst* = Plan P10b (Figure 10), and *BestPlanHigh* = Plan P11 (Figure 11) for R ⋈ S ⋈ T with no interesting orders. The subplan for the deep subtree of the outer join is different between P10a and P11, so they are not switchable. Thus, Rio enumerates SW_Low, which contains Plan P10c. If C_{High} of Plan P10c is close to that of P11, then {P10a, P10b, P10c} is a switchable plan. ■



Plan P11

Figure 11

If these techniques fail to find a switchable plan (Case (C.iv)), then Rio picks *BestPlanEst*–the optimal plan for the single-point estimates–and adds validity ranges, just like VRO.

5.4 Extensions to the Query Execution Engine

A switchable plan *S* defers the choice of which member plan to use for a join until the uncertain input sizes can be estimated accurately at run-time. *S* ensures that no (partial) work done by the pipeline containing the join is lost whenever the actual input sizes lie within the corresponding bounding box. Our implementation of switchable plans uses the following operators and communication framework:

- A *switch operator* that corresponds to the chosen switchable plan. This operator decides which member plan to use based on run-time estimates of input sizes, and instantiates the appropriate join operator and base relation access path.
- A *buffer operator* that buffers tuples until it can compute an input-size estimate needed by the switch operator.
- *Randomization-aware operators* that prefix their output with a random sample of their complete output.
- An *inter-operator communication mechanism* based on punctuations [25] that allows operators to send size estimates and to demarcate random samples in their output stream.

5.4.1 Implementing Switchable Plans

For a switchable plan chosen by Rio during optimization, the execution-plan generator creates a switch operator and a buffer operator. Figure 12 shows these two operators generated for the switchable plan in Figure 10. Note that the buffer operator is placed above the common subplan for R ⋈ S (marked in gray in both figures). The switch operator is placed above the buffer operator.

During query execution, the buffer operator buffers tuples from the deep subplan until it gets an *end-of-sample punctuation eos*(*f*). (Generation of such punctuations is described in Section 5.4.2.) Punctuation *eos*(*f*) signals that the set of tuples buffered so far is an *f* % random sample of the output of the deep subplan. Based on the number of buffered tuples *n*, 100*n/f* is a fairly accurate estimate of the final output cardinality of R ⋈ S. The switch operator uses this cardinality estimate to compute the total input size of R ⋈ S, and instantiates the appropriate member plan.

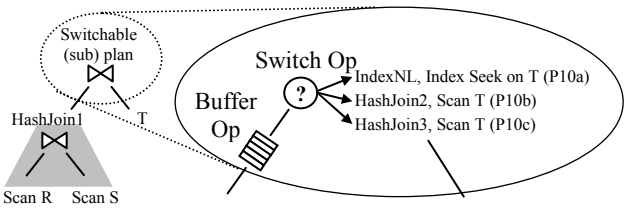


Figure 12–Implementation of switchable plan from Figure 10

Rio currently uses only the size of the deep subtree input R_D to the join to choose the best member plan. In terms of Figure 8, this limitation means that for a switchable plan P={P_{lo}, P_{est}, P_{hi}}, where P_{lo}, P_{est}, and P_{hi} were chosen for (lo_D, lo_B), (est_D, est_B), and (hi_D, hi_B) respectively (recall Section 5.3.2), Rio has to choose among P_{lo}, P_{est}, and P_{hi} based solely on the estimate of $|R_D|$. P_{lo} is picked if $|R_D| \in [lo_D, (lo_D + est_D)/2]$, P_{est} is picked if $|R_D| \in [(lo_D + est_D)/2, (est_D + hi_D)/2]$, and P_{hi} is picked if $|R_D| \in [(est_D + hi_D)/2, hi_D]$. If $|R_D| < lo_D$ or $|R_D| > hi_D$, then the switch operator triggers re-optimization after adding the collected estimate of $|R_D|$ to the catalog.

5.4.2 Random-Sample Processing During Execution

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 13, tuples output as part of the random sample are not generated again.

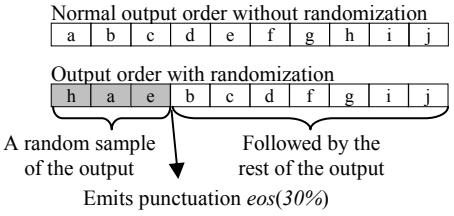


Figure 13 - Random samples in the operator output

Reordering the output of an operator *O* is not an option if any of the operators above *O* in the plan depend on the order of *O*'s output. Thus, random sample generation seems inapplicable to operators such as sorts and ordered scans from B-trees. However, there are ways around this problem. For example, the buffer operator above *O* can regenerate the order using a merge of the initial sample with the later output. Furthermore, blocking operators[2] like sorts provide simpler ways of estimating input sizes without requiring random samples or buffering. We plan to address these issues in detail in future work.

[2] A blocking operator reads all of its input before producing any output.

Next we describe how *eos*(*f*) punctuations are generated by table scans and certain join operators. Note that our techniques never transform a non-blocking operator into a blocking operator.

5.4.2.1 Randomization in Table-Scan Operators

We developed two techniques to enable a scan operator over a table *T* to first return a random sample of tuples from *T*:

(i) If tuples in *T* are laid out in random order on disk, a sequential scan will produce the tuples in the desired order. Whether *T* has a random layout pattern or not can be a physical property of the table, enforced when the table is created and updated. Additionally, such a layout pattern can be detected using the Kiefer-Kolmogorov-Smirnov test when *runstats* is invoked to collect statistics on T; see [5]. This additional statistic can be maintained in the catalog.

(ii) An *f* % random sample of *T*, denoted *T_sample*, can be maintained explicitly as a separate table, e.g., using the techniques from [10]. Each tuple in *T* contains an extra bit to denote whether the tuple is also present in *T_sample* or not. At run-time the table scan first returns tuples from *T_sample*, followed by an *eos*(*f*). Then it scans *T*, returning all tuples not contained in *T_sample*. Note that having tuples duplicated in *T_sample* and *T* allows indexes over *T* to be built and used without any changes. The storage overhead is minimal.

5.4.2.2 Randomization in Join Operators

Adding randomization to the nested-loops join operators–tuple, block, and index–was straightforward. These operators simply pass on the *eos*(*f*) punctuations from their outer input, and ignore *eos*(*f*) from their inner input. A join sample produced in this fashion is a true random sample of the join if the outer table's join column is a foreign key referencing the inner table [1].

To producing a random sample first from a hybrid hash join, we made the following modifications to the standard algorithm:

(i) First, tuples from the probe input are read into memory until an *eos*(*f*) punctuation is received. These tuples represent an *f* % sample of the complete probe input. The join operator inserts these tuples into an in-memory hash table.

(ii) Next, the build input is read and partitioned completely. In addition, as these tuples are being processed, they are immediately joined with the in-memory sample of the probe input. Joining tuples are sent in the join output. At the end of this phase, an *eos*(*f*) punctuation (using the value of *f* received from the probe) is generated, and the in-memory sample is discarded. The tuples output so far correspond to taking an *f* % sample from the probe and joining it with the complete build. This sample is guaranteed to be a true join random sample if the probe input's join column is a foreign key referencing the build input [1].

(iii) The scan of the probe input, which was paused after the *eos*(*f*) in Step (i), is resumed. The tuples are partitioned and joined with the memory-resident build partitions.

(iv) The on-disk partitions are joined to complete the join.

6. EXPERIMENTS

In this section we describe an extensive experimental evaluation of the Rio prototype. We compare Rio with the traditional optimizer (termed TRAD in Section 5.1) and with the Validity-Ranges re-optimizer (termed VRO in Section 5.1) under a variety of conditions. In our experiments we used a synthetic data generator provided by IBM. The generated dataset has four tables whose properties are shown in Table 1.

Table 1 – Summary of dataset used in the experiments

Table	Size, # of Tuples	Sample Correlated Attrs
Accidents (A)	420 MB, 4.2 M	accident_with & damage, seat_belt_on & driver_status
Cars (C)	120 MB, 1.7 M	make & model & color
Owner (O)	228 MB, 1.5 M	city & state & country
Demographics (D)	60 MB, 1.5 M	age & salary & assets

All experiments were done on a 1.7 GHz Pentium machine with 2 MB L2 cache, 512 MB memory, and a single 5400 rpm disk. The buffer cache size is 128 MB. Each hybrid hash join operator is allocated a fixed amount of memory which we vary in some of the experiments; the default value is 50 MB. Buffer operators in Rio and VRO are allocated the same amount of memory as a hybrid hash join. The buffers spill to disk when they fill up. B-tree indexes were available on all primary-key attributes. Equi-height and end-biased histograms were available on all integer attributes. The bounding box computation in Rio happens as described in Figure 7 with Δ_+=0.6 and Δ_-=0.1. The cost threshold for robustness tests is 20% (Section 5.3.2). The random-sample percentage for size estimation is 1% (Section 5.4.2).

6.1 Two-way Join Queries

Our first experiment studies the performance of TRAD, VRO, and Rio with respect to the error in estimates. We use a query joining Accidents (A) with Cars (C) on the car_id attribute. (All joins we consider are foreign key to primary key joins.) There is a selection predicate on A, denoted σ(A), of the form A.accident_year > [year], where [year] is a parameter whose value is varied in this experiment. We removed the equi-height histogram on attribute A.accident_year from the catalog to force the optimizer to use the default selectivity estimate of 0.1. Thus, the optimizer always estimates |σ(A)|=42MB. By varying the value of [year], we vary the error between the estimate of |σ(A)| and its actual size.

6.1.1 Using Robust Plans

The memory limit for a hybrid hash join was set to 150MB in this experiment. When |σ(A)| is less than the size of C (120MB), the optimal plan is a hybrid hash join with σ(A) as the build, denoted Plan P_{AC}. When |σ(A)| > 120MB, the optimal plan is a hybrid hash join with C as the build, denoted Plan P_{CA}. (120MB corresponds to around 1.8 in Figure 14.) Although B-tree indexes are available on the join attributes, index-nested-loop joins never outperform hybrid hash joins in our setting.

Figure 14 shows query completion times, including both optimization and execution times, for TRAD, VRO, and Rio as we vary the error in the estimate of |σ(A)|. The error plotted on the x-axis is computed as $|\sigma(A)|_{Actual} / |\sigma(A)|_{Estimate} - 1$. A positive error indicates an underestimate and a negative indicates an overestimate. Figure 14 also shows the performance of the optimal plan which we determined manually in each case.

Since the optimizer's estimate of |σ(A)| is 42MB, TRAD always picks Plan P_{AC} which is optimal at |σ(A)|=42MB. As |σ(A)| is increased (and the estimation error increases), the cost of Plan P_{AC} increases linearly at a small rate until |σ(A)|=150MB.

(|σ(A)|=150MB corresponds to an error around 2.5 in Figure 14.) When |σ(A)|>150MB, the hybrid hash join in Plan P_{AC} starts spilling to disk. Because of this extra IO, the cost of Plan P_{AC} increases at a steep rate when |σ(A)|>150MB, as shown by the plot for TRAD in Figure 14.

VRO always starts with the same plan as TRAD, i.e., Plan P_{AC}. However, VRO adds a validity range to the join and verifies this range before starting the join execution. The upper bound of the validity range for the hybrid hash join in Plan P_{AC} is 120MB: if |σ(A)|>120MB, then Plan P_{CA} performs better. Therefore, as long as |σ(A)|≤120MB, the validity range is not violated and the performance of VRO matches the performance of the optimal plot in Figure 14. When |σ(A)|>120MB, the validity range is violated and VRO is forced to re-optimize. Plan P_{CA} is picked on re-optimization. VRO cannot reuse the work done by the pipeline in execution in Plan P_{AC} when re-optimization was invoked, namely the scan of A and evaluation of σ(A) up to that point. This loss of work results in the region in Figure 14 where VRO performs worse than TRAD. However, as the error increases, the re-optimization pays off quickly because when |σ(A)|>150MB, the join in Plan P_{AC} spills to disk while P_{CA} scans A and C only once.

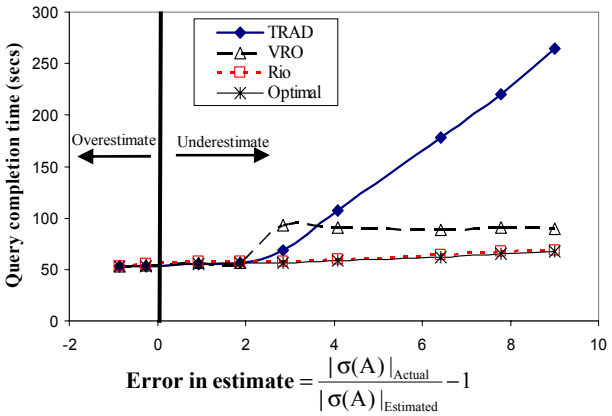


Figure 14 – σ(A) ⋈ C, 150MB per hash join

Rio first computes bounding boxes for |σ(A)| and |C|. Since there are no selection predicates on C, the estimate of |C| available from the catalog is accurate. To illustrate robust plans, in this experiment alone we set Δ_+ and Δ_- in Figure 7 to very high values so that the bounding box on |σ(A)| is [0MB, 420MB]. Rio identifies that Plan P_{CA} is a robust plan within this bounding box. (Rio identifies Plan P_{CA} to be a robust plan even if the bounding box is smaller.) Because the bounding box [0MB, 420MB] covers the entire range considered in the experiment, Rio runs Plan P_{CA} at all points in Figure 14. Although Plan P_{CA} is not optimal at all points in the bounding box, note that Rio's performance is close to the optimal plot at all points in Figure 14, showing the robustness of Plan P_{CA}. Since |C| is less than the memory available to the hash join, P_{CA} always finishes in one scan of A and C.

For our default settings of Δ_+ and Δ_-, the bounding box on |σ(A)| is [16.8MB, 193.2MB]. In this case Rio used a combination of solutions (re-optimization, switchable plans, and robust plans) to provide near-optimal performance. This graph is omitted because Section 6.1.2 shows Rio's performance in a similar situation.

6.1.2 Using Switchable Plans

Our next experiment, reported in Figure 15, considers the same query as in the previous section, but now hash joins are allocated only 50MB of memory for in-memory hash partitions. In this experiment, the behavior of Optimal, TRAD, and VRO regarding the choices of plans and re-optimization points are the same as in the previous section. However, Rio behaves differently. Rio computes the bounding box on |σ(A)| to be [16.8MB, 193.2MB]. The large width of the box corresponds to the high uncertainty in |σ(A)| since this estimate used a default value of selectivity. The bounding box on |C| has zero width since an accurate estimate of |C| is available from the catalog. Rio finds that Plan P_{AC} is optimal at (|σ(A)|,|C|)=(16.8MB, 120MB), which is the lower corner of the bounding box, and also at the estimated point (|σ(A)|,|C|) = (42MB, 120MB). However, for (|σ(A)|,|C|) = (193.2MB, 120MB), which is the upper corner of the bounding box, Plan P_{CA} is optimal. Furthermore, neither P_{AC} nor P_{CA} is robust in this case. However, Rio identifies that plans P_{AC} and P_{CA} are switchable plans (see Section 0). Therefore, for this query, Rio starts with a plan containing a switch operator with the two hybrid hash joins corresponding to P_{AC} and P_{CA} as member plans. Rio estimates |σ(A)| during execution. Based on this estimate, Rio chooses one of the two joins or it re-optimizes.

The accident_year attribute in A is not correlated with the layout of A on disk, so a sequential scan of A produces tuples in random order to estimate the selectivity of σ(A) (recall Section 5.4.2). Rio gets a very accurate estimate of |σ(A)| from the default setting of 1% sampling. For example, when |σ(A)|=6MB in Figure 15, which corresponds to an error of -0.85 and lies outside the bounding box, Rio invokes re-optimization. Since the optimizer now has accurate estimates of |σ(A)| and |C|, it correctly picks Plan P_{AC} which is optimal at this point. Note that Rio's performance is very close to that of the optimal plan for |σ(A)| = 6MB, which shows that the overhead incurred by Rio to sample 1% of A, obtain a run-time estimate of |σ(A)|, and to re-optimize the query is very small.

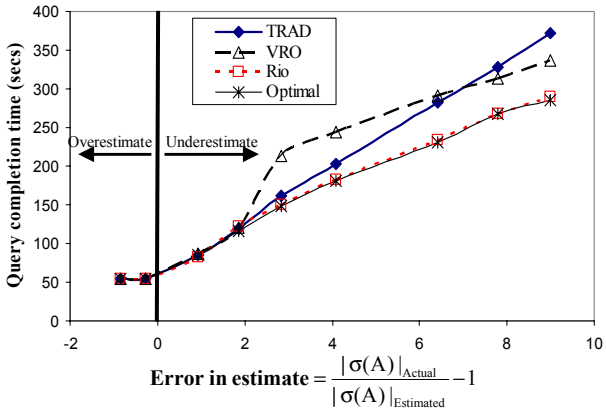


Figure 15 – σ(A) ⋈ C, 50MB per hash join

When |σ(A)| lies within the bounding box computed by Rio, re-optimization is avoided. In this case, the switch operator picks Plan P_{AC} or Plan P_{CA} appropriately, avoiding loss of work. For example, the switch operator picks Plan P_{AC} when |σ(A)|=32MB, which corresponds to an error of -0.26 in Figure 15. Plan P_{CA} is picked when |σ(A)|=160MB, which corresponds to an error of 2.84 in Figure 15. When |σ(A)|>193.2, which lies outside the

Table 2 – Plans used by different optimizers at sample points A, B, C, and D in Figure 16

Point	|σ1(A)|	TRAD	VRO	Rio	Optimal
A	**6 MB**	P17a	Inside validity range, runs Plan P17a	Outside bounding box, re-optimize, picks Plan P17a	P17a
B	**80 MB**	P17a	Inside validity range, runs Plan P17a	Inside bounding box, switch operator picks P17a	P17a
C	**160 MB**	P17a	Outside validity range, re-optimize, picks P17d	Inside bounding box, switch operator picks P17d	P17b
D	**310 MB**	P17a	Outside validity range: re-optimize, picks P17d	Outside bounding box, re-optimize, picks Plan P17b	P17b

bounding box, Rio will re-optimize with a fairly accurate value of |σ(A)| estimated via sampling. In this case, the optimal Plan P_{CA} gets picked. Therefore, Rio's performance is always close to that of the optimal plan for this query.

6.2 Three-way Join Queries

We now repeat the experiments in Section 6.1 with a query joining A, C, and O. There are selection predicates on A.accident_year (σ1) and O.cars (σ2). We removed the equi-height histogram on A.accident_year so that the optimizer uses a default estimate, and we vary the estimation error as in Section 6.1. The results are shown in Figure 16. The cardinality of σ2(O) is estimated accurately from an equi-height histogram.

The optimal plan for this query for low values of |σ1(A)| is Plan P17a shown in Figure 17. For higher values of A, Plan P17b in Figure 17 becomes optimal. Plan P17a is also the optimal plan for the single-point estimates of input sizes, hence TRAD always picks Plan P17a. Therefore, in the left part of Figure 16, TRAD performs as well as the optimal plan, but its performance deviates more and more from the optimal as the error increases.

Rio starts with the Plan P17c shown in Figure 17. This plan has two switch operators corresponding to the two joins. (Buffer operators are not shown in Figure 17.) The two member plans in the first switch operator are (i) hybrid hash join with σ1(A) as build and C as probe, and (ii) hybrid hash join with C as build and σ1(A) as probe. The switch operator will choose between these plans based on a run-time estimate of |σ1(A)| computed from a 1% sample of A. The two member plans in the second switch operator are (i) hybrid hash join with σ1(A) ⋈ C as build and σ2(O) as probe, and (ii) hybrid hash join with σ2(O) as build and σ1(A) ⋈ C as probe. The choice between these two plans will be made based on an estimate of |σ1(A) ⋈ C| from a 1% sample of σ1(A) ⋈ C obtained by sampling the join (recall Section 5.4.2.2). The bounding box on |σ1(A)| is the same as that in Section 6.1. The bounding boxes on |C| and |σ2(O)| effectively have zero width since these estimates are known to be accurate. When |σ1(A)|=6MB (Point A in Figure 16 and in Table 2), which corresponds to an error of -0.85 and lies outside the bounding box, Rio invokes re-optimization and picks the optimal Plan P17a. Similarly, when |σ1(A)|=160MB (Point C in Figure 16 and in Table 2), which corresponds to an error of 2.84 and is within the bounding box, both switch operators will pick the base relation input as the build, and execute Plan P17d in Figure 17. Thereby, when |σ1(A)|=160MB, Rio avoids re-optimization and the loss of pipelined work which results in the difference of around 72 seconds between Rio and VRO in this case.

The performance of Rio is always close to that of the optimal plan in Figure 16 except for an intermediate range of estimation errors. In this region, Rio picks Plan P17d which turns out to be suboptimal compared to Plan P17b. This region is a transition region where Plan P17d stops being optimal with respect to Plan P17b. Because of an overestimate of the join selectivity of C ⋈ σ2(O), Rio continues to pick Plan P17d as the optimal plan beyond the actual transition point. However, as the error in |σ1(A)| increases, Rio converges to the optimal plan again around an error of 4 in Figure 16.

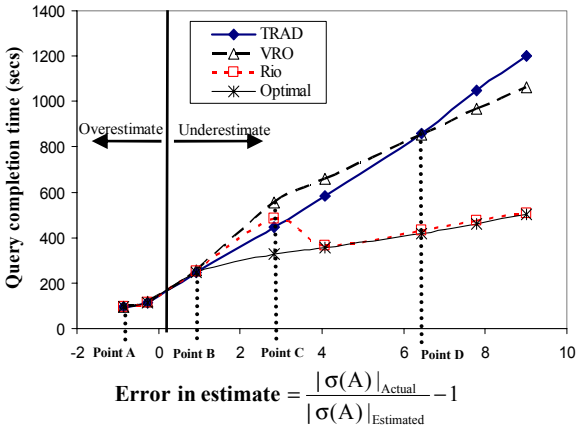


Figure 16 – σ1(A) ⋈ C ⋈ σ2(O), 50MB per hash join

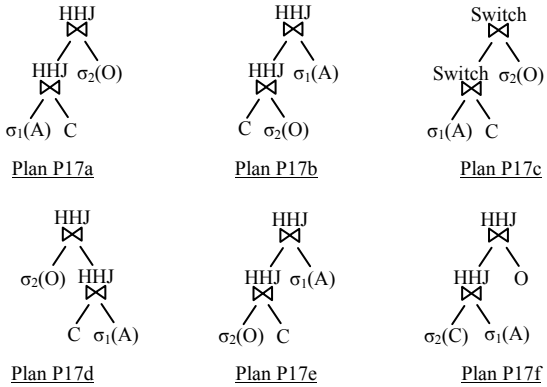


Figure 17 – Plans for A ⋈ C ⋈ O used in experiments

VRO starts with the same Plan P17a as TRAD, but with validity ranges added. When |σ1(A)|≤120MB, none of the validity ranges are violated. (|σ1(A)|=120MB corresponds to around 1.8 in Figure 16.) When |σ1(A)|>120MB, the validity range on σ1(A) ⋈ C is violated and VRO is forced to re-optimize. Note that at this point, VRO does not have an estimate of the actual size of |σ1(A)|. Based on the amount of A it has seen so far, VRO always picks Plan P17d on re-optimization and adds validity ranges. In addition to the overhead of re-optimization and the loss of pipelined work, the choice of Plan P17d illustrates one of the big problems with VRO. VRO gets stuck in a suboptimal plan as the validity ranges in Plan P17d will never fail because of an underestimate of |σ1(A)|: there is no better plan to join C and σ1(A) for large |σ1(A)| than the hybrid hash join with σ1(A) as the probe, even though there is a better plan for the entire query. A similar situation arises for the second join since σ1(A) is part of the probe input here as well. Hence, as illustrated by the results in

Figure 16, VRO performs badly as the estimation error in |σ1(A)| increases. This experiment illustrates one of the pitfalls of reactive re-optimization where the execution plan is decided before the issues affecting re-optimization are considered.

6.3 Correlation-based Mistakes

So far the estimation errors we considered were due to selection predicates on an attribute on which there was no histogram. A more common case of estimation errors is the presence of correlated attributes, which we consider in this section. We use a three-way join query on A, C, and O with selection predicates σ1(A) and σ2(O). Figure 18 shows the performance of three queries Q1, Q2, and Q3 which have different sets of correlated predicates on A, causing the optimizer to underestimate |σ1(A)| in each case. (Correlations usually lead to underestimates [20].) For example, Query Q2 contains predicates A.accident_with = "car", A.driver_status = "injured", and A.seat_belt_on = "on". |C| and |σ2(O)| are always estimated accurately. Figure 18 indicates that the estimation errors caused by correlated attributes result in performance trends for TRAD, VRO, and Rio similar to those shown in Sections 6.1 and 6.2. The reasons for these trends are also similar to those observed in Sections 6.1 and 6.2. The optimal plan for each query is Plan P17e in Figure 17 which Rio picks either because it is a robust plan (Q1) or because Rio discovers the estimation error and the actual estimate quickly because of randomization (Q2 and Q3).

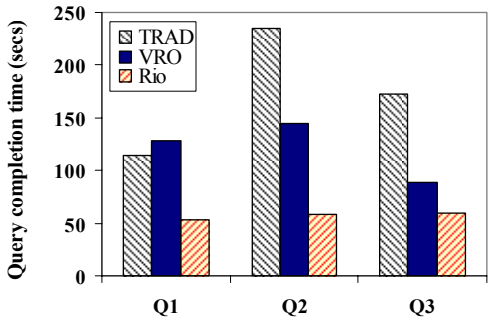


Figure 18 – Errors due to correlated predicates

6.4 Thrashing

So far we considered queries where the size of a single input is estimated incorrectly. In this section we consider the performance of VRO and Rio when the size of more than one input is estimated incorrectly. We use a three-way join query on A, C, and O with selection predicates σ1(A) and σ2(C). |σ1(A)| is underestimated significantly because σ is on an attribute with no histograms, while |σ2(C)| is underestimated slightly because the histogram on the corresponding attribute was built from a small sample of C. For this query, VRO thrashes and takes 690.38 seconds compared to 327.57 seconds for Rio. VRO starts with the optimal plan for the estimated statistics which is similar to Plan P17a in Figure 16. Because |σ2(C)| is underestimated, VRO computes an incorrect validity range for |σ1(A)|. This validity range is violated at run-time, and re-optimization picks Plan P17f. Since VRO does not have correct estimates of |σ1(A)| or |σ2(C)| at this point, it computes incorrect validity ranges which fail again. This thrashing results in the factor two slowdown of VRO compared to Rio. Rio invokes re-optimization once for this query when its run-time estimate of |σ1(A)| falls outside the bounding box. Because Rio estimates |σ1(A)| accurately at run-time using sampling, and also uses bounding boxes to allow for error in the estimate of |σ2(C)|, it finds the optimal plan in the first re-optimization step.

6.5 Increasing Query Complexity

In this section we compare the relative performance of TRAD, VRO, and Rio as we increase the number of joins in the query. The results are shown in Figure 19.

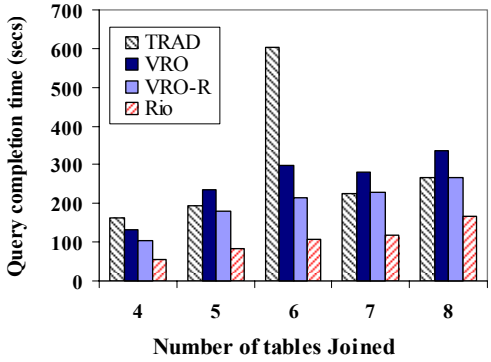


Figure 19 – Increasing query complexity

The dataset provided to us had four tables only (the actual dataset has around 30 tables [20]). For this experiment, we vertically partitioned each table into two and padded each partition with string fields to make it the same size as the original table. Each query had correlated predicates on half of the joined tables. Figure 19 shows the same trends observed in previous sections. The fraction of time spent by Rio and VRO in optimization steps was less than 1.7% in all cases in Figure 19. Roughly, the cost of each optimization phase in Rio is three times the cost of the single optimization phase in TRAD.

Figure 19 also shows the relative performance of VRO-R, which is the validity-ranges optimizer enhanced with our random-sample processing techniques from Section 5.4.2. While randomization improved the overall performance of VRO by reducing the time required to trigger re-optimization, the amount of wasted work, and the number of re-optimization steps, Rio still outperforms VRO-R by a significant amount.

7. FUTURE WORK

This paper proposes proactive re-optimization as a promising approach to deal with optimizer mistakes. We identified the core building blocks of proactive re-optimization: i) characterizing uncertainty in estimates of statistics using bounding boxes, ii) using the bounding boxes to pick robust plans and switchable plans, and iii) estimating statistics quickly and efficiently during execution. As a next step, we plan to evaluate our specific algorithms and implementation decisions against some alternative options:

- *Uncertainty and Bounding Boxes*. We used the uncertainty initialization and propagation rules from [18] to characterize the level of uncertainty in estimates and derive bounding boxes. An interesting alternative is to characterize uncertainty in terms of stochastic intervals [3].
- *Plan robustness*. Currently we characterize a plan as robust if its cost is close to optimal at three points in the bounding box.

Both the location and the number of these points in the bounding box require further study. Furthermore, alternative notions of plan robustness, e.g., based on expected costs [9] or confidence thresholds [3], will be considered.

- *Switchable plans*. We considered a fairly restricted notion of switchable plans based on the complete reuse of execution work. More flexible definitions, e.g., allowing re-ordering of operators in a pipeline, may give the optimizer more room to find switchable plans.
- *Random-sample processing.* Our approach so far is to merge random-sample processing with query execution to reduce the overhead. Random-sample processing could be used more aggressively to reduce the uncertainty in statistics even before starting query execution, introducing a new challenge in determining how much statistics collection to do in advance. A more general area of future work is to explore how randomization and ordered output can coexist best in Rio, e.g., in the context of Top-K queries.

8. ACKNOWLEDGEMENTS

We are extremely grateful to Jennifer Widom for helpful feedback and discussions. We would like to thank Guy Lohman and Volker Markl for providing us the DMV data and workload generator.

REFERENCES

[1] S. Acharya et al. Join Synopses for Approximate Query Answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, June 1999.

[2] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[3] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, June 2005.

[4] S. Babu and P. Bizarro. Adaptive Query Processing in the Looking Glass. In *Proc. of Second Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2005.

[5] H. Brönnimann et al. Efficient data reduction with EASE. In *Proc. of the Ninth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, Aug. 2003.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, June 1999.

[7] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004.

[8] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Trans. on Database Systems*, 9(2): 163-186, 1984.

[9] F. Chu, J. Halpern, and P. Seshadri. Least Expected Cost Query Optimization: An Exercise in Utility. In *Proc. of the 1999 ACM Symp. on the Principles of Database Systems*, May 1999.

[10] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. on Database Systems*, 27(3): 261-298, 2002.

[11] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data,* May 1989.

[12] J. Hellerstein, R. Avnur, and V. Raman. Informix Under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery Journal*, 4(4), Oct. 2000.

[13] A. Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data,* Jun. 2003.

[14] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, May 1991.

[15] Y. Ioannidis et al. Parametric Query Optimization. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, Aug. 1992.

[16] Z. Ives et al. An Adaptive Query Execution System for Data Integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data,* June 1999.

[17] Z. Ives, A. Halevy, and D. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004.

[18] 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*, June 1998.

[19] M. Kitsuregawa, M. Nakayama, and M. Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proc. of the 1989 Intl. Conf. on Very Large Data Bases*, Aug. 1989.

[20] V. Markl et al. Robust Query Processing through Progressive Optimization. In Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data, June 2004.

[21] H. Paques, L. Liu, and C. Pu. Distributed Query Adaptation and Its Trade-offs. *In Proc. of 2003 ACM Symp. on Applied Computing*, March 2003.

[22] V. Poosala et al. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, June 1996.

[23] P. Selinger et al. Access Path Selection in a Relational Database Management System. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, May 1979.

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

[25] P. Tucker et al. Exploiting Punctuation Semantics in Continuous Data Streams. *Transactions on Knowledge and Data Engineering*, 15(3): 555-568, 2003.

[26] T. Urhan, M. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[27] S. Viglas. Novel Query Optimization and Evaluation Techniques, Ph.D. Thesis, Department of Computer Sciences, University of Wisconsin-Madison, Jun 2003.