# Taking the Edge off Cardinality Estimation Errors using Incremental Execution

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

Cesar Galindo-Legaria
Microsoft
Redmond, WA, USA
cesarg@microsoft.com

**Abstract:** Query optimization is an essential ingredient for efficient query processing, as semantically equivalent execution alternatives can have vastly different runtime behavior. The query optimizer is largely driven by cardinality estimates when selecting execution alternatives. Unfortunately these estimates are largely inaccurate, in particular for complex predicates or skewed data.

We present an incremental execution framework to make the query optimizer more resilient to cardinality estimation errors. The framework computes the sensitivity of execution plans relative to cardinality estimation errors, and if necessary executes parts of the query to remove uncertainty. This technique avoids optimization decisions based upon gross misestimation, and makes query optimization (and thus processing) much more robust. We demonstrate the effectiveness of these techniques on large real-world and synthetic data sets.

## 1 Introduction

In most of today's database systems data access is hidden behind declarative query interfaces. The user (or a query generator) submits a query that describes the users intent, and the database system chooses the most efficient way to produce the requested data. This query optimization step, i.e., the transformation from a declarative query into an imperative execution plan, can have drastical impact on the performance of query processing. It is largely driven by a cost model that predicts how expensive certain operations are, and as such is used to find the most efficient execution alternatives for the given query. One of the most central components of the cost model is the cardinality estimation that predicts the result sizes of intermediate results.

What is interesting about cardinality estimation is that on one hand it has a very large impact on the selected execution plan, and on the other hand it tends to produce estimates that are very far off from time to time. Some people even claim that cardinality estimation is so brittle and unreliable that one should try to avoid relying on cardinality estimates at all (see for example [Cha09] for a discussion). We are a bit more optimistic and believe that for many queries and data sets the current cardinality estimation methods work reasonably well. Estimating the result cardinalities for the well known TPC-H benchmark for example is relatively easy, as both queries and data distributions are simple. Real-world data sets are more difficult for estimation purposes, but even their estimates are usually not that bad, at least for base tables. On the other hand, bad cardinality estimates that can be orders of magnitude off are an undeniable reality, either because the data is heavily skewed or because

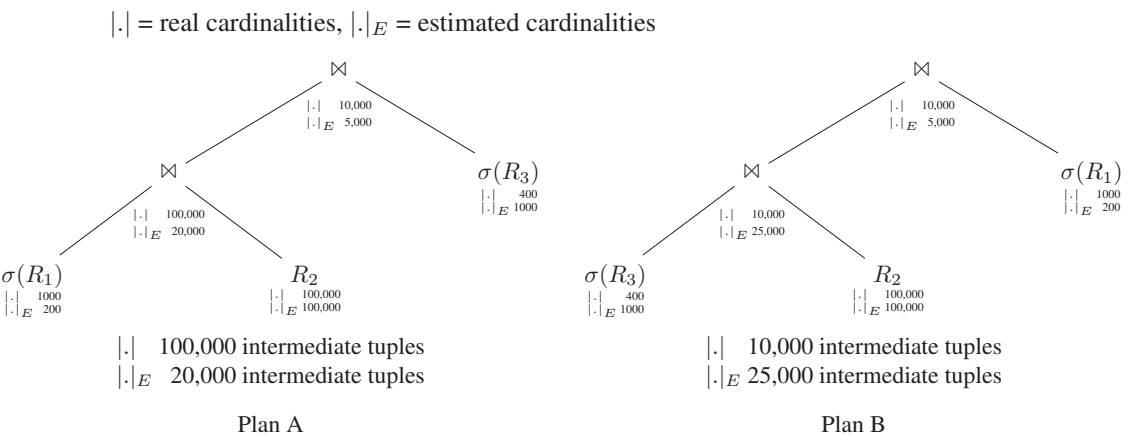



Figure 1: Impact of Cardinality Estimation Errors

the query predicates are complex and/or correlated. For example, the result cardinality of the following (admittedly constructed) SQL query over the TPC-H schema is quite hard to estimate:

```
select *
from lineitem,order
where l_orderkey=o_orderkey and
      log(l_extendedprice)>4 and
      log(o_totalprice-l_tax)>4
```

Such complex filter predicates are uncommon, but they do occur, and they are almost guaranteed to lead to poor cardinality estimates. Unfortunately the infrequently occurring bad estimates are much more noticeable than the more common good estimates, as they can lead to very poor execution plans. This is illustrated by an example in Figure 1. It shows two possible execution plans for a join query with three relations, annotated with real result cardinalities and estimated result cardinalities, where the estimates for two leaf nodes are somewhat off. When we look at the estimated cardinalities both plans seem to be very similar, with Plan A being slightly preferable due to the lower number of intermediate results. But when we look at the real number we see that in reality Plan A is much worse, producing 10 times the number of intermediate results of Plan B. Note that this mistake (preferring Plan A over Plan B) was not caused by a single estimation error, but only by combination of two unrelated errors. This is a quite common scenario, and implies that it is not sufficient to look at the accuracy of individual estimates, but that the whole execution query plan has to be taken into account.

The main idea of this work is to mitigate the effects of these cardinality estimation errors by using *incremental execution*: The query optimizer starts optimizing the query, and when it notices that cardinality estimation errors may lead to suboptimal execution plans it executes parts of the query to get the correct cardinalities. We give a formal description of this approach later, but roughly said the query optimizer executes the query fragments where the cardinality estimates are uncertain but important for the total plan choice. For the example

in Figure 1 this means that the optimizer might decide to execute $\sigma(R_1)$ or $\sigma(R_3)$ (but not both, as we will see in Section 3) to get the cardinalities right, and will then be able to pick the right plan. This incremental execution paradigm makes the optimization process much more robust against estimation errors, as estimation errors can be corrected during the optimization process now. Incremental execution induces some costs, of course, we therefore use incremental execution only when our algorithm decides that it is necessary for the given query.

The main contributions of this paper are as follows:

1. we present an algorithm that decides if incremental execution is necessary for a given query, and schedules the incremental executions as needed.
2. we show that this algorithm is guaranteed to lead to optimal execution plans under certain constraints, and is theoretically sound even if optimality cannot be guaranteed.
3. we present an error propagation framework that allows for deriving the error bounds necessary for deciding about incremental execution.
4. we demonstrate the validity of these techniques inside a commercial database system using large real-world and benchmark workloads.

The rest of the paper is structured as follows: First, we discuss related work in Section 2. Then, we introduce the incremental execution algorithms in Section 3, and show that they are theoretically sound. After that, we explain the error propagation framework necessary for incremental execution decisions in Section 4, followed by techniques to reduce the runtime costs of incremental execution. Finally, we give experiment results for real-world and benchmark workloads in Section 6. Conclusions are drawn in Section 7.

## 2 Related Work

Cost based query optimization has already been pioneered by System R [SAC+79], and accordingly cardinality estimation is a very well studied field. A comprehensive overview over the standard cardinality estimation techniques is given in [Ioa03], but these techniques usually do not consider the problem addressed in this paper, namely that cardinality estimates will be wrong at some point. Some histogram techniques acknowledge the fact that they are error prone and try to minimize the error, for example V-Optimal histograms [JKM+98] or some wavelet construction mechanisms like [GK05], but regarding the impact on the resulting execution plan they basically remain best-effort. A recent work tries to minimize the effect that cardinality-misestimations can have on query execution [MNS09], but it does not address the issue of recovering from large estimation errors. Their motivation is very similar to ours, but the approach is very different. They construct histograms in a way that minimizes the multiplicative error, which is very useful for the error computation discussed in Section 4, but they do not look at the issues caused by large estimation errors. Thus they delay the moment where misestimations will lead to bad plans, but when it happens they cannot cope with the estimation error. Therefore it makes sense to use their histograms in combination with our approach (as then the error estimates will be tight and low), as both approaches are really complementary. There has been some effort to cope with estimation errors within the optimizer, usually in the form of a feedback loop (see e.g., [SLMK01, LLZZ07]). The fundamental problem of these approaches is that

the query feedback is only available after the query was executed. The next query might profit from the new information, but the current query will still suffer from estimation errors. Furthermore some application scenarios have a lot of unique queries, which makes exploiting query feedback difficult.

A recent work that is similar in spirit to our incremental execution approach is the ROX optimizer [KBMvK09] for XQuery processing. It tries to address the cardinality estimation problem by relying on estimates only for the next step: When faced with a join ordering decision, it uses sampling to predict the join selectivity, executes the most promising join, materializes the result, and then optimizes again. As a consequence it always operates using exact cardinalities (as all inputs are materialized), the only uncertainty is the join selectivity (which is computed using sampling). The ROX optimizer fits nicely into the containing MonetDB framework, which materializes all intermediate results anyway, but in general it seems to be wasteful to materialize everything. The experiments in [KBMvK09] used a relatively small XML data set, but we found in our experiments that materializing all intermediate results causes a significant overhead when the intermediate results are larger than main memory (see Section 6). Furthermore the join ordering strategy proposed by ROX is overly greedy (being basically equivalent to [Feg98]), which can lead to suboptimal plans even without cardinality estimation errors.

While query optimizers traditionally have not done much to cope with cardinality estimation errors (besides improving the general accuracy, of course), there has been quite a lot of work on the runtime side. A good overview over these techniques is given in [DIR07]. These adaptive runtime techniques try to change the execution strategy according to the observed data distributions. The most extreme example are Eddies [AH00, TD03, RDH03], which – at least in principle – can adjust the data flow for each individual tuple. More conventional approaches [CG94, KD98, MRS+04] re-optimize parts of the query during query execution as they observe the cardinalities of intermediate results. The main problem there is deciding when to re-optimize. First, deciding at which point in the execution plan the cardinality should be checked (which usually requires materialization), and second, computing for which cardinalities the current execution plan should be changed. These are non-trivial issues, and the current adaptive runtime techniques have no satisfying solutions for them.

## 3 Using Incremental Execution

We now discuss how to incorporate incremental execution into query optimization, or more precisely, into plan construction. First, the name ”incremental execution” is not completely unambiguous, as there are multiple ways one could use an incremental execution paradigm. One extreme would be executing each query operator individually and then re-optimize after each step [KBMvK09]. But such a scheme seems wasteful, as incremental execution induces overhead, and, even more importantly, enforces serial execution of plan fragments. We therefore propose to use incremental execution only when needed, i.e., only to prevent plan construction mistakes caused by gross misestimates.

### 3.1 General Approach

We use the following incremental execution model: First, we construct the optimal execution plan using our cost model. (The plan optimality is relative to this cost model, of course). Then we identify plan fragments where cardinality estimation errors might have lead to wrong plan decisions higher up. Slightly simplified, we look for plan fragments where knowing the correct cardinality is important. We then execute the plan fragment, materializing the result, and thus getting the exact cardinality. If the new cardinality indicates that we have to choose a different plan we re-optimize, using the materialized result as available input. The advantage of this model is that we only execute plan fragments that we would have executed anyway, as they are parts of the (presumably) optimal plan. Furthermore we might decide not to execute any plan fragment if the cardinalities are not critical to the choice of plans.

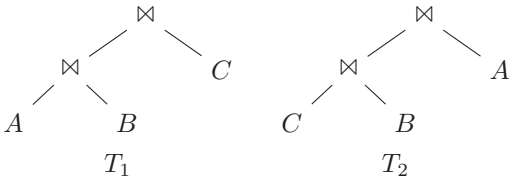
To incorporate incremental execution into query optimization we have to provide answers to two questions:

- given an optimal execution plan (relative to a cost model) and the cardinality estimation data, can we decide that the plan is indeed optimal?
- if not, which plan fragment should we execute to decide optimality afterwards?

As an illustrational example consider the following query fragment:

```
select *
from   (...)  A, (...)  B, (...)  C
where  A.x=B.x and B.y=C.y
```

Note that $A$, $B$ and $C$ could be anything from base relations to complex subqueries. When disregarding cross products, there are only two possible join trees for joining $A$, $B$ and $C$ (ignoring commutativity):




We now assume that $T_1$ was chosen as optimal plan (according to the cost model). This means that based upon the cardinality estimates, the costs of $T_1$ are lower than the costs of $T_2$. The interesting question is now, how much do the cardinalities have to change such that $T_2$ becomes cheaper than $T_1$. This is equivalent to computing the maximum cardinality estimation error for which $T_1$ is still guaranteed to be optimal.

The exact computations depend on the cost function. We use the cost function that minimizes the sizes of intermediate results here, but other cost functions could be used as well:

$$C(T) = \begin{cases} 0 & \text{if } T \text{ is a relation } R_i \\ |T| + C(T_1) + C(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

When using this cost function for $T_1$ and $T_2$, we notice that the optimality of $T_1$ does not depend on $|B|$. Due to the structure of the query $B$ must occur in the first join, and its cardinality does not affect the optimal relative order of $A$ and $C$. Therefore there is no point in using incremental execution for $B$, as its outcome would not change the constructed execution plan. The sizes $|A|$ and $|C|$ do have an impact on the plan. We stated that $T_1$ is supposed to be cheaper than $T_2$. Now assume that we underestimated $|A|$ by a factor $\alpha_A$ and overestimated $|C|$ by a factor $\alpha_C$. We denote the join selectivities with $f_{AB}$ and $f_{BC}$. Then, we can formulate the fact that $T_1$ is cheaper than $T_2$ as follows:

$$\begin{aligned} C(T_1) &\leq C(T_2) \\ f_{AB}(\alpha_A|A|)|B| &\leq f_{BC}|B|(\frac{1}{\alpha_C}|C|) \\ \alpha_A\alpha_C &\leq \frac{f_{BC}|C|}{f_{AB}|A|} \end{aligned}$$

In other words, as long as our estimation errors $\alpha_A$ and $\alpha_C$ are below the threshold $\frac{f_{BC}|C|}{f_{AB}|A|}$ we know that we picked the right join order and do not need incremental execution. Intuitively, this mimics the fact that large differences in relation sizes lead to clear join orders (the threshold is large), while small differences make it much easier to make mistakes (the threshold is small). One crucial part in this reasoning is the computation of estimation errors, of course, see Section 4 for details.

If the errors are above the threshold we might potentially make a mistake, and it therefore makes sense to use incremental execution. We have multiple plan fragments that could potentially be executed (in the example $A$ and $C$), and only one of these must be picked. In principle any of them could be chosen, and different selection strategies appear to be reasonable (e.g., minimizing costs or minimizing materialized results). In our experiments we found that a very good strategy seems to be choosing the plan fragment that has the maximum estimated error associated with it. In the example we would execute $A$ if $\alpha_A \geq \alpha_C$, and $C$ otherwise. By executing the largest error first we remove the biggest chunk of uncertainty from the optimization problem, and therefore expect to perform less incremental executions in total. We also experimented with executing the smallest intermediate results first (as these are cheap to materialize), but this performed worse in our experiments.

### 3.2 Algorithms

After the high-level discussion of the incremental execution idea we now discuss the concrete algorithms. The basic framework is shown in Figure 2. Given a query $Q$, it first optimizes the query using the cost model, then checks if some parts of the resulting execution plan $P$ warrant incremental execution. If not, it knows that $P$ is indeed the best plan and returns it. Otherwise it examines all incremental execution candidates and picks the candidate $P_C$ that has the largest estimation error $\alpha$. This plan fragment is then executed, its result is stored in a temporary table, the query is updated to use the temporary table instead of the plan fragment, and the query is optimized again (using the cardinalities

OPTIMIZEINCREMENTAL($Q$)
**while true**
  $P$=OPTIMIZE($Q$)
  $C$=FINDINCREMENTALEXECUTIONCANDIDATES($P$,$P$)
  **if** $C = \emptyset$
    **return** $P$
  $\hat{C} = \{P | P \in C \wedge \nexists P' \in C : P' \in P\}$
  $P_C = \arg\max_{P \in \hat{C}} \alpha(P)$
  execute $P_C$, store the result in $R$
  replace $P_C$ with $scan\ R$ in $Q$

Figure 2: Using incremental execution

derived from the incremental execution). Note that it is not necessary to re-optimize the full query again, all parts that are independent of $P_C$ can be reused directly. Further note that depending on the structure of the query the optimization time decreases exponentially with the number of operators contained in $P_C$, as these operators are removed from the search space. The repeated optimization is therefore not time critical in practice.

The main difficulty in using incremental execution is identifying plan fragments where the estimation errors can cause plan changes. The algorithm for finding candidates that can affect the join ordering is shown in Figure 3. It recursively traverses the execution plan until it reaches a join operator. For each join, it examines the "ancestors" of the join (i.e., the operators being executed after the join), and checks if would have been possible to execute them instead of the current join. This is illustrated in Figure 4. We assume that $A$, $B$, $C$, and $D$ are complex subqueries. The join $\bowtie_3$ (which joins $A$ and $B$) has two ancestors, $\bowtie_1$ and $\bowtie_2$. When examining these joins we see that $A$ could have been joined with $D$ instead, and $B$ with $C$. To be sure that $\bowtie_3$ was the right choice we must therefore first compute

$$\frac{f_{AD}|D|}{f_{AB}|B|} = \frac{90}{5} = 18$$

and compare it with

$$\alpha_B \alpha_D = 1.2 * 1.5 = 1.8.$$

Here the error was less than the threshold, so $A$ should indeed be joined with $B$ first. Then, as we could also join $B$ with $C$, we compute

$$\frac{f_{BC}|C|}{f_{AB}|A|} = \frac{20}{1} = 20$$

and compare it with

$$\alpha_A \alpha C = 10 * 2.5 = 25.$$

Here the error is larger, i.e., we are not sure that $\bowtie_3$ was indeed the right choice. Both $A$ and $C$ are candidates for incremental execution, we will execute $A$ first to remove the most uncertainty from the system. Similar computations are performed for the other join operators. The actual computations get a bit more complicated as the input of the other joins are no longer base expressions, but the general principle is the same.

FINDINCREMENTALEXECUTIONCANDIDATES($P$,$P_{root}$)
$C = \emptyset$
**for each** $P'$ input of $P$
  $C = C \cup$FINDINCREMENTALEXECUTIONCANDIDATES($P'$,$P_{root}$)
**if** $P = T_1 \bowtie T_2$
  $C_1 = \emptyset$, $C_2 = \emptyset$
  **for each** $P'$ ancestor of $P$ in $P_{root}$
    **if** $P' = T_1' \bowtie T_2' \wedge P \in T_1'$
      **if** $T_1$ could be joined with $T_2'$
        $C_1 = C_1 \cup \{T_2'\}$
      **if** $T_2$ could be joined with $T_2'$
        $C_2 = C_2 \cup \{T_2'\}$
    **if** $P' = T_1' \bowtie T_2' \wedge P \in T_2'$
      **if** $T_1$ could be joined with $T_1'$
        $C_1 = C_1 \cup \{T_1'\}$
      **if** $T_2$ could be joined with $T_1'$
        $C_2 = C_2 \cup \{T_1'\}$
  **for each** $T' \in C_1$
    **if** $\alpha(T_2)\alpha(T') > \frac{f_{T_1T'}|T'|}{f_{T_1T_2}|T_2|}$
      $C = C \cup \{T_2, T'\}$
  **for each** $T' \in C_2$
    **if** $\alpha(T_1)\alpha(T') > \frac{f_{T_2T'}|T'|}{f_{T_2T_1}|T_1|}$
      $C = C \cup \{T_1, T'\}$
**return** $C$

Figure 3: Identifying candidates

We only considered using incremental execution to be certain about join ordering, as this usually has the largest impact on the overall query performance, but similar computations could be added for other operators, too: The optimizer must check if the cardinality estimation error could lead to a plan change, and execute the relevant plan fragments if the cardinality has to be known exactly. Note that we do not assume that the query is join-only, i.e., other operators like selections or group-by can occur within the operator tree (in particular group-by statements are frequently the cause for severe estimation errors), but we currently only consider plan changes affecting the join order when deciding about incremental execution. These were the most critical parts of the queries in our experiments with real-world queries (see Section 6).

### 3.3 Theoretical Foundation

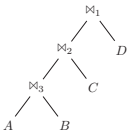
The candidate selection algorithm in the previous section uses some simplifying assumptions and some heuristical criteria, but it was derived from a solid theoretical basis. Now we will therefore explain the simplifying assumptions, and discuss the optimality guarantees that still hold even in the presence of these assumptions.

The most severe simplification used by the algorithm is that it disregards the estimation

query graph:


example sizes: $|A| = 20\ |B| = 100\ |C| = 200\ |D| = 100$
selectivities: $f_{AB} = 0.05\ f_{BC} = 0.1\ f_{AD} = 0.9$
errors: $\alpha_A = 10\ \alpha_B = 1.2\ \alpha_C = 2.5\ \alpha_D = 1.5$

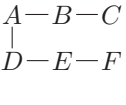
execution plan:




$C_i$ sets for $\bowtie_1$: $C_1 = \emptyset\ C_2 = \emptyset$
$C_i$ sets for $\bowtie_2$: $C_1 = \{D\}\ C_2 = \emptyset$
$C_i$ sets for $\bowtie_3$: $C_1 = \{D\}\ C_2 = \{C\}$

Figure 4: Example plan with computation results from Figure 3

query graph:




execution plan:

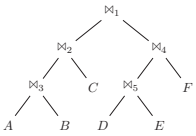



Figure 5: Bushy execution plan

error for the top-most join selectivities. In the example, the algorithm decides about the join of $A$ with $B$ or $D$ by comparing $\alpha_B\alpha_D$ with $\frac{f_{AD}|D|}{f_{AB}|B|}$. The comparison itself is theoretically justified, but it implicitly assumes that the values $f_{AD}$ and $f_{AB}$ are known exactly, while in reality the join selectivity will only be known approximately. As a consequence the algorithm could decide that the join order is correct and that no incremental execution is necessary, as the errors in join inputs are not severe enough, even though the error on the join selectivity itself caused a bad join order. This is unfortunate, but unavoidable. The error on the join selectivity can only be known after executing the join itself, and then it is too late to change the join order. This observation is also the reason our algorithm pretends that the top-most join selectivities are known exactly. Note that we only assume this for an individual join when considering plan alternatives. If the input of the join contains other join operators we take the estimation error of these join predicates into account (see Section 4). We might therefore detect an estimation error one operator too late, but in practice this does not seem to be a severe limitation.

The second simplification concerns the way the algorithm finds join alternatives. For each join operator the algorithm checks its ancestors to find alternative join partners, producing

$O(n)$ join alternatives (where $n$ is the number of base expressions). When only considering linear join trees (e.g., left-deep join trees) the resulting $C_i$ lists contain precisely the possible join alternatives. However when considering bushy join trees, there can be an exponential number of join alternatives, which are not all considered by the algorithm. This is illustrated in Figure 5: When examining the join $\bowtie_3$, the algorithm will determine that $A$ could be joined with $B$ and with $(D \bowtie_5 E) \bowtie_4 F$, even though there are more alternatives (like for example joining only with $D$). However this is mainly a theoretical concern, and does not not affect optimizations based upon data flow minimization like minimizing the sizes of intermediate results. This can be seen by symmetry arguments. First, the algorithm considers all joins contained in the execution plan, in particular also $\bowtie_4$ and $\bowtie_5$. If these contain uncertainties that warrant incremental execution it will execute them first, before considering executing $(D \bowtie_5 E) \bowtie_4 F$, as sub-trees are always considered before super-trees. If the algorithm does not use incremental execution we are certain about the relative order of $D$, $E$, and $F$, as we always consider the best plan under the given cost model. But then it must always be beneficial to perform $\bowtie_4$ and $\bowtie_5$ before $\bowtie_1$, at least concerning the data flow, as otherwise the optimizer would have pulled the joins up. It is therefore usually not necessary to consider the exponential number of alternatives.

These two limitations are concessions to the practical usage of incremental execution, as lifting them would require unreasonably large processing costs (in particular for the first limitation). The second limitation is largely harmless, the first one is a bit unfortunate, but does not seem to be easily avoidable. However, we will now show that the fundamental construction of the incremental execution algorithm is sound, in the sense that it can guarantee optimal execution plans for certain classes of queries. This can be seen by considering the IK/KBZ family of algorithms [IK84, KBZ86]. These algorithms construct the optimal linear join trees for acyclic join queries with ASI cost functions (e.g., minimizing the sizes of intermediate results). They roughly work in two steps, where the first constructs the precedence graph that describes which join has to be performed before another join becomes possible (to avoid cross products). The second step sorts the join operators by rank (i.e., the perceived "benefit" of the join) and merges them together according to the rank and the restrictions imposed by the precedence graph.

As the IK/KBZ algorithms construct linear trees, it is sufficient to identify an execution plan with a sequence of relations (joining from left to right). Then, the rank function of a sequence $S$ is defined as

$$rank(S) = \frac{T(S) - 1}{C(S)}$$

where $T(S)$ is the result size change (i.e., the selectivity relative to the first relation) and $C(S)$ are the costs per input tuple. For a relation $R_j$ that is ranked relative to a precedence root $R_i$ this results in a rank of

$$rank(R_i R_j) = \frac{|R_j| f_{R_i R_j} - 1}{|R_j| f_{R_i R_j}} = 1 - \frac{1}{|R_j| f_{R_i R_j}}.$$

Using this formulation it is clear that if one relation $R_j$ has a lower rank than a relation $R'_j$, and the cardinality estimation errors $\alpha_{R_j}, \alpha_{R'_j}$ are bound by

$$\alpha_{R_j}\alpha_{R'_j} \leq \frac{f_{R_iR_j}|R_j|}{f_{R_iR'_j}|R'_j|}$$

then the relative rank of $R_j$ and $R'_j$ remains the same. Or, in other words, using our incremental execution algorithm guarantees finding the optimal execution plan in this case.

For general queries we cannot guarantee optimality, but as illustrated above there is strong evidence that the algorithm is sound and that it will find the critical join alternatives. For linear execution plans we could even derive an optimality bound, as the worst case depends purely on the limitations of join selectivity estimation errors.

# 4 Error Propagation

The incremental execution framework from Section 3 needs to know the error, i.e., the estimation uncertainty, associated with each cardinality estimation. In most database systems this information is not readily available, we therefore now discuss how we can estimate and propagate error bounds within execution plans.

At a first glance computing the estimation error seems to be nearly as difficult as the estimation itself; at least the two problems are closely related. However in practice the error estimation is a much more good-natured problem, as the error estimate does not have to be as accurate as the cardinality estimate. Misestimations of the error do have consequences, of course. Overestimating the error can lead to unnecessary incremental executions, while underestimating the error can lead to suboptimal executions plans. But overall a bad error estimate usually has much less severe consequences than a bad cardinality estimate. Furthermore it is much easier to learn the typical estimation error from observed query executions than to learn the cardinalities themselves (in particular since the error bound does not have to be tight). Therefore it is relatively easy to integrate the error propagation techniques explained below into an existing database system.

Before discussing the error propagation framework we first have to define the error metric that we use. The incremental execution framework from Section 3 needs to know the factor $\alpha$ that gives the maximum derivation from the real cardinality. Given an algebraic expression $E$ we therefore define the error function $\alpha(E)$ as follows:

$$\begin{aligned} |E| &:= \text{result cardinality of } E \\ |E|_E &:= \text{estimated result cardinality of } E \\ \alpha(E) &:= \frac{\max(|E|, |E|_E)}{\min(|E|, |E|_E)} \end{aligned}$$

Thus an error of $\alpha(E) = 2$ means that the estimate is at most twice as large as the result cardinality (or at least half the size of the result cardinality). Note that we implicitly assume that $|E| \geq 1$ and $|E|_E \geq 1$, i.e., we assume that the query result is non-empty. Empty queries are a special case that has to be handled (and detected) efficiently by the runtime system, the query optimizer always assumes non-empty results.

$$\begin{aligned}
\alpha_S(R_i) &= 1 \\
\alpha_S(\sigma_p(E)) &= \alpha_S(E)\alpha_S(p) \\
\alpha_S(E_1 \times E_2) &= \alpha_S(E_1)\alpha_S(E_2) \\
\alpha_S(E_1 \bowtie_p E_2) &= \alpha_S(E_1)\alpha_S(E_2)\alpha_S(p) \\
\alpha_S(\Gamma_{agg}(E)) &= \alpha_S(E)\alpha_S(agg) \\
\alpha_S(\rho a \to b(E)) &= \alpha_S(E) \\
\ldots &\quad \ldots
\end{aligned}$$

Figure 6: Error propagation within an operator tree

Using this error metric we now have to derive error bounds for algebraic expressions. We distinguish two kinds of error bounds, first the *maximum* error $\alpha_M$ that can be derived from schema information, and second the *structural* error $\alpha_S$ that stems from predicate analysis and error propagation within the the operator tree. Both metrics provide bounds for the real error, thus

$$\alpha(E) = \min(\alpha_M(E), \alpha_S(E))$$

The maximum error is derived from the known cardinality bounds that are maintained in most query optimizers anyway. Using the upper cardinality bound $|E|_E^{max}$ and the lower cardinality bound $|E|_E^{min}$ we can define the maximum error as

$$\alpha_M(E) := \max(\frac{|E|_E}{|E|_E^{min}}, \frac{|E|_E^{max}}{|E|_E}).$$

This bound is a hard error bound, but in most cases it is not very useful as $|E|_E^{max}$ tends to be very large (it assumes that all predicates always match etc.). Still, it is a useful bound, as in some cases it is tight. The simplest example are (sub-)queries of the form

**select count**(*) **from** ... **where** ...

Here we know that the result cardinality is exactly one tuple. Other examples include constraints on key attributes, group by queries with grouping on attributes with known domains (e.g., keys), etc. In these cases the maximum error that is purely derived from schema information will give useful bounds. Note that $\alpha_M$ is really a bound, not a function that distinguishes exactly known cardinalities from uncertain cardinality. In particular $\alpha_M(E)$ can be $> 1$ even in the examples mentioned above due to additional predicates, but it tends to be a tight bound in these cases.

For general queries the maximum error will be very loose, we will have to examine the algebraic expressions themselves to get tighter bounds. The basic principles of error propagation are shown in Figure 6. For base relations the error is 1, as the cardinality is usually known exactly. Selections introduce an error, therefore the total error is the error of the input times the error caused by the selection predicate. We will discuss predicates below. Cross products are simple, as the two input errors can simply be multiplied here (this follows naturally from the definition of the Cartesian product). A join is a cross product followed by a selection, the error propagates accordingly. Group by operators ($\Gamma$) are a bit

special. In principle they are similar to selections, but in practice it is often hard to predict the number of resulting groups. This will be discussed further below. Finally, there is a number of operators that do not affect the cardinality at all (e.g., $\rho$), these just propagate the errors up. As we can see, the main problem here is computing the error induced by individual predicates.

For simple predicates of the form $x = 7$ or $x \geq 4$ this error can be computed relatively easily. Database systems usually maintain statistical synopses like histograms about their data, and we can give error bounds for these: When constructing a histogram bucket there will be minimum frequency $f_{min}$ of values within the bucket, a maximum frequency $f_{max}$ and the average frequency $f_{avg}$ that is usually used when estimating the result cardinality. The maximum error induced by this bucket is therefore

$$\max(\frac{f_{avg}}{f_{min}}, \frac{f_{max}}{f_{avg}}).$$

During histogram construction we can derive this value for each bucket, and then remember the maximum over all buckets as the maximum error induced by this histogram.

This gives a hard bound, but might be too large in practice. As we maintain the maximum error, this value is very susceptible to outliers. Ideally we would use histograms that minimize the maximum error [MNS09], or, if we have to cope with existing systems that cannot easily change their histogram implementation, we would not take the maximum error but the 95% quantile or use some similar dampening technique. The later is not as satisfying from a theoretical point of view, as we might now miss incremental execution candidates, but it can greatly reduce the number of incremental executions, as most of the time the error will not be that large.

For more complex predicates we can start by combining estimated errors for simple predicates, which works reasonably well for $\wedge$ and $\vee$, but at some point we must fall back to guessing. This is similar to selectivity estimation, which will also have to fall back to guessing at some point. Interestingly this means that in the cases where we have to guess, we basically know that the estimation error will be high, as the selectivity estimation guesses, too. Similar for *group by* operations, either we have domain information available, in which cases we can derive an error bound (even though it will frequently be large [CCMN00]), or both the error estimate and the selectivity estimation will have to be guesses. In experiments we found that guessing the error is much easier, and in fact reasonable error bounds for complex predicates can be derived by examining available query feedback. The best way to address complex predicates is therefore probably a statistics warehouse based upon query feedback similar to [MMK+05].

## 5 Reducing the Runtime Costs

Incremental execution is a very useful technique, as it limits the effects of estimation errors, but it comes with a cost. Even though the experiments in Section 6 show that the incremental execution frameworks materializes only a few intermediate results, the theoretical worst case would be that (nearly) every operator materializes all of its inputs, roughly increasing the execution time by factor of two. In this section we therefore present

techniques to reduce the runtime costs of incremental execution. Note though that these techniques are strictly optional. Our experiments show that even a database system without specific runtime support can greatly benefit from incremental execution.

One very nice property of incremental execution is that the mechanism only executes plan fragments that would have been executed anyway. This means that even in the worst case, where incremental execution does not correct a single mistake resulting from estimation errors, the overhead stems only from materialization. On the other hand, materialization costs can be quite high, in particular when looking at pipelining plans: Most database systems try to avoid materializing and copying intermediate results as much as possible, passing data between operators in a pipelined fashion [Lor74]. These system distinguish pipelining operators, i.e., operators that simply pass their input data along, and pipeline breaks, i.e., operators that explicitly materialize and copy data. For pipeline breakers the additional costs of incremental execution are not that high, as they materialize anyway, but for pipelining operators materializing their input can add a noticeable overhead.

Fortunately this problem is not too severe in practice. As mentioned in Section 3, we concentrate on using incremental execution for join operators, and most join implementations are pipeline breakers anyway. We will discuss techniques for speeding up incremental execution in the presence of joins below. Pipelining operators are usually much more light-weight, the most prominent example is the selection operator. Here, incremental execution could potentially be expensive, but in fact it would never be used. As discussed in Section 3, incremental execution is only triggered if the cardinality of the intermediate result could affect the plan choice. This is not the case for selections, not even in the presence of expensive predicates [HS93]. Assuming a selection $\sigma_i$ has a selectivity $s_i$ and causes $c_i$ costs per input tuple, we observe that

$$\begin{array}{rrcl} & C(\sigma_2(\sigma_1(R))) & \leq & C(\sigma_1(\sigma_2(R)) \\ \Leftrightarrow & c_1|R| + c_2 s_1|R| & \leq & c_2|R| + c_1 s_2|R| \\ \Leftrightarrow & c_2 s_1 - c_2 & \leq & c_1 s_2 - c_1 \\ \Leftrightarrow & \frac{s_1-1}{c_1} & \leq & \frac{s_2-1}{c_2} \end{array}$$

Or, to phrase it differently, the selections have to be sorted by $\frac{s_i-1}{c_i}$, independent of the input cardinality of the selection operator. Accordingly, incremental execution is not necessary for placing selections. The same is probably true for most other pipelining operators, as these tend to be decreasing unary operators. One notable exception is the nested loop join, we will discuss this below.

Pipeline breakers, in particular joins, are more complex and are usually candidates for triggering incremental execution on their input. Now the key observation here is that by their very nature, pipeline breakers are particularly well suited for incremental execution! As these operators materialize data anyway, we can use this to get incremental execution more or less for free. This is illustrated in Figure 7. When the query optimizer needs to know the input cardinality of a hash join for its plan decision, it conceptually breaks the execution plan into two parts (marked with *incr. break*). The first part executes the input plan and materializes the result (and potentially triggers a re-optimization step). The second part takes this materialized result, builds a hash table from it, and then probes the hash table with the data from the second input expression (shown on the left hand side

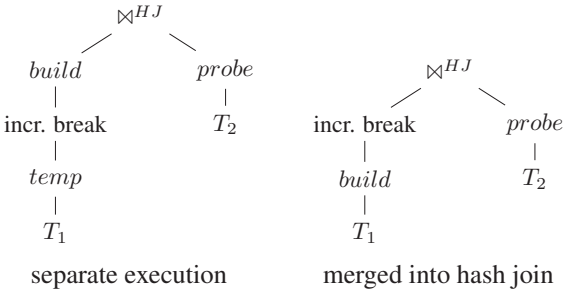



Figure 7: Merging Incremental Execution into Hash Joins

of Figure 7). Note that if the original cardinality estimate turns out to be accurate, the runtime system executes two materializing operators next to each other, namely *temp* and *build*. Therefore, one can improve the runtime performance by using only one of them, and directly materialize into the hash table (shown on the right hand side of Figure 7). If the optimizer sticks to the original plan after incremental execution this technique reduces the overhead of incremental execution to nearly zero, as again we only execute steps we would have executed anyway. If the optimizer does change the plan we have used a slightly more complex materialization operator than necessary, but in this case the benefits of the better plan far outweighs the costs of the more complex materialization.

Admittedly merging incremental execution into regular operator processing is not always as simple as the the example from Figure 7. For example materializing the *probe* side is more problematic, as the operator does not plan to materialize it. For symmetric join operators like the Grace Hash Join this is not an issue; in general an easy option would be to swap the roles of probe and build if both are reasonably small. But even if one indeed adds a materialization phase, it should prepare (e.g., partition) the data to help the subsequent operator. This strategy depends a lot on implementation details of the operators involved, but we found that most pipeline breakers can materialize any of their inputs in a way that helps subsequent processing. The same is true for nested loop joins, even though they are not pipeline breakers. Materializing the left side (i.e., the outer side) leads to block nested loop joins, which greatly reduce the number of passes over the inner side. Materializing the inner side can greatly reduce the costs if the inner side contains a complex execution plan.

In general, if the optimizer decides to trigger incremental execution, the runtime system should try to make use of this execution step. Note that once we materialize an intermediate result, we not only know its cardinality, but we have seen all the data. This allows for passing domain information throughout the execution plan, which can greatly reduce execution times. Therefore, if we decide to materialize an intermediate result, we should always build bitmap filters over join attributes that are used later on, as we get these bitmap filters nearly for free.

Overall incremental execution can be made quite cheap by exploiting the natural characteristics of the operators involved. The only disadvantage is that this requires some (minor) changes to the underlying runtime system, which is not always easy in commercial database systems. In the experiments in Section 6 we therefore measured both the behavior without

| | comp. [s] | exec. [s] | total [s] |
|---|---|---|---|
| out-of-the-box optimizer | 151.4 | 2437.5 | 2588.9 |
| bottom-up join ordering | 19.7 | 3508.0 | 3527.7 |
| with incremental execution | 1255.8* | 515.5* | 1771.3 |

* no clear distinction between compilation and execution

Figure 8: Query Processing for 40 Real-World Queries

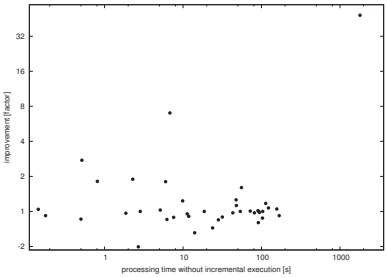



Figure 9: Effect of Incremental Execution on Individual Queries from Figure 8

explicit runtime support and the benefit of these runtime techniques.

# 6 Evaluation

We studied the effects of incremental execution on several large data sets. For the database system we used a development version of SQL Server, in which we integrated our techniques. To avoid a potential bias by the query optimizer, which out-of-the-box does not always use exhaustive search and therefore might produce plan differences due to timeouts, we implemented an exhaustive bottom-up join-ordering strategy [MN08] as basis for incremental execution. Note that the two optimizer approaches are not directly comparable. The bottom-up search guarantees that no join order is ignored just because of heuristical time reductions (which is important for the incremental execution experiments), but it misses some optimizations from the standard optimizer, in particular group-by optimizations. For completeness we always give results for both optimizers, but use the bottom-up construction to study the effects of incremental execution. Note that unless otherwise noted the results are without the runtime techniques from Section 5, i.e., they are purely query optimization effects.

For each data set and each of the approaches we ran a workload of queries and measured compilation time and execution time. If a query took more than half an hour (1800s) to execute we aborted it and counted the execution time as 1800s (this happened for the non-incremental algorithms). All experiments were conducted on a HP Compaq dc7900 with 8GB main memory and an Intel Core 2 Quad Q9400 CPU running Windows Server Enterprise.

## 6.1 Real-World Data

As a first data set we used data and queries provided by a customer where we knew that cardinality estimation had some difficulties. The data set is about 100GB in size (including indexes) and the 40 queries perform data-warehouse queries with reasonably complex predicates.

| | comp. [s] | exec. [s] | total [s] |
|---|---|---|---|
| out-of-the-box optimizer | 98.3 | 3418.9 | 3517.3 |
| bottom-up join ordering | 19.2 | 2140.1 | 2159.4 |
| with incremental execution | 386.6* | 1593.8* | 1980.5 |

* no clear distinction between compilation and execution

Figure 10: Query Processing for 99 TPC-DS Queries (scale factor 10GB)

We studied the query processing for all three approaches, the out-of-the-box optimizer, the bottom-up join ordering approach, and bottom-up join ordering including incremental execution. The results are shown in Figure 8. We notice two things: First, the bottom-up join ordering performs significantly worse than the out-of-the-box optimizer for this data set. This is caused by a combination of missing group-by movement and cardinality misestimation, which causes the bottom-up approach to choose a very poor execution plan. The second observation is that bottom-up join ordering with incremental execution performs much better than both, reducing query processing significantly, as it uses incremental execution to correct the problems of the non-incremental bottom-up optimizer. Note that there is no good notion of compilation time vs. execution time for incremental execution, we counted everything up to the final plan generation as compilation time and the final execution as execution time.

When looking at individual queries it is interesting to see that for most queries incremental execution has no effect all, but for some queries where the cardinality estimation has made a serious mistake it drastically improves query processing. This is shown in Figure 9: The $x$-axis is the query processing time of the bottom-up join ordering without incremental execution, and the $y$-axis shows the change by incremental execution. A value of $\pm 1$ means that both approaches need the same query processing time, $+2$ means incremental execution improves query processing by a factor of 2, $-2$ means a degradation by a factor of 2. Note that the $y$-axis is on a logarithmic scale, which means that most queries are quite close to 1, i.e., incremental execution has no effect. However for some queries it improves query processing, in particular for a query that would have been very expensive otherwise. This matches the expectation we have of incremental execution: It is a tool to prevent gross mistakes caused by misestimations, which is exactly what happens in this data set.

Note that we do not just improve some outliers. Preventing outliers is the whole point of incremental execution! Customers do not really notice when most of their queries speed up by 10%. But when a single query slows down by a factor of 10 they notice immediately. Incremental execution allows us to mitigate these outliers caused by estimation errors.

Another interesting question is how often incremental execution is triggered. After all the query optimizer will only use incremental execution if some cardinalities were critical for plan generation choices. For this data set, the optimizer triggered 43 incremental executions in 32 out of the 40 queries. Which means that while many of these queries needed some incremental execution to clarify uncertainties, in most queries it was sufficient to execute one small part of the query to be certain about the correct join order.

### 6.2 TPC-DS

The customer data set was one of our original motivations for looking at incremental execution, but it has the disadvantage of not being publicly available. We therefore also studied some benchmarks to get results that are more easily reproducible. Unfortunately

| build side | | |
|---|---|---|
| | no plan change | plan change |
| regular spool | 1.2% | 1.2% |
| hash table spool | 0% | <0.1% |
| probe side | | |
| | no plan change | plan change |
| regular spool | 7.9% | 7.9% |
| hash table spool | <0.1% | 0.5% |

Figure 11: Overhead of forced Incremental Execution for a single Join

most synthetic benchmark data sets tend to be overly simplistic, with uniform value distributions, independence, etc., which does not exhibit some of the cardinality estimation problems visible for real-world data. One notable exception is TPC-DS [NP06], which contains data skew (but still no correlations between attributes), and more complex queries. This causes some challenges for cardinality estimation, although it is still much easier than for the real-world data set.

The results for the 10GB scale factor are shown in Figure 10. (Note that this is not an official benchmark result and the number are from a development version of SQL Server). Again, incremental execution improves query processing, but the gains are more minor, as the original cardinality estimates were already quite good. Furthermore we noticed that we use incremental execution much less than for the real-world data set, simply because it is not necessary in most cases. Incremental execution was triggered in 37 out of the 99 queries, executing 91 query fragments in total. As incremental execution was used much less than for the other data set, we manually checked the 10 most expensive queries where incremental execution was not used to make sure that no opportunity for incremental execution was missed, and indeed that seems to be the case. This is encouraging, as it means that the incremental execution mechanism adapts to the necessity of incremental execution.

For the TPC-DS data set we also tried using a ROX-style optimizer [KBMvK09] that executes one operator at a time and then greedily re-optimizes after each execution. This gave very disappointing results, with an total execution time of over 5,840s and over 250GB of intermediate results materialized on disk due to poor plan choices caused by the greedy algorithm. We did not pursue this any further for the other data sets, as it seems to be clear that the greedy ROX optimization strategy is too simple for many of the more complex queries.

### 6.3 Runtime Improvements

So far we intentionally showed results without using any of the runtime techniques from Section 5, as commercial database systems currently do not offer specific support for incremental execution. And indeed incremental execution performs very well, even without runtime support. However, we implemented a prototype of our techniques and used it to study the improvements for TPC-DS queries.

As an initial micro-benchmark, consider the join from TPC-DS Query 1 between *time_dim* and *store_sales*. One input of the join is relatively small, and will be used as *build* side, and

the other is reasonably large. In the context of a larger query, both might be candidates for incremental execution. We therefore study the overhead of incremental execution relative to regular execution along multiple dimensions: First, we measure the overhead of incremental execution when using regular *spool* operators versus the hash table spooling from Section 5. Second, we differentiate between the build or the probe side. And finally, we study the costs when we change the plan (i.e., cannot reuse the hash table and just use it as spool). The results are shown in Figure 11. Two things are noticeable. First, incremental execution is not that expensive here, even with regular spool operators, as the intermediate result can be materialized into main memory. Second, materializing directly into the hash table greatly improves performance, and nearly completely removes the overhead of incremental execution, even disregarding the potential benefits of better estimates.

For the complete TPC-DS query suite the effect is a bit more difficult to characterize fairly. The runtime techniques help a lot, but some of these gains stem from implementation changes, and even more from additional benefits like the piggy-back bitmap filter construction. By comparing query execution times we estimated that ignoring the gains (better plans, more filters, etc.), the overall overhead of incremental execution is reduced to about 1% for the 99 TPC-DS queries when using the techniques from Section 5. This is a very modest price for robustness regarding estimation errors, as this 1% overhead can frequently prevent outliers of a factor of 10 or more, which means that overall incremental execution speeds up query processing quite a lot.

## 7 Conclusion

For complex queries cardinality estimation errors are nearly unavoidable, and can lead to very poor executions plans. We propose using incremental execution to address the cases where cardinality estimation cannot reach the accuracy required for optimization purposes. By identifying plan alternatives and their sensitivity to estimation errors we can limit incremental execution to the cases where it is really necessary. Our experiments on large real-world and benchmark workloads showed that our incremental execution techniques can greatly reduce the effects of cardinality estimation errors, and thus lead to more robust query processing.

Future work should include integrating a feedback loop into our incremental execution framework similar to [SLMK01, LLZZ07], as the materialized intermediate results could provide a lot of information about cardinalities, error rates, value distributions etc., which would be useful for the query optimization but are currently discarded.

## References


[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*, pages 261–272, 2000.

[CCMN00] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, pages 268–279, 2000.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of Dynamic Query Evaluation Plans. In *SIGMOD Conference*, pages 150–160, 1994.

[Cha09] Surajit Chaudhuri. Query optimizers: time to rethink the contract? In *SIGMOD Conference*, pages 961–968, 2009.

[DIR07] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[Feg98] Leonidas Fegaras. A New Heuristic for Optimizing Large Queries. In *DEXA*, pages 726–735, 1998.

[GK05] Minos N. Garofalakis and Amit Kumar. Wavelet synopses for general error metrics. *ACM Trans. Database Syst.*, 30(4):888–928, 2005.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *SIGMOD Conference*, pages 267–276, 1993.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[Ioa03] Yannis E. Ioannidis. The History of Histograms (abridged). In *VLDB*, pages 19–30, 2003.

[JKM+98] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal Histograms with Quality Guarantees. In *VLDB*, pages 275–286, 1998.

[KBMvK09] Riham Abdel Kader, Peter A. Boncz, Stefan Manegold, and Maurice van Keulen. ROX: run-time optimization of XQueries. In *SIGMOD Conference*, pages 615–626, 2009.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *VLDB*, pages 128–137, 1986.

[KD98] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD Conference*, pages 106–117, 1998.

[LLZZ07] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality estimation using sample views with quality assurance. In *SIGMOD Conference*, pages 175–186, 2007.

[Lor74] Raymond A. Lorie. XRM - An Extended (N-ary) Relational Memory. *IBM Research Report*, G320-2096, 1974.

[MMK+05] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. Consistently Estimating the Selectivity of Conjuncts of Predicates. In *VLDB*, pages 373–384, 2005.

[MN08] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD Conference*, pages 539–552, 2008.

[MNS09] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009.

[MRS+04] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*, pages 659–670, 2004.

[NP06] Raghunath Othayoth Nambiar and Meikel Poess. The Making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using State Modules for Adaptive Query Processing. In *ICDE*, page 353, 2003.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*, pages 23–34, 1979.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[TD03] Feng Tian and David J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *VLDB*, pages 333–344, 2003.