Robust Query Processing through Progressive Optimization

Volker Markl[*], Vijayshankar Raman[*], David Simmen[+], Guy Lohman[*], Hamid Pirahesh[*], Miso Cilimdzic[#]

[*]IBM Almaden Research Center, San Jose, CA, USA
[+]IBM Silicon Valley Lab, San Jose, CA, USA
[#]IBM Toronto Lab, Toronto, Canada

{marklv, lohman, pirahesh}@almaden.ibm.com, {ravijay, simmen}@us.ibm.com, cilimdzi@ca.ibm.com

Abstract

Virtually every commercial query optimizer chooses the best plan for a query using a cost model that relies heavily on accurate cardinality estimation. Cardinality estimation errors can occur due to the use of inaccurate statistics, invalid assumptions about attribute independence, parameter markers, and so on. Cardinality estimation errors may cause the optimizer to choose a sub-optimal plan. We present an approach to query processing that is extremely robust because it is able to detect and recover from cardinality estimation errors. We call this approach "progressive query optimization" (POP). POP validates cardinality estimates against actual values as measured during query execution. If there is significant disagreement between estimated and actual values, execution might be stopped and re-optimization might occur. Oscillation between optimization and execution steps can occur any number of times. A re-optimization step can exploit both the actual cardinality and partial results, computed during a previous execution step. Checkpoint operators (CHECK) validate the optimizer's cardinality estimates against actual cardinalities. Each CHECK has a *condition* that indicates the cardinality bounds within which a plan is valid. We compute this *validity range* through a novel *sensitivity analysis* of query plan operators. If the CHECK condition is violated, CHECK triggers *re-optimization.* POP has been prototyped in a leading commercial DBMS. An experimental evaluation of POP using TPC-H queries illustrates the robustness POP adds to query processing, while incurring only negligible overhead. A case-study applying POP to a real-world database and workload shows the potential of POP, accelerating complex OLAP queries by almost two orders of magnitude.

1 Introduction

Database management systems (DBMSs) traditionally compile SQL queries once and retain the resulting Query Execution Plan (*QEP*, or just *plan*) for repeated execution in the future, either stored in the database (for static optimization [CAK+81]) or in an in-memory cache (for dynamic queries), to save re-optimization cost. Most modern query optimizers determine the best plan for executing a given query by mathematically modeling the execution cost for each of many alternative QEPs and choosing the one with the cheapest estimated cost. The execution cost is largely dependent upon the number of rows (the *row cardinality*) that will be processed by each operator in the QEP, so the optimizer first estimates this incrementally as each predicate is applied by multiplying the base table's row cardinality by a *filter factor* – or *selectivity* – for each predicate in the query [SAC+79, Gel93, SS94, ARM89, Lyn88]. The estimation process typically begins with statistics of database characteristics that were collected prior to optimization, such as the number of rows for each table, histograms for each column [IC91, PIH+96, PI97], or sampled synopses [HS93].

While query optimizers do a remarkably good job of estimating both the cardinality and cost of most queries, many assumptions underlie their mathematical models, such as the currency of the database statistics and the independence of predicates. Outdated statistics or invalid assumptions may cause significant estimation errors in the cardinality, and hence the cost of a plan, causing sub-optimal plans to be chosen. One proposed solution is to continually re-optimize the plan as each row (or group of rows) is accessed [AH00], but this incurs impractically large re-optimization costs.

This paper introduces a practical compromise between the extremes of static optimization and continual optimization, called *progressive query optimization* (POP). POP provides a plan "insurance policy" by lazily triggering re-optimization in the midst of query execution whenever cardinality estimation errors indicate that the QEP might be sub-optimal. It does this by adding one or more *checkpoint* operators (CHECK), which compare the optimizer's cardinality estimates with the actual number of rows processed thus far, and trigger re-optimization if a pre-determined threshold on the error is exceeded. The threshold is to ensure that a better alternative QEP exists if it is exceeded. In this way, POP adds robustness to query processing, as suboptimal plans are not executed to completion anymore. This robustness substantially reduces the run-time of queries that were optimized using significantly wrong cardinality estimates. If the optimizer's estimates are fairly accurate, the only overhead incurred by POP is the added CPU cost of counting the rows for each CHECK and comparing them to the threshold. Only if the optimizer has grossly misestimated the cardinality at some CHECK, and thus is executing a plan that is likely to have disastrous performance, will the cost of re-optimization and re-execution be incurred. By treating the results computed up to CHECK as a temporary materialized view, the optimizer can both exploit its actual cardinality during re-optimization and reuse much of the already computed results during re-execution.

SIGMOD 2004, June 13–18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00.

The main contributions of this paper are (a) the concept and various flavors of the CHECK operator, (b) a novel method of determining validity ranges for QEPs, and (c) a rigorous performance analysis of a prototype of POP. We explore alternative checkpointing schemes that impose different levels of overhead, and investigate the tradeoffs in placing CHECK at different places in the QEP. We also introduce CHECK operators that permit re-optimization of pipelined plans without erroneously returning duplicate rows, a consideration implicitly ignored by prior re-optimization schemes in the literature.

In the remainder of Section 1 we first address the risk-opportunity trade-off of POP before surveying related work. Section 2 describes the overall concept of POP. Section 3 discusses various flavors of CHECK. Section 4 elaborates on CHECK placement. An experimental evaluation of a prototype of POP implemented on a leading commercial DBMS is given in Section 5. Section 6 presents the results of a case study using our prototype for a real-world database with complex OLAP queries. In Section 7 we discuss future work before we present our conclusions in Section 8.

1.1 Risk-Opportunity Tradeoffs

POP uses re-optimization to consecutively refine a query plan, with the goal of improving QEP quality and performance. There are two dimensions along which one can evaluate a re-optimization scheme:

- **Risk** – Risk refers to the degree to which re-optimization might not be worthwhile and may instead lead to performance regression. Performance regression may occur whenever re-optimization of a query results in the selection of the same or even worse (!) plan (improved cardinality estimates can and do lead to worse plans as we see in our experiments – this occurs because of cardinality errors canceling each other out). Regression can also occur when much of query execution needs to be repeated because intermediate results were not reused or if the query has a low run-time, thus re-optimization not being worthwhile.
- **Opportunity** - Opportunity refers to the aggressiveness of the re-optimization scheme for a query. Each CHECK added to a plan represents an opportunity for triggering re-optimization. The higher the number of CHECK operators the higher the opportunity for re-optimization. Opportunity is directly correlated to risk. Highly opportunistic schemes also are high risk because there is a greater possibility of slowing down query execution.

The recent direction in continuous query optimization (e.g. Eddies[AH00]) and many papers on streams (e.g. [MS02]) focus on maximizing opportunity while ignoring risk. In contrast, POP seeks to minimize risk and overhead through judicious placement of CHECK.

1.2 Related Work

While there has been a large body of work in query optimization, most work only addresses static optimization of queries.

[SLM+01] uses errors observed during previous query executions to optimize *future* queries. We complement this work by re-optimizing the *current* query.

General re-optimization of the currently running query was first introduced in [KD98], where – upon estimation error detection – the SQL statement of the currently running query is re-written to access specially materialized intermediate results as standard table access. We improve upon [KD98] in both the opportunity and risk metrics. First, [KD98] only re-optimizes hash joins and only if query results are not pipelined, whereas we present more generally applicable re-optimization schemes and study their risk-opportunity tradeoffs. Second, [KD98] uses an ad hoc cardinality error threshold to determine whether to re-optimize. We introduce a sensitivity analysis during optimization to calculate the validity range for estimates within which the plan is valid. Moreover, instead of always rewriting the original SQL query to reuse the hash join result, we give the intermediate results as option to the optimizer to reuse only if beneficial.

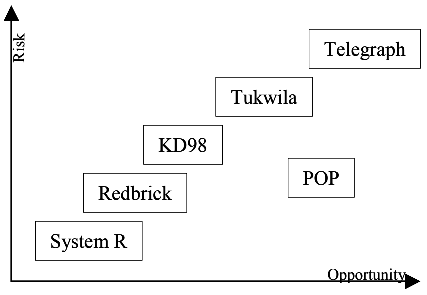

Figure 1: Risk/Opportunity Tradeoff of Various Re-Optimization Schemes

In the Tukwila system [Ives02], re-optimization is done by partitioning the data at each re-optimization point, with a final cleanup phase to combine results from previous phases. The main problems with this approach are: (a) each phase is executed without using the state generated by the previous phases, and (b) the final cleanup uses a specialized pipelined join algorithm rather than invoking the optimizer. The Query Scrambling project [UFA98] also re-optimizes queries, but its focus was on handling delayed sources.

In addition to the limitations discussed above, all of these systems externally re-write SQL queries to re-use prior results. This is only viable for simple read-only queries. Side effects like update operations cannot in general be rewritten into semantically correct SQL queries after partial execution. In contrast, POP works on arbitrary SQL queries possibly composed of sub-queries, updates, compiled in trigger and view maintenance operations, and so on. Furthermore, POP can handle concurrent update transactions or updates of common sub-expressions without compromising consistency and isolation --- the locking information needed for consistency is tied to the row identifier (rid), and is lost in all approaches that externally re-write queries.

A completely different strategy is to optimize routing of each row separately. Ingres [SWK76] used a simple scheme, in which each row is routed independently to *nested loop joins*. The Telegraph project generalizes this to a very fine granularity of re-optimization, in which a separate Eddy operator is used to continually adapt the row routing between other operators [AH00, RDH02]. Per-row routing gives high opportunity for re-optimization, but imposes a big overhead in steady state. Moreover, the Eddy routes each row along a greedy, locally optimal path that does not consider the overall query execution cost. While this works fine for Telegraph's interactive processing metric, a regular optimizer is needed to handle the more common completion time or total work metrics. Integrating Eddies with a traditional query optimizer remains a challenge for future work.

Among commercial systems, the DEC RDB system [AZ96] ran multiple access methods competitively before picking one. To the best of our knowledge, the only commercial DBMS currently shipping with a form of POP is the Redbrick DBMS, which specializes in processing queries over star schemas. The specific star schema plan used is not fully determined until execution time. Intermediate results of all dimension table accesses are first computed. The cardinality of those intermediate results is then used to select the appropriate method for accessing the fact tables. While this product uses progressive re-optimization, it does so only for a very specific query execution strategy. The issues of arbitrary CHECK placement, join re-ordering, and intermediate result re-use are not addressed.

The closest analogy to our validity range computation method is the work on parametric optimization (e.g. [CG94, HS02]) where different plans are generated for different intervals of the optimization parameters. The main problem here is the combinatorial explosion of the number of plans that need to be generated, stored, loaded, and decided among at runtime. We avoid this explosion by embedding validity range computation into the optimizer pruning phase (Section 2.2).

2 Progressive Query Optimization

Progressive Query Optimization (POP) is comprised of several key aspects for protecting against query processing disaster due to the choice of a suboptimal QEP.

1. POP can detect a suboptimal QEP in the midst of execution and cause it to be re-optimized. Alternating optimization and execution steps can occur any number of times. Partial result records can be pipelined to the application at each execution step using techniques to prevent duplicate rows from being returned to the application.
2. During each execution step, POP monitors the actual values of key estimated parameters used to select the QEP and feeds this information back into a re-optimization step. This aspect of POP improves the likelihood that an optimal plan will be selected for the next execution step.
3. POP also makes materialized partial results available for reuse during the next execution step. Rather than force the optimizer to reuse these partial results by rewriting the query or some other means, they are packaged as materialized views in order to take advantage of the optimizer's ability to make a cost-based decision with regard to their reuse (see section 2.3 for more details).

Checkpoints are the POP points of control. A checkpoint inserted into a QEP is effectively an assertion to ensure that optimization parameter estimates agree with the actual values for those parameters as measured during query execution. Our current research focuses on the monitoring of cardinality estimates; however, a checkpoint could monitor other properties as well. A checkpoint monitors the number of rows flowing from a producer to a consumer during query execution. It may also buffers rows that it sees. A checkpoint suspends query execution and triggers re-optimization if the number of rows it sees violates the check condition. In our prototype of POP a check condition defines the cardinality range (or *check range*) for which the check condition is true. Determining check ranges depends on the ability to compute the *validity range* for each subplan *P* rooted with plan operator o, which defines for each input stream into *o* the range of cardinalities for which *o* is the optimal root operator for *P* as discussed in more detail in section 2.2. Our system implements various flavors of checkpoints (as discussed in section 3),

Checkpoints are manifested in POP plans by CHECK operators. CHECK has no relational semantics. Each CHECK has a *check range* parameter defining a range of cardinalities [*l*, *u*]. The check range is dependent on the cardinality estimate as well as the remainder of the QEP above the CHECK. CHECK is successful when the actual cardinality *a* is within the check range, i.e., $a \in [l, u]$. If CHECK succeeds, query processing will continue normally; otherwise, query execution is terminated and re-optimization is triggered. Actual cardinality estimates measured during the partial execution of the query, occurring up to the point where the check range was violated, are fed back into the re-optimization phase. Moreover, materialized intermediate results are made available for re-use during the re-optimization phase. The decision as to whether or not intermediate results are reused during re-optimization is based upon cost analysis. As described later, it may under certain circumstances be preferable to avoid reusing these results.

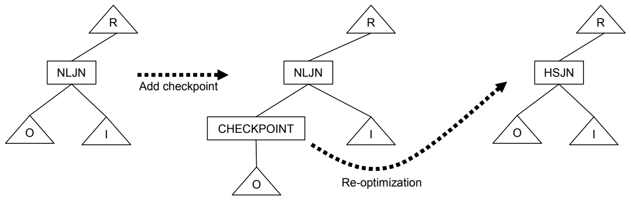

Figure 2: Adding CHECK to the outer of a NLJN

An example of POP is given in Figure 2. The QEP in the left part of the figure joins the outer (O) and inner (I) sub-plans using the (index) nested-loop join (NLJN) method before processing the remainder of the plan (R). The choice of the operator joining O and I depends heavily on the cardinality estimate for the result of the sub-plan O. Usually the optimizer will prefer NLJN for joining O and I, when the cardinality of O is small relative to I and there is an index on I to apply the join predicate. If the cardinality of O is much larger than estimated, another join method, such as hash-join (HSJN) or merge-join (MGJN), might be more efficient, and thus preferred by the optimizer.

Since the choice of an inappropriate join method can result in performance degradations of orders of magnitude, adding CHECK to the outer sub-plan of an NLJN helps to prevent the execution of sub-optimal plans and thus bad query response times. CHECK added above O in the middle part of Figure 2 ensures that the NLJN method is optimal not only for the cardinalities estimated at optimization time, but also for the actual cardinalities measured at runtime, thus making this plan more robust. When the check range is violated, re-optimization of the query is triggered, which might result in a significant change in the QEP such as replacing NLJN in

Figure 2 with a more suitable join method such as hash join (HSJN).

2.1 Architecture of POP

Extending a DBMS with POP capability involves:

a) Adding logic to the plan generator of the query optimizer to determine the check range by determining the cardinality range for which any given operator is optimal in the current plan.

b) Adding logic to the post-pass of the optimizer for deciding the most judicious location of CHECKs

c) Adding code generator logic for translating CHECK into executable code

d) Adding logic to the runtime system for interpreting CHECK.

e) Adding logic to exploit intermediate results when CHECK fails, so that work already done can be reused during re-optimization.

To illustrate those enhancements to the architecture of a DMBS, Figure 1 distinguishes the *initial run* (first query execution until the violation of the check range triggered re-optimization) and the *re-optimization run* of a query for explanatory purposes. Actually, the re-optimization run could again add CHECKs to the new QEP and become the initial run for a second re-optimization.

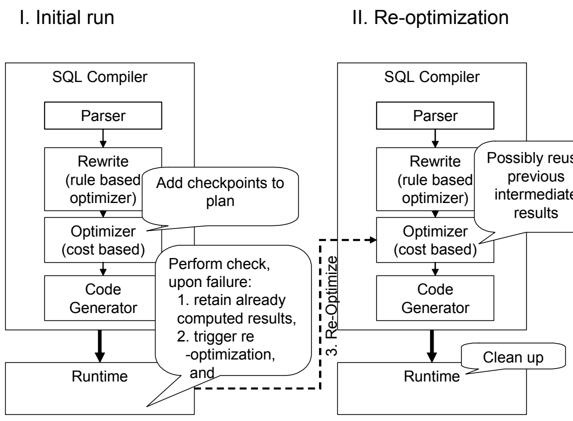
I. Initial run
II. Re-optimization
SQL Compiler
Parser
Rewrite (rule based optimizer)
Add checkpoints to plan
Optimizer (cost based)
Code Generator
Perform check, upon failure: 1. retain already computed results, 2. trigger re -optimization, and
Runtime
3. Re-Optimize
SQL Compiler
Parser
Rewrite (rule based optimizer)
Possibly reuse previous intermediate results
Optimizer (cost based)
Code Generator
Clean up
Runtime

Figure 1: Progressive Optimization architecture

During the initial optimization of a query, the post-pass of the optimizer adds CHECK operators to the QEP based on the reliability of an estimate as well as the potential harm of an estimation error. When CHECK is executed, the check range is compared to the actual cardinality observed by the runtime system. If the check range is violated, the runtime system retains intermediate results together with their actual cardinality values and triggers re-optimization of the query. Actual cardinalities measured during the initial run help the re-optimization step avoid the same mistake. After optimization and execution of the query in the re-optimization run, cleanup actions are necessary to remove the intermediate results and free locks on tables and indexes used during the initial run.

2.2 Computation of Validity Ranges

It is crucial to minimize risk of POP by re-optimizing only when we are sure that the plan will change. In general, this is the parametric query optimization problem, computing the optimal plan for every possible combination of parameter values [CG94, HS02]. For POP we avoid this exponential explosion of parameters by forming a *validity range* for each edge of the QEP.

Definition: Consider a plan edge *e* that flows rows into operator *o,* and let P be the subplan rooted at *o.* The *validity range* for *e* is an upper and lower bound on the number of rows flowing through *e,* such that if the range is violated at runtime, we can guarantee P is suboptimal with respect to the optimizer's cost model. This range is defined conservatively, i.e., even within the validity range P may become suboptimal with respect to alternative QEP we do consider. This conservative definition is fine, since we only want to avoid needless re-optimization.

The main advantage of validity ranges over parametric optimization is that we need not enumerate beforehand all possible optimal plans under all possible parameter values – we only need the cardinality ranges within which the chosen plan remains optimal. However we cannot use ad hoc thresholds on cardinality errors because the effect of cardinalities on query optimality is very complex. A 100x error in cardinality of the NATION table of a TPC-H schema may make no difference to plan optimality, whereas a 10 percent increase in ORDERS may turn a two-stage hash join into a three-stage hash join, making the query plan highly suboptimal.

POP computes validity ranges during the plan enumeration and pruning phases of the optimizer through a plan sensitivity analysis. It iteratively narrows the validity range for each input to an operator of the currently optimal plans, when pruning alternative plans during optimization.

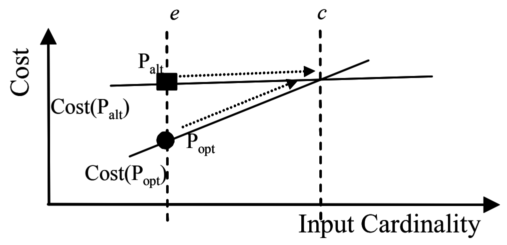

Figure 4: Computing the Upper Bound of a Validity Range

Suppose that during dynamic programming, plan P_{opt} with root operator o_{opt} is being compared with another plan P_{alt} having the same properties (joined tables, applied predicates, sort order, projected columns) and different only in the root operator $o_{alt.}$ Suppose that P_{opt} dominates, and we prune P_{alt} due to its higher cost.

The cost for P_{opt} and P_{alt} is a function of the cardinalities of the input edges of the root operator. Consider one edge with estimated cardinality *e.* Figure 4 illustrates how we can narrow the upper bound of the validity range of this edge. As we prune plan P_{alt}, we determine if there exists an input cardinality $c > e$ such that the cost functions cost(P_{alt}, *c*) and cost(P_{opt}, *c*) intersect. We do this by solving for the root of $cost(P_{alt}, c) - cost(P_{opt}, c) = 0$. When a root operator has multiple input edges (e.g., joins), we need to find the roots by treating the cost functions of $P_{opt}$and P_{alt} as multi-variate functions of the input cardinalities.

This process is repeated for each alternative plan that is pruned with P_{opt}. Each time we check if the root *c* is less than the current upper bound and adjust the bound accordingly.

Root-Finding through Modified Newton-Raphson

Prior work on parametric optimization assumes that cost functions can be modeled using simple analytical models like piecewise-linear functions. In practice, however, cost models for plan operators are extremely complex, exceeding several thousand lines of code. Moreover these cost models are continually being refined as their shortcomings are revealed in particular environments. Therefore it is not practical to analytically compute the validity ranges because the formulas will have to be continually updated as the code changes.

In addition to being complex code, the optimizer cost functions are also not smooth, not even always continuous. We even have identified situations where the cost functions monotonically decrease with the input cardinalities. So simple binary-search techniques as in [GW89] will not work.

Instead we convert validity range estimation into a numerical solving problem. In practice we have found the Newton-Raphson method to be very effective, when combined with a method to escape discontinuities and non-differentiable points in the cost functions.

```
1) If operator $o_{opt}$ with child edge e prunes operator $o_{alt}$.
2) Let ub be the current upper bound for e's cardinality
   (initialized to infinity).
3) while cost($o_{opt}$) < cost($o_{alt}$) do (cap at 3 iterations):
    a. currDiff = cost($o_{alt}$) - cost($o_{opt}$)
    b. card(e) = card(e) ×1.1 //need another pt to find gradient
    c. newDiff = cost($o_{alt}$) - cost($o_{opt}$)
    d. if newDiff < 0 break
    e. // Newton Raphson is diverging: jump
       if (newDiff > currDiff)   card(e) ×= 10
    f. card(e) ×= 1 + newDiff/(11× (-newDiff+currDiff))
    g. If ub >card(e) set ub = card(e)
```

Figure 5: Validity Range Estimation Method (for upper bound *ub*, similar method applies for lower bounds)

Figure 5 above shows our modified Newton-Raphson method. Notice that Newton-Raphson terminates as soon as we get a cost inversion. We can stop the approximation at any point and our suboptimality detection is guaranteed to be conservative; it will not compute a false suboptimality bound. Erring on the conservative side is acceptable because it will not cause re-optimization with the risk of regression. In experiments we have determined that merely three iterations of Newton-Raphson results in finding a good validity range.

Structurally Equivalent Plans

The method described above is very simple. It is easily embedded into a standard dynamic programming optimizer, by enhancing the prune function. The only overhead is the repeated evaluation of the cost functions for operators o_{opt} and o_{alt} with alternate cardinalities.

Our method is limited as it only detects suboptimality of the root operator and P_{opt} and P_{alt} must share the same input edges. It is easy to see for example that the validity range might miss a cross-over point with a plan that uses a different join order (and hence has different input edges).

Formally, we can show that our method will find suboptimality with respect to all *structurally equivalent* plans.

Definition: Two plans P1, P2 with identical properties (joined tables, predicates, projected columns and sort order) are *structurally equivalent* if they share the same set of edges, where an edge is defined by the set of rows flowing through it during query execution.

Theorem: Suppose that during execution of a plan P with edges $\{e_1, e_2, \ldots, e_m\}$, the edges $e_{i1}, e_{i2}, \ldots, e_{ik}$ are seen to be erroneous (i.e. cardinality of rows flowing along them is incorrectly estimated). Let $o_{i1}, o_{i2}, \ldots, o_{ik}$ be the operators incident to and above each of these edges, and $P_{i1}, P_{i2}, \ldots, P_{ik}$ be the subplans of P rooted at these operators. The following statements are equivalent (proof by induction [MR+04]):

(a) P is suboptimal with respect to another plan P' that has the same set of edges $\{e_1, e_2, \ldots, e_m\}$

(b) At least one of $P_{i1}, P_{i2}, \ldots, P_{ik}$ is suboptimal given the cardinality errors in those edges in $\{e_1, e_2, \ldots, e_m\}$ that lie under them.

(c) At least one of $o_{i1}, o_{i2}, \ldots, o_{ik}$ is a suboptimal operator given the cardinality errors in $\{e_1, e_2, \ldots, e_m\}$ that are in its input edges**.**

Structural equivalence includes plans with alternative choices for physical operators (e.g. different join algorithms), and also plans with reverse orders of the inner and outer children of various operators. However, it excludes plans with alternative choices of join orders. It is this restriction that allows use to reduce our search space for suboptimality and avoid enumerating all possible optimal plans. Moreover, even if we were able to search for suboptimality against plans with alternative join orders, it would be a very risky strategy. Consider a plan that uses the join order $(R \bowtie S) \bowtie T$. During run time, we can never observe the cardinality of $R \bowtie T$. So if we were to assert that $(R \bowtie S) \bowtie T$. is suboptimal, we would be making an arbitrary guess as to the correlation of the predicates on the R and T tables. The problem is that the optimizer will assume independence, and this often leads to cardinality underestimates because positive correlations are common. Not explicitly modeling validity ranges for join order changes helps to avoid such guesses to minimize regression risk. Note, however, that when a validity range is violated, the new optimal plan with respect to the optimizer's model may very well create a new join order. We are merely conservative in not basing validity ranges on join order changes except for commutation, but we do not prevent the optimizer from choosing a new join order for the re-optimized query.

2.3 Exploiting Intermediate Results

In order to efficiently re-optimize, already computed intermediate results should be exploited whenever possible. We exploit materialized views (MV) to easily and elegantly integrate POP with intermediate result exploitation.

Before recursively calling the SQL compiler, CHECK promotes each intermediate result to a temporary MV, having the cardinality of the intermediate result in its catalog statistics. Thus exact cardinalities are available for all intermediate results for re-optimization. During re-optimization the optimizer will also consider table accesses to the materialized views as an alternative sub-plan that is compared to sub-plans that re-create intermediate result from scratch. The optimizer could even create an index on the materialized view before re-using it if worthwhile.

Re-optimization takes place in the same transaction as the initial partial execution and holds all locks acquired previously. Therefore it is guaranteed that all persisted results are still transactionally correct when re-execution takes place.

To minimize the overhead and thereby the risk of re-optimization, these intermediate results are not necessarily written out to disk. Rather the temporarily MV has a pointer to the actual runtime object for the scan from the current execution. If this view is reused, the fields of this in-memory object are modified to satisfy the new plan (e.g., the internal id's for each column of this scan may change when the plan changes).

The standard mechanisms for matching MVs to a query is used to determine if the MV created from the intermediate result can be used for some part of the query. Once the intermediate results have been matched to the query, the query optimizer will construct plans that exploit each matched MV in addition to the original plans, using the known cardinality for the subplan corresponding to that MV in all cases, and then choose the cheapest plan as usual. In most cases, a plan that re-uses the MV representing the intermediate result should win. Unlike regular MVs, however, the runtime system has to remember to remove any of these temporarily materialized views after completing query execution.

If the plan under CHECK performs a side-effect (insert/delete/update), the intermediate results must always be matched and reused – otherwise the side-effect would be applied twice.

Intuitively it seems that intermediate results should always be reused rather than be thrown away. But this is not always true. A wrong initial choice of join order, for instance, might create a prohibitively large intermediate result that would have been avoided with a different join order. Moreover, we have found that many cardinality estimation errors are due to violations of the independence assumption between predicates, and are therefore underestimates, leading to larger-than-expected intermediate results. Using this intermediate result could incur a much higher cost than restarting from scratch. Instead of always using intermediate results, POP gives the optimizer the choice whether or not to use the intermediate results. This choice is based on the optimizer's cost model, which is enhanced by better cardinality and statistics information obtained from the previous partial execution of the query.

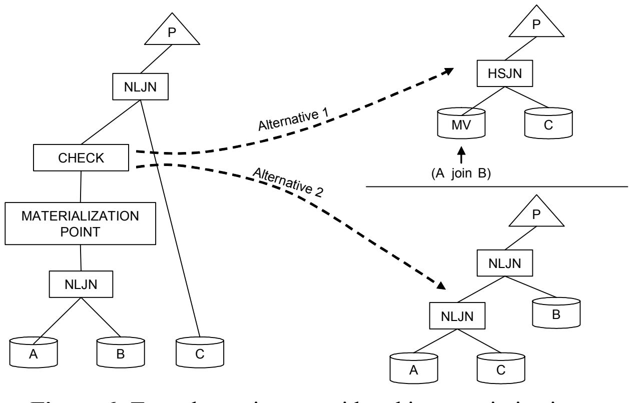

Figure 6: Two alternatives considered in re-optimization

The right part of Figure 6 shows two alternatives QEPs among other alternatives that the query optimizer will consider when re-optimizing the QEP in the left part of the figure at the CHECK. Alternative 1 reuses the materialized view created from the intermediate result at the materialization point below CHECK, whereas Alternative 2 uses a different join order and does not reuse the previous work. The optimizer's cost model will decide which alternative to choose for the re- optimized query.

3 Variants of CHECK

The main metrics to evaluate CHECK are the *risk* and *opportunity* of re-optimization at the checkpoint. An additional metric is its usability in *pipelined plans*, i.e., QEPs that do not have any operators that block row processing, but stream all rows directly to the user in order to reduce the time that that user has to wait before seeing the query's first results. Re-optimization in this case might be triggered after some results have already been returned. Without buffering or *compensating* for those rows, re-optimization will result in unexpected duplicates, which is inconsistent with the semantics of the original query.

We now present five flavors of CHECK to meet these challenges: *lazy checking (LC)*, *lazy checking with eager materialization (LCEM), eager checking without compensation (ECWC), eager checking with buffering (ECB)*, and *eager checking with deferred compensation* (ECDC). The first three apply only to non-pipelined plans, and the last two apply to all plans.

3.1 Lazy Checking

Lazy checking (LC) piggybacks on *materialization points*, i.e., points in a QEP where an entire intermediate result is materialized before proceeding with further operators of the plan. Examples for such materialization points are a) the SORT operator (which sorts its input, e.g. for a sort-merge join or group-by), b) the TEMP operator (which creates a temporary table, e.g., for caching subquery results), and c) the build side of the hash join operator. Placing CHECK above a materialization point means that the cardinality of the materialization point will be checked exactly once, that is, after the materialization has been completed. Materialization points are ideal checkpoints for two reasons. First, LC needs no compensation, because no results could have been output before re-optimization. Second, the materialization creates intermediate results that can be reused by the re-optimized query.

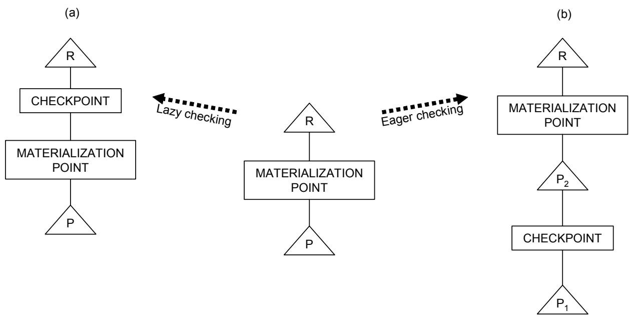

Figure 7: Lazy checking (LC) and eager checking without compensation (ECWC)

Lazy checking is depicted in the left half of Figure 7, where the QEP in the middle of the figure processes its sub-plan P and materializes the result of P at a materialization point. After materialization, the result is further processed by sub-plan R. The left part of the figure shows how POP adds LC above the materialization point.

3.2 Lazy Check with Eager Materialization

Although materialization points allow very efficient re-optimization, they may not occur frequently. If we want to check above a QEP node and there is no materialization, an alternative is to explicitly add a MATERIALIZATION-CHECK pair that first materializes the result and blocks any pipelining. Upon complete construction of the materialized intermediate result, the check range is evaluated. We call this flavor of checkpoint Lazy Checks with Eager Materialization (LCEMs).

We cannot add LCEMs recklessly because of the extra overhead of materialization. Instead we use the following heuristic. Among the various join operators in the plan, merge joins typically have naturally-occurring materializations on both inputs, and hash joins have materialization on the build side. So it is mainly the various varieties of NLJN that may have no materialized inputs and therefore need LCEMs. Therefore our heuristic is to add LCEMs on the outer side of every NLJN (unless the outer stream already has a materialization operator).

For the common case of equi-joins, the fact that the optimizer picked NLJN over HSJN or MGJN suggests that the cardinality of the outer is small (because the cost of NLJN is roughly the outer cardinality times the cost of probing or scanning the inner). If the optimizer's cardinality estimate was correct, materializing the outer will not be too expensive, as we verify experimentally in Section 5. If not, it will be worth the overhead to avoid such a mistake.

3.3 Eager Checking (ECWC, ECDC, ECB)

A main weakness of lazy checking is that the materialized result may be too large, and it may be suboptimal to compute them at all. Sometimes this can have serious implications: if the intermediate result cardinality was badly underestimated, there may not be enough temporary space to hold the materialized result! Eager Checking is an aggressive alternative that re-optimizes without waiting for materialization, thereby reacting faster to cardinality errors. Clearly, results could have been output to the user by that time, in which case we must compensate for this. Furthermore, eager checking may result in throwing away work, and thus are of higher risk than lazy checking. There are 3 flavors of eager checking:

Eager Checking without Compensation

An Eager Check without Compensation (ECWC) is a checkpoint that has a materialization point as its ancestor, i.e., which is executed later, and therefore needs no compensation. The right half of Figure 7 shows how CHECK is pushed down below a materialization point, breaking the sub-plan P into two sub-plans P_1 and P_2 and performing eager checking on P_1.

Eager CHECK operators can also be placed in pipelined (sub)plans, and thus may require compensation in order to avoid false duplicates. We distinguish the following two kinds of eager CHECK operators:

Eager Check with Buffering

An *Eager Check with Buffering (ECB)* is a combination of CHECK and a buffer, testing if the actual cardinality is above or below a certain threshold. ECB buffers the rows passing through it until it is confident that ECB will either fail or succeed. It thus supports pipelining, though with a delay.

Specifically, an ECB with a threshold range [0, *b*) or [*b*, ∞] accepts and buffers up to *b* rows like a valve. An ECB with range [0, *b*) ([*b*, ∞]) will succeed (fail), when its child in the QEP returns no more rows and the buffer contains less than *b* rows at this time. An ECB with range [0, *b*) ([*b*, ∞]) will fail (succeed), when the *b*th row is inserted into the buffer. If ECB fails, re-optimization is triggered. If ECB succeeds, pipelined execution continues. The parent operator above ECB will first process the rows from the buffer. If the buffer is exhausted for a [*b*, ∞] ECB, further rows are requested from the operator below the ECB.

ECBs can be implemented with a *buffered check* (BUFCHECK) operator. Figure 8 illustrates a BUFCHECK with buffer B on the outer sub-plan O of a NLJN. This buffer blocks the join until either the buffer has been filled or O finishes. ECB can be used instead of LCEM for checking the outer cardinality of a NLJN, because pipelining can be blocked for a short while in order to ensure that NLJN is the proper join method. An ECB can also help SORT or HSJN builds, if these run out of temporary space when creating their results, by re-optimizing instead of signaling an error.

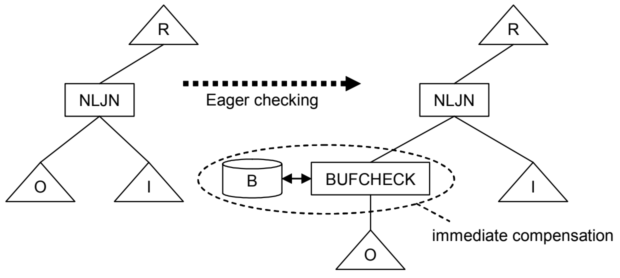

Figure 8: Eager checking with Buffering

ECB and LCEM

Note that an ECB can easily *morph into an LCEM* by simply waiting to re-optimize (on a check failure) until its input is exhausted.

Eager Check with Deferred Compensation

For queries only containing select, project and join (SPJ) operators we can avoid delaying pipelining by using another flavor of Eager Check called *Eager checking with deferred compensation* (ECDC) that transfers each row to its parent operator in a pipelined manner. To allow for compensation in case of re-optimization, the identifiers of all rows (rids) returned to the user are stored in a side table S. The new plan of the query needs to compensate for these prior results by doing an anti join between S and the new result stream.

ECDC is depicted in Figure 9. In the middle part of the figure, the pipelined plan P has been broken up at compile time into the sub-plans P_1 and P_2, and a checkpoint has been inserted between the two sub-plans. The RETURN plan operator in the figure denotes the operation that returns rows to the user. Because of deferred compensation, ECDC neither delays pipelining nor buffers any rows. However, in order to enable deferred compensation, an INSERT operator is inserted just below the return operator. INSERT uses a temporary table S to remember the rids of all rows that have been returned to the user. These rids may need to be constructed if the row has been derived from a base table. If re-optimization is triggered, the optimizer adds an anti join (set difference) plan operator on top of the re-optimized QEP P^* to compensate for already returned rows from the initial run of the query.

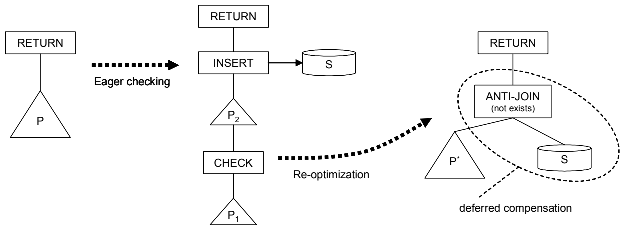

Figure 9 Eager checking with deferred compensation

Figure 10 shows the implementation of the check (CHECK) and buffered check (BUFCHECK) operators via an open/next/close model. The implementation of check can be simplified if the DBMS maintains counters for each plan operator. In this case, the check operator can directly refer to the counters of the operator below CHECK. Similarly, if CHECK is only placed above a materialization point, checking can be optimized to be only executed once (i.e., after the materialization has completed) and refer to the counter of the materialized intermediate result.

Figure 10: Check implementation for check range [low,high]

```
CHECK.OPEN:
  count = 0;
CHECK.NEXT:
  count++;
  r = childStream.next();
  if count > high
    call re-optimization;
  if count < low and r = EOF
    call re-optimization;
  else
    return r;
CHECK.CLOSE:
  ∅
```

```
BUFCHECK.OPEN:
  count = 0;
  allocate B of size b;  // buffer
  for i = 0 to b do
    B[i] = childStream.next();
    if childStream.EOF()
      and i < low
    call re-optimization;
BUFCHECK.NEXT:
  count++;
  if high < count
    call re-optimization;
  if count < b
    return B[count];
  else
    return childStream.next();
BUFCHECK.CLOSE:
  free B;
```

3.4 Risks and Opportunities for each flavor of Checkpoint

Lazy checks (LCs) impose the least risk during query processing because their input is materialized and can be reused. But their opportunity is limited to materialization points in the plan.

Lazy checks with Eager Materialization (LCEMs) impose the additional overhead of materializing results, and could thus be more risky. So we choose to place LCEMs only on the outer side of NLJN, where cardinalities are likely to be small. By introducing these artificial materialization points, LCEMs provide greater re-optimization opportunities.

The main problem with LCs and LCEMs is that they wait for full materialization before re-optimizing. This can be bad if the result is much larger than expected -- LCEMs are especially affected, because there the materialization is artificially introduced.

Eager checks with Buffering (ECBs) avoid this problem by checking before materialization is completed. The penalty is that the sub-plan being materialized has to be completely rerun, modulo other materialization points within it. In general we want to couple both approaches, placing an LCEM above an ECB so that the ECB can prevent the materialization from growing beyond bounds. The relative risk of inserting the ECBs vs. the LCEM depends on the relative costs of rerunning the outer and materializing the results. Also, like any eager CHECK, ECBs terminate early and thus will not enable the optimizer to use the correct cardinality for the subplan during re-optimization. They merely give the optimizer a lower bound for the correct cardinality that is higher than the previous estimate, ensuring that a different plan will chosen, but there is no guarantee that the new plan will be optimal.

ECWC and ECDC give much greater opportunities for re-optimization. ECWC can be placed anywhere below materialization points. ECDC works even in pipelined plans and requires only one buffer for the entire query, regardless of how many checkpoints exist in the QEP. Because of the anti-join post-processing of the re-optimized query, ECDC reduces the overhead of the initial run of the query and puts more of the cost upon re-optimization, which can be good if re-optimization is rare. As a penalty for this virtually unlimited opportunity for re-optimization, ECWC and ECDC have high risk, because they fail to retain work done.

4 CHECK Placement

Table 1 summarizes the 5 flavors of checkpoints.

LCEM and ECB checkpoints are placed on the outer side of nested loop joins during plan enumeration. After the optimal plan has been chosen, LC checkpoints are placed above materialization operators. ECWC and ECDC checkpoints can be placed arbitrarily.

In our current implementation, the materialization points we consider are SORTs and TEMPs. The two other kinds of reusable results that arise during query processing are: (a) the build side of hash joins, and (b) rid-lists generated from indexes. We have found SORT and TEMP reuse alone to provide for significant performance improvements, but plan to enhance our prototype to reuse further intermediate results in order to make re-optimization even more efficient.

Our validity range estimation ensures that checkpoints will not trigger re-optimization unless an alternative better plan is available. However, LCEM and ECB checkpoints induce the overhead of an extra materialization even with no re-optimization. Moreover, even if a better plan is available, we might throw away so much additional work using eager checking (with ECB, ECWC and ECDC checkpoints) that the overall execution is slower. As we intend to be conservative, the default behavior of our prototype is to only place LC and

Table 1: Placement, Risk and Opportunity for various flavors of checkpoints

Checkpoint Type	Placement	Risk	Opportunity
Lazy Check (LC)	CHECK Above materialization points	Very Low -- only context switching	Low, only at materialization points
Lazy Check with Eager Materialization (LCEM)	CHECK-Materialization pairs on outer of NLJN	Context Switching + materialization overhead	Materialization points and NLJN outers
Eager Check with Buffering (ECB)	BUFCHECK on outer of NLJN.	High – exact cardinality of subplan below ECB not available	Can reoptimize anytime during materialization
Eager Check without compensation (ECWC)	CHECK below materialization points	High – may throw away arbitrary amount of work during reoptimization	Anywhere below a materialization point
Eager Check with deferred compensation (ECDC)	CHECK and INSERT before reoptimization; anti-join afterwards	High – may throw away arbitrary amount of work during reoptimization	Anywhere in the plan of an SPJ-query

LCEM checkpoints, where LCEM is placed to guard the outer of a NLJN. (in one experiment we also show the opportunity of ECB checkpoints by selectively enabling them for that experiment). General purpose placement of ECB, ECWC, and ECDC is disabled by default.

To avoid needless materialization, we have an additional restriction on checkpoint placement: there must be alternative query plans for the part above the checkpoint. This information is obtained from the optimizer itself during plan enumeration. We also do not place CHECK operators in simple queries with an estimated cost below a certain threshold, in order to avoid the cost of monitoring and re-optimization for those kinds of queries.

More generally, a checkpoint is useful only if the cardinality estimate at that point may be erroneous. It remains future work to investigate a detailed confidence model for cardinalities at plan edges. The number of times assumptions were used instead of actual knowledge in order to compute an estimate might be a starting point for a heuristics for a reliability measure.

5 Performance Analysis

We study the performance of POP and the various flavors of checkpoints using a prototype implemented in a leading commercial DBMS. While this section analyzes the robustness that POP adds to query processing as well as its risk and opportunities, we give performance numbers of applying POP in a real-world use-case in Section 6. We currently implement LC, LCEM, ECB and ECWC checkpoints. For code simplicity, we implement BUFCHECK by placing a TEMP over a CHECK, with the TEMP acting as buffer. As mentioned before, we reuse materialized results from TEMP and SORT operators. Our current implementation does not reuse hash join builds or rid-lists during re-optimization.

We first present experiments describing the benefits of POP in improving query robustness. Next in Section 5.2 we analyze the risk and opportunity of each flavor of checkpoint.

We used the TPC-H schema and queries to analyze the performance of POP in this section. All experiments were performed on a lightly-loaded Power PC with a Power3-II 375Mhz CPU, 4MB L2 cache, 3 GB real memory, 2GB swap file.

5.1 Robustness in Case of Estimation Errors

We illustrate how POP improves robustness in query processing using TPC-H Q10, which is a complex join of LINEITEM, ORDER, and CUSTOMER. To simulate and analyze estimation errors, we replace the literal in the selection predicate on LINEITEM with a parameter marker. In this case the DBMS does not know the value of the literal at optimization time, and chooses a default selectivity, resulting in a default plan using the following join order and join operators: (CUSTOMERS HSJN (LINEITEM NLJN ORDER)). At run time, we bind the parameter marker to all possible values of the literal, thereby varying the selectivity on LINEITEM.

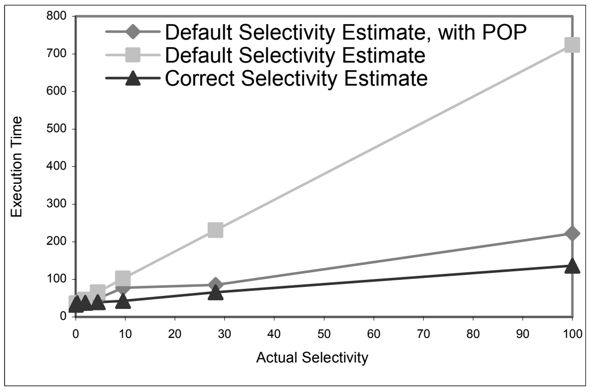

Figure 11: Robustness of TPC-H Q10 with POP

Figure 11 shows the execution time of TPC-H Q10 depending on the actual selectivity of the predicate on LINIETEM in three situations: (a) with POP and with parameter markers, such that the selectivity estimate for the predicate on LINEITEM used by the optimizer is a constant default value, (b) without POP and with parameter markers, again using a constant selectivity, and (c) without parameter markers, so that the selectivity estimate for LINEITEM is accurate, giving the QEP optimal with respect to the optimizer's model as a reference point. Parameter markers lead to highly suboptimal execution times without POP. POP is able to keep the execution time close to optimal across the whole range of selectivities. The optimal query goes through 5 different optimal plans as we vary the selectivity: from (CUSTOMER HSJN (LINEITEM NLJN ORDER) for the highly selective predicate on LINEITEM to (LINEITEM HSJN (CUSTOMER HSJN ORDER)), if that predicate is not selective. POP is only 2 times slower than the optimal plan in worst case, and in this case is about four times faster than the plan that the optimizer would actually use. In general, POP is within a factor of two of the optimal plan in response time, achieving speed-ups of almost an order of magnitude over the initial plan chosen by the optimizer, thus adding significant robustness to query optimization.

5.2 Risk and Opportunity Analysis

Having seen the overall benefit of POP on robustness and response time, we now study overhead of the individual flavors of checkpoints vs. the opportunities they provide (for re-optimizing the query). Out of the 5 flavors of POP in Table 1, we analyze LC, LCEM and ECB checkpoints along both these domains. The other two flavors– ECNC and ECDC – can be placed almost anywhere in the query plan, but can result in wastage of arbitrary amounts of work. We do not quantitatively analyze these two high-risk, high-opportunity flavors.

As mentioned before, we define the overhead of a checkpoint as its performance impact on a *dummy re-optimization* that does not change the QEP. We separately add LC, LCEM, and ECB checkpoints to plans as in Table 1, and measure the total execution times with and without re-optimization.

Risk Analysis

Since LCs are placed just above existing materializations, the only overhead is for context switching and re-invoking the optimizer. We explicitly disable hash-join for this experiment so that the optimizer generates lots of materialization points so that we can study the LC overhead extensively.

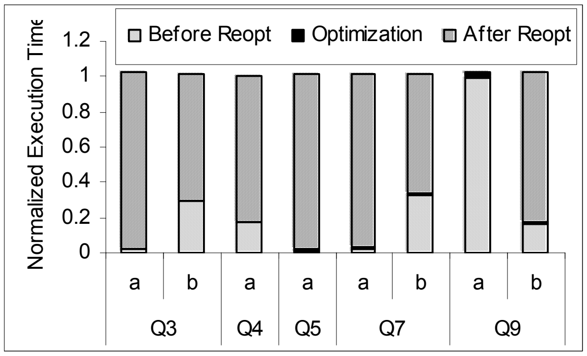

Figure 12: Normalized Execution time with LC re-optimization (1 is the execution time without re-optimization)

Figure 12 plots the normalized execution time for selected TPC-H queries. Each query is run once without triggering re-optimization and once or twice with re-optimization. The figure shows the execution time calling re-optimization normalized by the regular execution time for each query. For queries Q3, Q7, and Q9, the QEP had multiple checkpoints. The bars denoted by *a* and *b* in the figure show two separate executions of these queries with re-optimization triggered from different checkpoints in the same QEP. The left slanting region is the component of execution time before re-optimization, the right slanting region is the component after the re-optimization, and the small gap between them (almost invisible) is the time taken for the additional optimization at the checkpoint. The overhead that POP introduces is - negligible, about 2-3%.

The next experiment tries a more daring approach to re-optimization. We re-enable all joins, and proactively add LCEM check/materialization points on the outer of all NLJNs. We then run TPC-H queries without any re-optimization. Figure 13 plots the increased cost because of adding materialization points, normalized by the regular total execution time. The negligible overhead in Figure 13 clearly validates our hypothesis that if NLJN is picked over hash join, the outer is small enough to be aggressively materialized.

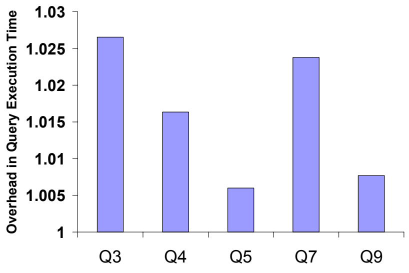

Figure 13: Cost of Lazy Checking with Eager Materialization

The last kind of checkpoint we analyze are ECBs, where we re-optimize even before the materialization is complete. The overhead here depends entirely on the work done till then, so we present it as part of opportunity analysis.

Opportunity Analysis

Our next experiment studies the frequency of opportunities for each flavor of re-optimization. We add LC, LCEM and ECB checkpoints to plans as in Table 1, but disable the actual re-optimization so that the entire query is executed and all checkpoints are encountered. Figure 14 shows a scatter-plot of the occurrence of these opportunities during query execution. Note that the ECB checkpoint opportunities are ranges given as dashed lines.

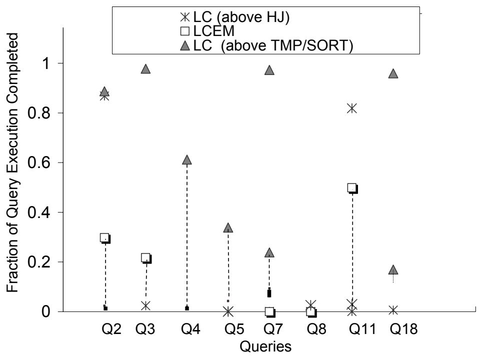

Figure 14: Opportunities for various kinds of checkpoints

The figure shows that even the low-risk LC and LCEM checkpoints occur quite regularly in query execution. Even granting that the ones occurring towards the end of query execution cannot help, we have one or two checkpoints in the middle of execution and one or two at the very beginning. When we add ECBs, a sizable fraction of the query duration (the dashed lines) is available for re-optimization. But the overhead is that we must redo the fraction of the query that is already completed – this ranges from 0 to about 60% in the figure. Many re-optimization opportunities are closely clustered together, especially in the early stages of query execution. This is because joins over the smaller tables typically separate materialization points.

6 POP in Action

In this section we apply POP to a real-world database and customer workload, using an 8-way PowerPC with 1.4 GHz Power4 CPUs, 32 GB RAM, 56 FASTT managed disks with a total of 36 GB net storage space. The database holds data of a department of motor vehicles (DMV), consisting of more than 30 tables and more than 100 indexes. The major tables of the database are the CAR and OWNER table storing 8 million respectively 6 million records. The overall size of the database is 7.4 GB. The CAR table contains major correlations, like a correlation between the columns MAKE, MODEL, COLOR, and MODEL, WEIGHT. There are also correlations when joining CAR and OWNER, like correlations between ZIP, MAKE and AGE, MAKE. We use 39 real-world queries obtained from the DMV to evaluate POP. The queries are very complex decision support queries, joining more than 10 tables in average.

Although the DMV workload did not use any parameter markers, it contained many other pitfalls that caused the optimizer to use wrong estimates: Many of the queries restrict several correlated columns, thus creating major cardinality estimation errors as the optimizer uses independence to combine the selectivities of these columns. Moreover, many of the queries uses complex predicates like substring comparisons, LIKE-predicates, and complex IN-lists and disjunctions. All of these predicates are additional sources of estimation errors. The largest cardinality estimation errors we have observed in the DMV queries exceed six orders of magnitude! For these complex real-world queries it is hardly possible for the optimizer to determine the right query plan based on its basic statistics and assumptions.

With POP no query runs longer than 5 minutes, whereas without POP the longest query took more than 20 minutes.

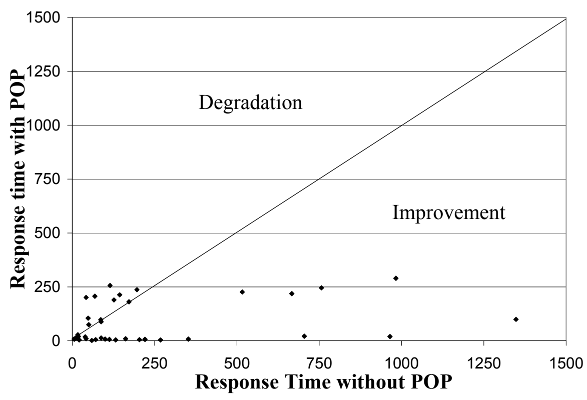

Figure 15: Scatter Plot of Response Times with and without POP on the DMV database

The scatter plot of the response times in Figure 15 shows that while 22 queries receive an improvement with POP, we notice a slight to moderate performance degradation in 17 queries. This performance degradation is due to two facts: In some circumstances the better cardinality information available to the optimizer during re-optimization resulted in the choice of a worse plan (!) because two estimation errors had canceled each other out during the initial run of the query, and no longer did so after re-optimization. In addition, we use a simplistic cost model for the cost for re-using an intermediate result, and this model leads to over-eager re-optimizations. Improving the optimizer's cost functions can solve the first problem. The second problem arises because we wanted to study re-optimizations extensively in this prototype and so used a generous cost model for reoptimization. So we are confident we can avoid this performance degradation when transferring this work into the product.

Figure 16 shows the speedup or regression experienced by each individual query. While POP reaches impressive speedups of almost two orders of magnitude, the maximum regression due to a wrong optimizer decision during re-optimization was a factor of 5.

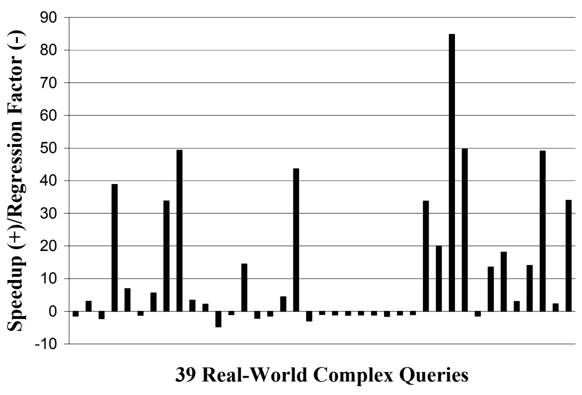

Figure 16: Speedup and Regression of each Query

Overall, POP adds significant robustness to the processing of the DMV queries, impressively speeding up several long-running queries.

7 Future Work

Synchronization in Parallel DBMSs

While implementing CHECK is relatively simple and straightforward for serial uni-processor environments, the cardinality counters it uses must be globally synchronized in symmetric multi-processor and shared nothing environments. Such synchronization can be a costly operation that can substantially delay query processing, and must be viewed as another risk of checkpointing in multi-processor environments. Alternatively, one can locally re-optimize a partial QEP executed on one node if the check range for this node alone is violated. Local checking in multi-processor environments would require that between global synchronization points (*exchange* operators in Volcano [GM93]) each node may change its plan, thus giving each node the chance to execute a different partial QEP.

Checking Opportunities

POP can be considered to be a more conservative mode of query execution, which is useful for complex ad-hoc queries or queries with parameter makers where statistics or the optimizer's assumptions are not considered to be reliable. In volatile environments like these, the optimizer can favor operators that enable further re-optimization opportunities over other operators. For example, sort-merge offers more chances than does hash-join for lazy re-optimization on either input if the check range is violated, so plans with merge joins with POP are more robust to misestimates of cardinalities than the corresponding hash-join plans. Interesting research issues arise here, e.g., when to use which of the five flavors of checkpoints.

Ensuring Termination

POP introduces the risk of iteratively re-optimizing a query many times. In order to ensure termination, heuristics have to be used, for instance by limiting the number of re-optimization attempts or by forcing to the use of intermediate results after several attempts in order to ensure that progress is indeed made. While our current prototype uses a crude heuristics, limiting the number of re-optimizations to 3, a more elaborate scheme deriving the number of permitted re-optimization attempts from the query complexity and the trust in the optimizer's cardinality estimates could be developed.

Considering Uncertainty during Re-optimization

POP learns actual cardinalities, which are compared to uncertain estimates during re-optimizations. The cost of the new join order in Figure 6, for instance, is measured without using actual knowledge about the cardinality of the join result, and compared to a cost computed using an actual cardinality. It might be useful to penalize plans without actual knowledge for the uncertainty in their cardinality estimates in order to avoid or at least alleviate the problem of wrong decisions based on partial knowledge und uncertainty. However, this significantly impacts the optimizer's cost model and requires more research.

Learning for the Future

POP only helps the query that is currently under execution. As a future extension it would be desirable to combine POP with techniques like LEO [SLM+01].

8 Conclusions

Progressive Optimization (POP) provides a flexible mechanism to build QEPs that are robust to optimizer mis-estimations, by inserting CHECK operators into a traditional QEP to test at execution time criteria under which the remainder of a QEP is still optimal. When these criteria are violated, the optimizer is invoked to progressively refine the plan, exploiting the additional information and results computed thus far. Different flavors of CHECK operators permit progressively refining both pipelined and non-pipelined plans and plans with various degrees of risk and opportunity. CHECK operators can also be used to re-optimize when parameters other than the cardinality are out of bounds, such as memory consumption, execution time, or even the overall system load. We have prototyped in a commercial DBMS the more conservative form of checkpoints that safeguard materialization points -- such as SORTs or TEMPs -- and the outer of each NLJN. Opportunities for basically risk-free checkpoints occur frequently in long-running real-world queries. Based on our experience building the prototype, an industrial-strength prototype of POP can be added easily to a DBMS. Experiments using our prototype have shown that POP drastically improves robustness of query execution, speeding-up complex queries by up to two orders of magnitude, while incurring only a negligible overhead of around 2-3% of the overall query execution time for queries not benefiting from POP.

9 References

AH00 R. Avnur and J. M. Hellerstein, Eddies: Continuously Adaptive Query Optimization, SIGMOD 2000

ARM89 R. Ahad, K.V.B. Rao, and D. McLeod, On Estimating the Cardinality of the Projection of a Database Relation, TODS 14(1), 1989

BC02 N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization, SIGMOD 2002

C+81 D. D. Chamberlin et al., Support for Repetitive Transactions and Ad-Hoc Query in System R, TODS 6(1), 1981.

CG94 R. Cole and G. Graefe. Optimization of Dynamic query evaluation plans, SIGMOD 1994.

Gel93 A. Van Gelder, Multiple Join Size Estimation by Virtual Domains, PODS 1993.

GM93 G. Graefe and W. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE, 1993

GW89 G. Graefe and K. Ward, Dynamic Query Evaluation Plans. SIGMOD 1989

HS93 P. Haas and A. Swami, Sampling-Based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics, IBM Research Report, 1993

HS02 A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. VLDB 2002.

IC91 Y.E.Ioannidis and S.Christodoulakis. Propagation of Errors in the Size of Join Results, SIGMOD 1991

Ives 02 Z. Ives, Efficient Query Processing for Data Integration, Ph.D thesis, University of Washington, 2002

KD98 N. Kabra and D. DeWitt, Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, SIGMOD 1998

MS02 S. Madden, M. Shah, J.M.Hellerstein, V. Raman. Continuously Adaptive Continuous Queries. SIGMOD 2000.

PIH+96 V. Poosala, et. al, Improved histograms for selectivity estimation of range predicates, SIGMOD 1996

PI97 V. Poosala and Y. Ioannidis, Selectivity Estimation without value independence, VLDB 1997

RAH03 V. Raman, A. Deshpande, and J. M. Hellerstein, Using State Modules for Adaptive Query Optimization. ICDE 2003

SAC+79 P.G. Selinger et al. Access Path Selection in a Relational DBMS. SIGMOD 1979

SS94 A. N. Swami and K. B. Schiefer, On the Estimation of Join Result Sizes, EDBT 1994

SWK96 M. Stonebraker, E. Wong and P. Kreps. The Design and Implementation of INGRES. TODS 1(3), 1976.

UFA98 T. Urhan, M.J. Franklin, and L. Amsaleg, Cost-based Query Scrambling for Initial Delays, SIGMOD 1998

SLM+01 M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO – DB2's Learning Optimizer, VLDB 2001

ZCL+00 M. Zaharioudakis et. al: Answering Complex SQL Queries Using ASTs. SIGMOD 2000