Non-Invasive Progressive Optimization for In-Memory Databases

Steffen Zeuch
Humboldt University of Berlin
zeuchste@informatik.hu-berlin.de

Holger Pirk
MIT CSAIL
holger@csail.mit.edu

Johann-Christoph Freytag
Humboldt University of Berlin
freytag@informatik.hu-berlin.de

ABSTRACT

Progressive optimization introduces robustness for database workloads against wrong estimates, skewed data, correlated attributes, or outdated statistics. Previous work focuses on cardinality estimates and rely on expensive counting methods as well as complex learning algorithms.

In this paper, we utilize performance counters to drive progressive optimization during query execution. The main advantages are that performance counters introduce virtually no costs on modern CPUs and their usage enables a non-invasive monitoring. We present fine-grained cost models to detect differences between estimates and actual costs which enables us to kick-start reoptimization. Based on our cost models, we implement an optimization approach that estimates the individual selectivities of a multi-selection query efficiently. Furthermore, we are able to learn properties like sortedness, skew, or correlation during run-time.

In our evaluation we show, that the overhead of our approach is negligible, while performance improvements are convincing. Using progressive optimization, we improve runtime up to a factor of three compared to average run-times and up to a factor of 4,5 compared to worst case run-times. As a result, we avoid costly operator execution orders and; thus, making query execution highly robust.

1. INTRODUCTION

The migration of databases from disk to faster memories such as RAM, Flash, or NVRAM fundamentally changes the cost balance in analytical data management systems: where disk-based system performance was largely dominated by disk bandwidth and latency, in-memory analytics systems have to consider CPU efficiency as a major contributor to performance. This requires a careful (re-)investigation of various design decisions underlying "classic" relational DBMSs with respect to these new considerations. This re-investigation has been done for many components of the classic analytical database design such as processing [4, 12], indexing [10, 11], and compression [18]. However, most of these techniques were developed to address hardware-specific cost factors such as cache thrashing, misprediction penalties, and synchronization costs. On the other hand, many of these techniques use new hardware features such as SIMD-instructions, transactional memory, or deep memory hierarchies in order to overcome hardware challenges like the *memory wall*.

In this paper, we apply this pattern to the idea of progressive optimization [14]. With progressive optimization, a physical query plan is adapted to the characteristics of the data subset that is currently processed. Following previous work, our approach is based on monitoring and analysis. However, unlike previous work, our approach has virtually no CPU costs by making extensive use of a handy feature of modern CPUs: the *Performance Monitoring Unit*. This unit allows the counting of performance-related events such as retired instructions, cache-misses, and branch mispredictions. By comparing the obtained numbers with those of fine-grained cost models at run-time, we might detect differences of the estimated from the actual costs, thus possibly kick-starting a re-optimization process. In fact, our approach effectively renders high quality decisions at query compilation time unnecessary because it provides better and more adaptive information at run-time. In addition to low CPU-overhead, such non-invasive monitoring extends the applicability of progressive optimization to cases when instrumentation is not an option such as binary UDFs or calls to external libraries. These benefits, however, hinge on the availability of appropriately accurate cost models. Consequently, our specific contributions include

- an unified cost model for memory accesses as well as branch misprediction costs in modern CPUs,
- an estimation component that derives data-specific characteristics such as selectivities and domains from performance event counters using non-linear optimization,
- a run-time execution component which balances the trade-off between the quality of the estimation and the required optimization time.

To illustrate the importance of avoiding bad plans when evaluating analytical queries on memory-resident data, we compared the cost of the worst and the best physical plan for Query 6 of the TPC-H benchmark:

Proceedings of the VLDB Endowment, Vol. 9, No. 14

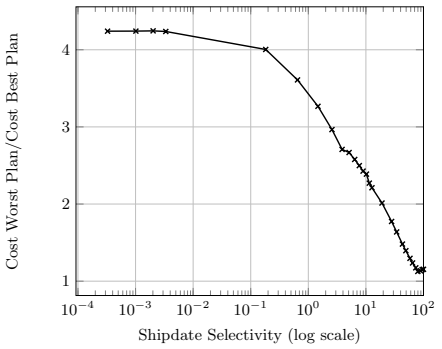


Figure 1: Best v. Worst Plan costs for TPC-H Query 6

```
SELECT sum(l_extendedprice * l_discount) as revenue
FROM lineitem
WHERE l_shipdate <= VALUE and l_quantity < 24
and l_discount between 0.06 - 0.01 and 0.06 + 0.01
```

We implemented Query 6 in C and use the order of the four selection predicates and the selectivity of the shipdate predicate as degrees of freedom. As shown in Figure 1, it is important to select an appropriate plan, especially when the selectivity of the shipdate condition is low. Furthermore, real life databases are bulk loaded and, hence, weakly clustered on the date column. As a consequence, the selectivity varies over the course of the table and thus different plans are optimal for different phases of the scan. Thus, quickly recognizing when a good plan has gone bad requires fine-grained monitoring. In the remainder of this paper, we describe an approach to perform such monitoring at negligible overhead.

The rest of this paper is structured as follows: We provide the necessary background on efficient in-memory data processing in Section 2. In Section 3, we present the hardware-conscious cost models we use as a basis for cost prediction. We describe the process of data characteristics inference in Section 4 and evaluate our approach in Section 5. Finally, we present related work in Section 6 before we conclude in Section 7.

2. BACKGROUND

This section provides the necessary background for measuring and estimating query execution performance. Section 2.1 describes how to transform a query expressed by relational algebra into machine code. Section 2.2 introduces branch-related and cache-related counters that enable us to reason about hardware utilization.

2.1 From Relational Algebra to Machine Code

In the following, we describe how a DBMS transforms a query into executable code using *Just-In-Time* compilation like Hyper [16]. For this transformation, we use the following query on the TPC-H data set. The query calculates the sum of discounts for all lineitems with a quantity less than 100 and a shipdate before February 2nd 1992.

```
Select sum(discount) from lineitem
where quantitiy <=100
and shipdate <= '1992-02-02'
```

This query can be transformed into code written in C (assuming a column-oriented data layout). We emphasize that we convert the shipdate column from date to time-stamp to replace an expensive string comparison with a cheaper integer comparison.

```
for (int i = 0; i < lineitem.size(); i++)
    if(quantity[i] <= 100)
        if(shipdate[i] <= timeStamp)
            sum += discount[i];
```

This C-program iterates over all elements in the lineitem table. For each $tuple_i$, it first checks if its quantity attribute is less or equal to 100. If $tuple_i$ qualifies and its second attribute is evaluated by the second predicate. If the shipdate of $tuple_i$ is before or equal to 1992-02-02 and thus the second predicate qualifies, its discount is added to the overall sum. In general, each predicate evaluation introduces a branch with two possible outcomes. From a performance perspective, there exist three important observations. First, the *quantity* attribute as the first predicate is accessed for each tuple, regardless of its selectivity. Second, the number of accesses to the second attribute *shipdate* depends on the selectivity of the first predicate. Therefore, shipdate is only evaluated for qualifying tuples of the previous predicate. Third, the access to the third column *discount* depends on the selectivity of the second predicate and is only evaluated if all preceding predicates qualify that tuple.

For this transformation, we choose one possible *Query Execution Plan (QEP)* that evaluates the quantity predicate first. However, we could also evaluate the shipdate predicate first to create another QEP. In this paper, we focus on multi-selection queries and refer to each possible order as one *predicate evaluation order* (PEO).

In a final step, a compiler translates the C-code into machine instructions. For each predicate evaluation, the compiler generates one comparison followed by a conditional jump instruction. Additionally, one such pair and an increment instruction for the loop counter is generated for the entire loop. The conditional jump determines the following execution path. If a tuple qualifies, the branch/jump is *not taken* and thus the execution continues with the next instruction. In contrast, if a tuple does not qualify, a branch is *taken* and therefore the program execution jumps to the end of the loop code to test the loop condition. In the latter case, the subsequent instructions to check the second predicate and update the sum are omitted.

2.2 Performance Counters

In this section, we introduce branch-related (Section 2.2.1) and cache-related (see Section 2.2.2) performance counters which allow us to reason about the performance of a QEP. Modern CPUs provide dynamic data obtained from so-called *performance monitoring units (PMU)* to measure the CPU and system resource utilization. For our approach we divide the relevant counters into *constant* counters that do not change their values among all possible PEOs and *mutable* counters. The number of branches taken is constant among all PEOs because all PEOs of the same QEP lead to the same query result and thus induce the same number of qualifying tuples. In contrast, the number of conditional branches, i. e., branches not taken and cache-related counters, vary among PEOs. We refer to Zeuch et al. [23] for a detailed introduction to performance counters for selections on modern CPUs.

2.2.1 Branch-related Counters

Branches strongly impact the query performance on modern CPUs. Therefore, CPUs consist of a dedicated *branch prediction unit* [7] which tries to predict the outcome of each branch. A wrongly predicted branch leads to pipeline flushes, poor instruction cache locality, and limited instruction level parallelism [2]. Ross et al. [19] investigated this effect for multi-selection queries and show, that the branch predictor correctly predicts branches for queries with very high or very low selectivities. On the other hand, queries with medium selectivities lead to many incorrect predictions which accumulate to the worst-case prediction behavior for a selectivity of 50%. The branch-related performance counters in modern CPUs allow us to capture this behavior by counting the number of right and wrong branch predictions. Furthermore, we may divide mispredictions into branches that are mispredicted as taken and branches that are mispredicted as not taken. Finally, PMUs are able to count the number of branches taken and not taken as well as their sum as the number of *conditional* branches [7]. In Figure 2, we plot these counters for a single selection query with varying selectivity. Whereas branch misprediction counters depend on CPU internal branching algorithms, branch taken/not taken counters depend solely on the generated code and thus they are independent of CPU characteristics such as prefetching or *out-of-order* execution.

We exploit the number of branches taken (b_T) to determine the number of qualified tuples by a PEO. If all predicates qualify, only one branch is taken at the end of the loop. In contrast, if one predicate does not qualify, two branches are taken (one to the loop condition and one back to the beginning of the loop). Using n as the number of tuples, we calculate the number of qualifying tuples by $2 * n - b_T$.

We exploit the number of branches not taken (b_{NT}) to determine characteristics of individual predicates during runtime. Each tuple induces $0...p$ branches not taken with p as the number of predicates. Zero branches not taken are induced if the first predicate does not qualify. In contrast, p branches not taken are induced if all predicates qualify. In between these boundaries, each descend of a tuple in the PEO increments the branch not taken counter by one for every tuple that qualifies. As a general performance rule, the less tuples qualify, i. e., branches not taken are induced by a PEO, the better the performance will be. The main reason is that each predicate evaluation induces additional work in terms of computation, memory accesses, and branching.

2.2.2 Cache-related Counters

The memory hierarchy of modern CPUs consists of registers, multiple layers of caches, main memory, and disks. In our approach, we focus on the utilization of the multi-level cache hierarchy to improve the performance of modern in-memory databases [2, 12, 4].

For our approach, we exploit the number of accesses to the L3 cache because they add up demand requests from upper cache levels as well as prefetching requests from the L1 or L2 prefetcher units. In contrast to L3 hits and L3 misses, the number of L3 accesses are independent of CPU characteristics such as prefetching algorithms or *out-of-order* execution. In this paper, we focus on multi-selection queries that exhibit in general no tuple reuse. Therefore, the number of L3 accesses are equal to the number of accesses to the L1 cache and the L2 cache plus prefetch accesses.

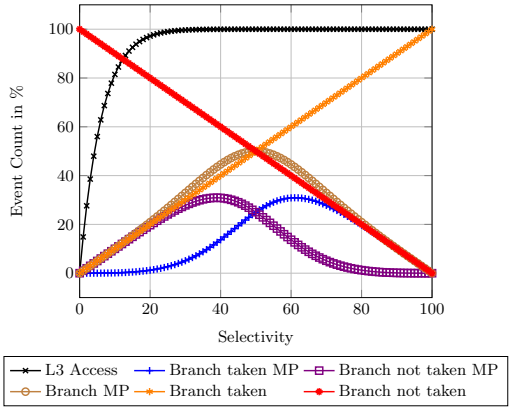


Figure 2: Counter Overview.

For a multi-selection query, the demand on the memory bus depends on the number of predicate evaluations. In general, a subsequent load and compare must be executed if the current predicate qualifies the tuple at hand. Thus, the selectivity as well as the order of the individual predicates impact the number of load operations. However, selectivities cannot be changed because they are determined by the individual predicate and the value distribution of the data set. Therefore, the PEO remains the most important query optimization parameter.

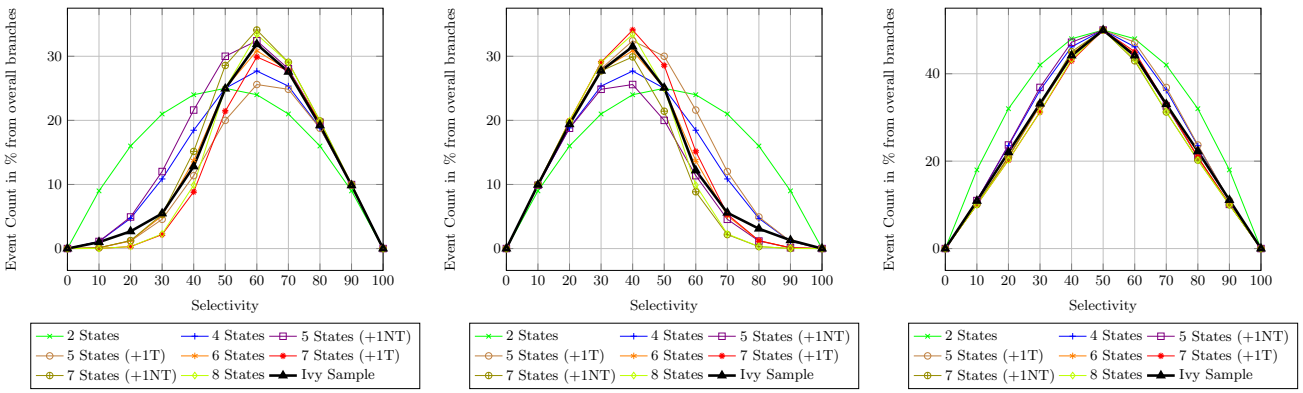
Finally, speculative execution and prefetching impact the utilization of the memory hierarchy in modern CPUs. Speculative execution predicts the outcome of a branch, i. e., if a tuple qualifies. If the prediction is correct, speculatively loaded instructions and data are executed earlier in time and thus the execution is accelerated. However, a wrong prediction induces unnecessary memory accesses and executes expendable instructions. Prefetching on the other hand tries to recognize memory access patterns and prefetches expected memory accesses [7]. Similar to speculative execution, a wrong prefetch induces unnecessary memory accesses and a correct prediction accelerates execution. We refer to Zeuch et al. [23] for an in-depth performance analysis of the relational selection operator using performance counters.

3. COST MODELS

This section presents the underlying cost models of our approach. We introduce a model for cache accesses in Section 3.1 and a model for branch mispredictions in Section 3.2. In Figure 2, we plot the performance counters which are modeled in this section.

3.1 Cache Cost Model

The extension of the generic cost model by Pirk et al. [17] allows us to model the cache accesses of different PEOs. We estimate induced cache accesses of a multi-selection query by exploiting two patterns. The first predicate introduces a *single sequential access pattern* which induces one random access for accessing the first cache line and one sequential access for each subsequent cache line. Each subsequent predicate introduces a *sequential scan with conditional read pattern* which induces cache accesses depending on the selectivity of the previous predicate. We refer to Pirk et al. [17] for a detailed description of this model. Furthermore, the generic cost model introduced by Manegold et al. [13] allows us to model the cache accesses for other relational



(a) Taken Misprediction. (b) Not Taken Misprediction. (c) All Misprediction.

Figure 3: Markov Model Bits.

operators such as joins or sorts by combining atomic access patterns.

Figure 2 shows L3 accesses for an increasing selectivity described by Pirk et al. [17]. The main reason for this behavior is the high number of random misses for small selectivities. These random misses occur because cache lines are omitted. In contrast, with increasing selectivity, the access probability per cache line increases and thus less cache lines are omitted. This behavior is reflected by the reduced number of cache line accesses that are only present in the range of 0-20% selectivity. For a selectivity larger than 20%, each cache line is accessed and thus the number of cache line accesses remains constant. This characteristic also applies for a multi-selection query.

Based on an extended evaluation of the cost model on modern CPUs, we modify the cost model by Pirk et al. [17] to double count the number of random misses which leads to more precise estimations. This modification models the effect, that a random cache miss induces one cache access for the cache line that was predicted but not used and one cache line access for the actually used cache line.

Finally, we provide a cost formular to model the cache accesses for equi-joins. There are two main factors that determine the relative costs of a sequence of join operators: the number of accesses and their locality. The former is determined by the selectivity of the operators preceding a join while the latter is a property of the underlying data distribution which is determined when loading the data. To effectively optimize the order of joins for our cost-based approach we have to take these factors into account. For that purpose, the generic cost model [13] contains equations to predict the number and type (random or sequential) of of cache misses. However, we found the equation for the number of cache misses in the original cost model was highly inaccurate. We, therefore, developed an alternative equation that yields significantly better predictions and is grounded in the external memory model [1]. We used Equation 1 to model random cache misses, the original model for sequential cache misses, and a multiplicative factor to distinguish the two.

$$M_i^r = \begin{cases} C_i & \text{if } C_i < \#_i \\ r * \left(1 - \frac{\#_i \cdot B_i}{R.n \cdot R.w}\right) & \text{if } C_i \geq \#_i \end{cases} \quad (1)$$

The number of accessed cache lines (C_i) is calculated from the size of the relation ($R.n$), the width of a tuple in bytes ($R.w$), the number of accesses (r) and the cache parameters line size (B_i) and capacity in lines ($\#_i$), as in the original model, using Equation 2.

$$C_i = R.n \cdot \left(1 - \left(1 - \frac{1}{R.n}\right)^r\right) \quad (2)$$

3.2 Branch Cost Model

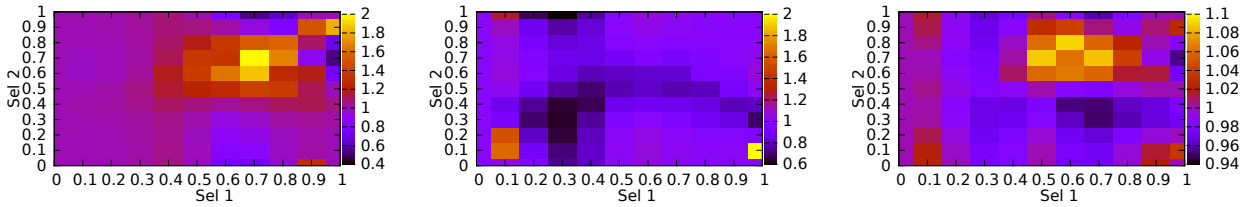
Zeuch et al. [23] point out, that branch mispredictions follow the number of branches not taken for a selectivity below 50% and the number of branches taken for a selectivity above 50%. Thus, for a selection with a selectivity below 50%, the branch predictor predicts that each tuple doesn't qualify (branch is taken) and therefore mispredicts each qualifying tuple (branch not taken). Hence, the number of branch mispredictions is equal to the number of branches not taken. On the other hand, for a selection with a selectivity above 50%, the branch predictor predicts that each tuple qualifies (branch is not taken) and thus mispredicts each not qualifying tuple (branch taken). Based on this observation, Zeuch et al. [23] calculate the number of branch mispredictions (BRMP) using the number of branches not taken (BNT) by:

$$BRMP(p) = \begin{cases} BNT(p), \text{if p} \leq 0.5 \\ BNT(1-p), \text{if p} > 0.5 \end{cases} \quad (3)$$

However, this estimation becomes inaccurate in the selectivity range around 50%. Therefore, in this paper, we propose a markov chain to model the branching behavior of modern CPUs. This markov chain uses a stationary distribution given the selectivity p as the transition probability. In Figure 5, we show a six-state markov chain. In the first three states, the branch predictor predicts the branch to be *not taken*. In contrast, in the last three states, the branch predictor predicts the branch to be *taken*.

The probability of a transition from one state to another is determined by the selectivity p. With a probability of p, a branch is not taken and the current state will transits to its left neighbor state. In contrast, with a probability of $1-p$, a branch is taken and the current state will transits to its right neighbor state. A markov chain allows us to predict the number of mispredictions as well as distinguish them into branches that are mispredicted as taken and branches that are mispredicted as not taken.

In Figure 3, we compare chains using a different number of states ranging from two to eight and the correctness of their prediction. Additionally, we introduce an uneven



(a) Measured/Predicted Not Taken Branch Mispredictions. (b) Measured/Predicted Taken Branch Mispredictions. (c) Measured/Predicted Branch Mispredictions.

Figure 4: Two Predicate Branch Mispredictions.

state count which favors either branches taken (+1T) or branches not taken (+1NT) by adding an additional state. We compare these predictions against real occurrences on a Ivy-Bridge CPU. Zeuch et al. [23] showed, that branching algorithms which impact the execution of a selection have not been changed over the last three micro-architectures Sandy-Bridge, Ivy-Bridge, and Haswell. Thus, we show only real occurrences on the Ivy-Bridge microarchitecture.

As shown in Figure 3, the six state markov chain estimates the number of taken and not taken branches as well as their corresponding sum almost exactly. Therefore, we use a six state markov chain in the remainder of this paper. Considering the number of all branch mispredictions (see Figure 3c), other state counts produce good predictions too. Although these state counts underestimates or overestimates branches taken/not taken, their sum is predicted precisely. Thus, they underestimate one event in the same portion as they overestimate the other. Note, the peak occurrence of mispredicted taken/not taken branches are shifted by 10% percent as opposed to the overall number of mispredictions. Finally, on AMD CPUs, we observe the most precise prediction using four states.

To calculate the branch taken/not taken mispredictions, we solve the following system of equations. Intuitively, the probability that a current state is reached is composed of the probability of the previous and subsequent state multiplied by the probability that these states change to the current state. We label the states as *strong not taken (SNT)*, *not taken (NT)*, *weak not taken (WNT)*, *weak taken (WT)*, *taken (T)*, and *strong taken (ST)*.

$$SNT = SNT * p + NT * p \tag{4a}$$
$$NT = SNT * (1-p) + WNT * p \tag{4b}$$
$$WNT = NT * (1-p) + WT * p \tag{4c}$$
$$WT = WNT * (1-p) + T * p \tag{4d}$$
$$T = WT * (1-p) + ST * p \tag{4e}$$
$$ST = ST * (1-p) + T * (1-p) * p \tag{4f}$$
$$SNT + NT + WNT + WT + T + ST = 1 \tag{4g}$$

By solving this system of linear equations, we can derive the formulas to calculate the probability that a selection with a selectivity p is in a certain state. Using the probabilities of individual states, we sum up the probability that a branch is taken by $B_{Tak} = WT + T + ST$ and the probability that a branch is not taken by $B_{NotTak} = SNT + NT + WNT$. Based on these probabilities, we determine the following estimation formulas for mispredictions (MP) and right predictions (RP). We calculate mispredicted branches taken B_{TakMP} and right predicted branches taken B_{TakMP}, mispredicted branches not taken $B_{NotTakMP}$ and right predicted branches not taken $B_{NotTakRP}$, and the sum of all mispredictions B_{MP} and right predictions B_{RP}. Note that we determine the actual number of mispredictions by multiplying these probabilities with the number of input tuples.

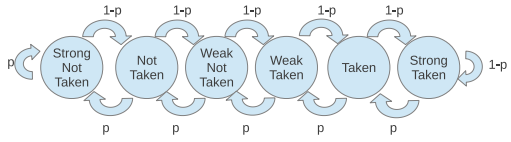


Figure 5: Markov Chain.

$$B_{TakMP} = (1-p) * B_{NotTak} \tag{5a}$$
$$B_{TakRP} = (1-p) * B_{Tak} \tag{5b}$$
$$B_{NotTakMP} = p * B_{Tak} \tag{5c}$$
$$B_{NotTakRP} = p * B_{NotTak} \tag{5d}$$
$$B_{MP} = B_{TakMP} + B_{NotTakRP} \tag{5e}$$
$$B_{RP} = B_{TakRP} + B_{NotTakRP} \tag{5f}$$

In Figure 6, we evaluate our prediction formulas against the latest Intel microarchitectures Nehalem, Sandy-Bridge, Ivy-Bridge, and Broadwell for a selection on 10M tuples. As shown, only Nehalem as the oldest microarchitecture partially differs from our predictions. In contrast, our prediction fits real occurrences on Sandy-Bridge, Ivy-Bridge, and Broadwell quite well. In particular, the overall branch mispredictions are estimated very precisely. However, in the selectivity range around 40% and 60%, there are minor deviations but the overall trend is predicted correctly. Compared to Zeuch et al. [23], we present a more accurate estimation that is able to distinguish between mispredicted taken and not taken branches.

For a multi-selection query, we extend our branch estimations to model each predicate $p_1 ... p_n$. Therefore, we replace the number of input tuples by the number of output tuples of the previous predicate. In Figure 4, we present branch estimations for a selection using two predicates as a 2D heat map. Each axis plots the selectivity of one predicate. At the interception point of two selectivities, we plot the relationship between the measured performance counter and our estimation. As shown, mispredicted branches not taken are underestimated slightly in the selectivity range of 60-80% for both predicates (see Figure 4a). In contrast, mispredicted branches taken are slightly overestimated in the selectivity range of 20-40% for the first predicate (see Figure 4b). Finally, overall branch mispredictions have a minor underestimation in the range of 60-80% for both predicates but the overall estimation differs in less than 10% (see Figure 4c). Despite some outliers, we predict branch events for multi-selection queries very precisely with only minor differences in some selectivity ranges.

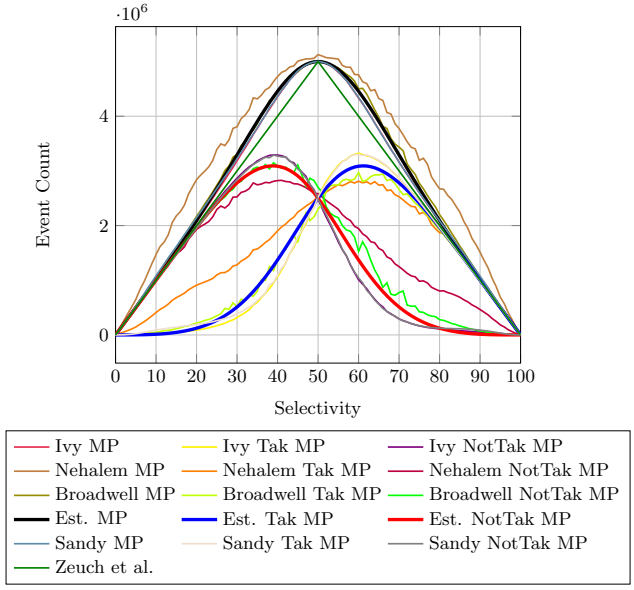


Figure 6: Branch Counter Overview.

4. OPTIMIZATION APPROACH

In this section, we present our progressive optimization approach which exploits the cost models presented in the previous section. Progressive optimization is valuable because determining the *best* predicate evaluation order (PEO) at compile time is rarely possible. The main reasons are uncertain or imprecise information at compile-time such as wrong cardinality estimates, skewed data, correlated attributes, outdated statistics, or user-defined functions which may lead to sub-optimal decisions [8]. Our optimization algorithm alleviates these uncertainties by deriving the selectivity of individual predicates during runtime.

We present our progressive optimization approach in three parts. First, we introduce the search space restriction in Section 4.1 that allows us to prune some areas of the search space. Second, we introduce the non-linear optimization algorithm that explores the pruned search space and our cost models to estimate individual predicate selectivities (Section 4.2). Third, we introduce an algorithm to create different start points inside the pruned search space for the optimization algorithm (Section 4.3). Finally, we summarize the entire optimization process in Section 4.4 before examining the impact of skew and correlation on our approach in Section 4.5.

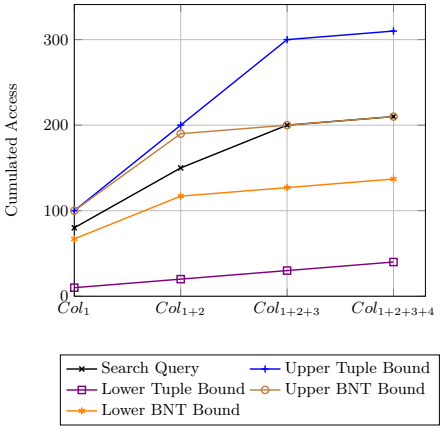


Figure 7: Search Space Restriction.

4.1 Search Space Restriction

The initial search space for given a query with p predicates encompasses a $p-$dimensional space with a possible selectivity between zero and 100% for each predicate. By exploiting the number of input and output tuples of a query, we might restrict this search space. The searched query has to be between the *upper* and *lower* tuple bound. Intuitively, the upper tuple bound represents the highest number of accesses to $col_1 ... col_n$ that is possible considering the given number of input tuples $tups_{in}$ and output tuples $tups_{out}$. In contrast, the lower tuple bound represents the lowest number of accesses to these columns. We define the number of accesses to $col_1 ... col_n$ by predicate $p_1 ... p_n$ for the upper and lower tuple bound as:

$$Upper\,Tuple\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{in}, \text{else} \end{cases} \quad (6)$$

$$Lower\,Tuple\,Bound(p) = tups_{out} \quad (7)$$

In Figure 7, we restrict the search space of an example query. The search query consists of four predicates that select 10 output tuples from 100 input tuples. The accesses to $[col_1, ..., col_4]$ are $[80, 70, 50, 10]$. The sum of these accesses (210) is equal to the number of branches not taken. Using the upper and lower tuple bound, we restrict the possible access intervals for $[col_1, ..., col_4]$ to the lower bound $[10, 10, 10, 10]$ and the upper bound $[100, 100, 100, 10]$. Figure 7 shows the cumulative accesses for our example.

To further restrict the search space, we use the number of branches not taken. The number of branches not taken are independent of runtime or CPU characteristics and thus exact. Furthermore, branches not taken (BNT) exhibit two important characteristics. First, branches not taken by predicate p_i represent the number of accesses to column col_i. Second, we can sample the number of branches not taken for an entire PEO and they correspond to the sum of the accesses to $col_1 ... col_n$. Therefore, the sampled number of branches not taken represents a definite integral among accesses to $col_1 ... col_n$.

Furthermore, we exploit special characteristics of accesses to the first and last column. All tuples in the first column col_0 are always accessed. In contrast, tuples in the last column col_n are accessed for each output tuple. Thus, we can define $access(col_0) = tups_{in}$ and $access(col_n) = tups_{out}$. Note, in the following, we argue among accesses to individual columns which can be converted into the selectivity product of $p_1 ... p_i$ by $\prod_{p=1}^{i} = Acc(col_i)/tups_{in}$. Using the selectivity product, we determine individual predicate selectivity by $p_i = \prod_{p=1}^{i} / \prod_{p=1}^{i-1}$.

We exploit the aforementioned characteristics to further restrict the search space of a search query. At first, cumulative accesses to $col_1 ... col_n$ match the sampled number of branches not taken. Thus, a query that introduces either more or less branches not taken cannot be the search query. Based on the sampled branches not taken and the special characteristics that accesses to col_i can only be less or equal to accesses to col_{i-1}, we can restrict the search space by a new lower and upper bound on the number accesses per column.

A upper BNT bound is defined by assigning accesses to $p_1..p_n$ such that p_i can access the maximum number of tuples. The maximum number of accesses by p_i requires that all previous predicates $p_0...p_{i-1}$ access as many tuples as p_i. Otherwise, the constraint that p_i is only allowed to access less or equal tuples as p_{i-1} would be violated. The remaining predicates $p_{i+1}...p_n$ access the minimum number of tuples ($tups_{out}$). If the maximum number of accesses by p_i exceeds the number of input tuples, we restrict p_i to $tups_{in}$ because no predicate can access more tuples than exist. In Figure 7, each query that has one sample point above the upper BNT bound cannot reach the desired number of output tuples. This would require a predicate to access less tuples than the number of output tuples. Thus, we define the upper BNT bound using BNT_{samp} as the sampled branches not taken:

$$Upper\,BNT\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{in}, \text{if Upper BNT Bound(p)} > tups_{in} \\ \frac{BNT_{samp} - (tups_{out} * (n-p-1))}{p+1}, \text{else} \end{cases} \quad (8)$$

Similarly, we define a lower BNT bound by distributing the number of branches not taken equally among $p_1...p_{n-1}$. Intuitively, each query that has one sample point below this line in Figure 7 cannot reach the desired number of output tuples because no subsequent branch is allowed to induce more BNT than the previous one. Thus, we define the lower BNT bound as:

$$Lower\_BNT\,Bound(p) = \begin{cases} tups_{out}, \text{if p = n} \\ tups_{out}, \text{if Lower BNT Bound(p)} < tups_{out} \\ \frac{BNT_{samp} - tups_{out} - ((p-1) * tups_{in})}{n-1}, \text{else} \end{cases} \quad (9)$$

Using the lower and upper BNT bound, we can restrict the search space for our example query ($[80, 70, 50, 10]$) in Figure 7 further. The accesses to $[col_1, ..., col_4]$ have to be in the interval between $[67, 50, 10, 10]$ and $[100, 95, 66, 10]$. As shown, using the upper and lower BNT bound, we are able to restrict the search space significantly.

4.2 Learning Algorithm

The main challenge for an algorithm that approximates selectivities of individual predicates is the ability to distinguish different queries. We showed that this distinction is possible for a query using one predicate (see Figure 2) and two predicates (see Figure 4). However, for a multi-selection query, we measure performance counters for the entire PEO execution and thus have to infer the individual predicate selectivity.

In our progressive optimization approach, we exploit four performance counters: branches not taken, branch taken and not taken mispredictions, and L3 accesses. These counters can be gathered simultaneously on modern CPUs. In Figure 8, we plot the predictions of these counters for a selection with two predicates on 10M tuples as a 2D heat map. The predictions are calculated by our cost models presented in Section 3. The selectivity of the first predicate is shown on the x-axis and the selectivity of the second on the y-axis. In general, we can distinguish two queries if they differ in at least one of these counter values. In Figure 8, a counter value is represented by the color of the square at the intercept point of the selectivities. For example, a query

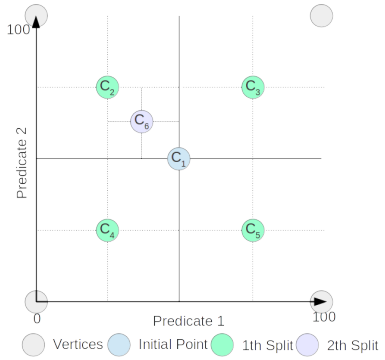


Figure 9: Start Point Selection.

using two predicates with 40% and 20% selectivity (orange square) differs from a query using two predicates with 20% and 40% selectivity (blue square) in the number of mispredicted branches not taken (see Figure 8b).

To learn the selectivities of individual predicates, we apply a non-linear optimization algorithm. We use the open-source library *NLopt*[1]. This library supports different algorithms including gradient-based and derivation free algorithms. Based on an extended evaluation of all available algorithms regarding their correctness and speed, we choose the *Nelder-Mead simplex algorithm* [15] as a local optimization algorithm because it performs best for our selectivity estimations.

Based on this decision, we define the following minimization function for the non-linear optimization that uses the difference between the sampled value and our estimation.

$$\begin{aligned} Costs = &(BNT_{samp} - BNT_{est}) + (L3_{samp} - L3_{est}) \\ &+ (BRNotMP_{samp} - BRNotMP_{est}) \\ &+ (BRTakMP_{samp} - BRTakMP_{est}) \end{aligned} \quad (10)$$

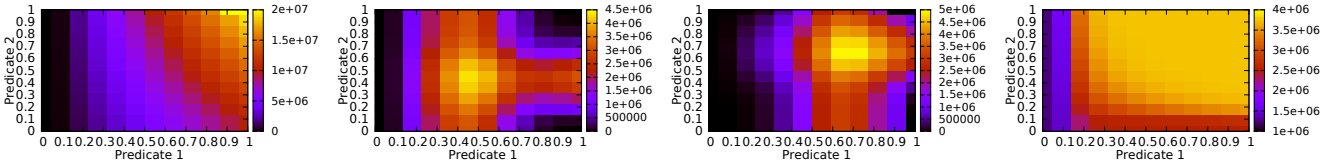
To restrict the optimization effort, we utilize the lower BNT bound (see Equation 9) and upper BNT bound (see Equation 8) as boundaries for the optimization. Additionally, we specify an absolute tolerance from the previous iteration and a maximum iteration count as termination criteria.

The algorithm proceeds as follows. In the first iteration, the algorithm calculates the minimization function from a given start point. Based on the calculated function value, the algorithm internally changes the individual selectivities to values between the upper and lower bound and recalculate the minimization function for these new values. The optimization terminates if the maximum iteration count is reached or the current optima differs less than specified by the absolute tolerance from the last iteration. In our tests, a maximum iteration count of 10k and an absolute tolerance of one result in the best estimations. As a result, the algorithms returns a selectivity estimation for each individual predicate.

4.3 Selection a Starting Point

In our optimization approach, our system of linear equation is under-defined because we cannot utilize as many performance counters as individual predicates exists. Furthermore, it is possible that two PEOs of the same query induce the same performance counter value in each exploited

[1]http://ab-initio.mit.edu/wiki/index.php/NLopt



(a) Predication: Branches Not Taken. (b) Predication: Misp. Branches Not Taken. (c) Predication: Misp. Branches Taken. (d) Predication: L3 Accesses.

Figure 8: Two Predicate Prediction.

counter. This scenario mostly occurs if on query induce an equal distribution of accesses and the other an extreme skew.

As a consequence, when performing the non-linear optimization only once, we could potentially terminate on a local optima. To encounter this problem, we specify a set of start points for our non-linear optimization algorithm and perform the optimization multiple times.

In Figure 9, we outline our approach to create a set of start points for a two-dimensional search space. At first, we create start points at the vertices of each dimension. In this example, we would create start points [0,0], [0,100], [100,100], and [100,0]. We then set the initial point using our null hypothesis. As our null hypothesis, we assume that the overall query selectivity distributes evenly among the predicates. Using this as a start point, we split the search space in 2^n sub-spaces. In Figure 9, the query induces a selectivity of 25% and thus we create the initial point C_1 which splits the search space into four equally sized squares.

For each additional start point, we search for the largest sub-space and return its centroid as a start point for the non-linear optimization algorithm. In Figure 9, the start points in the first splitting phase are C_2, C_3, C_4, and C_5. If an additional start point is required, C_6 would be created. Based on this algorithm, we create start points that are evenly distributed among the search space to avoid the termination on a local optima during the non-linear optimization. Additionally, we explore the largest unseen part of the search space for each new start point.

4.4 Progressive Optimization Algorithm

In Figure 10, we present our progressive optimization algorithm which drives the vectorized execution. First, we measure one vector execution and sample the required performance counters. As next steps, we repeatedly generate a start point (see Section 4.3), run the non-linear optimization algorithm (see Section 4.2), and compare the current optima against the previous optima. This sequence terminates if either no better local optima was found in the previous n iterations or if an overall iteration maximum m is reached. The values of n and m represent a trade-off between the quality of the estimation and the required optimization time. In our experiments, $n < 5$ and $m = 2^p$ with p as the number of predicates lead to the best trade-off between optimization time and estimation precision.

After the sequence terminates, we reorder the predicates according to the best estimation so far. A JIT-compiled system like Hyper [16] would compile a new binary for the new order. In contrast, a vectorized system like Vectorwise [21] could have pre-compile primitives that are chained in the new order. Using the new order, we execute another vector and sample the required performance counters again. If the performance counter values improve, the new order is used for the consecutive vectors. If they deteriorate, the old order is reestablished. Finally, vector execution continues until the next optimization cycle is scheduled.

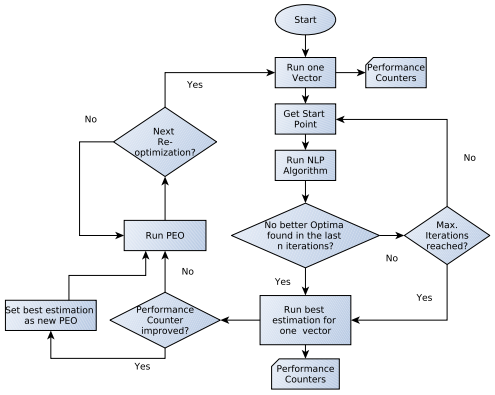


Figure 10: Optimization Sequence.

4.5 Skew and Correlation

Skewed data distributions and correlated attributes are two of the traditional challenges of database query optimization. Both are cases in which the quality of an optimized plan may be low because the cost estimator cannot accurately infer factors such as selectivity coefficients, the probability of collisions when building hashes, or data access locality. However, many of the tuning decisions that are informed by these decisions such as buffer sizes, hash functions, or selection strategies can be adapted during query execution. Processing in fine-grained partitions can, therefore, help to remedy poor decisions based on unpredictable data characteristics such as skew and correlation. In fact, our approach effectively renders high quality decisions at query compilation time unnecessary because it provides better and more adaptive information at runtime.

In our approach, skew is implicitly detected by periodically inspecting the performance of the execution. Thus, if the value distribution of the data set changes for a subset, we could detect this during the next optimization run. In contrast to skew which affects a single attribute, correlation affects a combination of attributes and violates the underlying assumption of modern query optimizers, that the value distribution of a seen data subset applies to the entire data set. It introduces low quality estimates because not seen data subsets may exhibit another distribution. As a consequence, selectivities might change significantly. We handle correlation in our approach by periodically execute different PEOs. With an increasing number of optimization runs per execution, more PEOs are executed and thus the probability that a better PEO is missed cause of correlations is reduced. Furthermore, by determining the amount of data seen by re-

cent PEOs, we are able to introduce special PEO changes to explore unseen data subsets and thus detect correlations.

5. EVALUATION

In this section, we evaluate our progressive optimization approach for different selectivity and value distributions. First, we present our experimental setup in Section 5.1. Then, we start our evaluation by comparing the execution of Q6 with and without our progressive optimization in Section 5.2. After that, we evaluate different selectivity distributions in Section 5.3 and different value distributions in Section 5.4. In Section 5.5 and Section 5.6 we showcase how sortedness can be exploited to reorder QEPs involving join operators. Finally, we investigate the overhead of progressive optimization in Section 5.7.

5.1 Experimental Setup

For our evaluation, we implement the original TPC-H query six (Q6) and several modifications in a C++ prototype. We utilize the common data generator to create a data set using scaling factor 100. This translates into approximately 4,7 GB of data per column of the lineitem table and approximately 600M tuples.

We evaluate our prototype on a Intel Xeon E5-2630 v2 processor. It contains six physical cores at 2.6 GHz frequency and provides 12 logical cores using hyper threading. Additionally, each core utilizes a separate 32KB L1 cache for data and instructions and a unified 256KB L2. All cores share a 15MB L3 cache.

5.2 TPC-H Common Case

In Figure 11, we execute all possible evaluation orders for the five predicates in Q6 (120 orders). This includes the slowest PEO with predicates ordered in descending selectivity order and the fastest PEO with predicates ordered in ascending selectivity order. The black line represents the base line for our evaluation which executes Q6 without progressive optimization. Therefore, we choose one PEO and stick to it for all vectors. The green line represent the runtime with progressive optimization. Overall, this query executes 600 vectors with 1M tuples per vector and we start our optimization approach after each 10th vector. We sort the results on total runtime of the common execution pattern without progressive optimization.

As shown in Figure 11, our approach improves run-time for this query regardless of the first initial PEO choice.

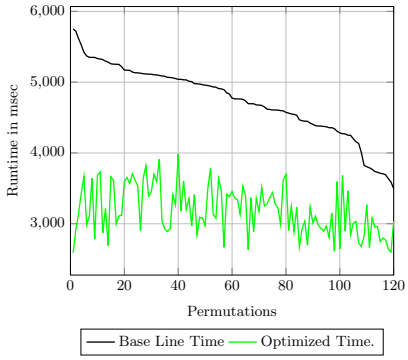


Figure 11: TPC-H Common Case.

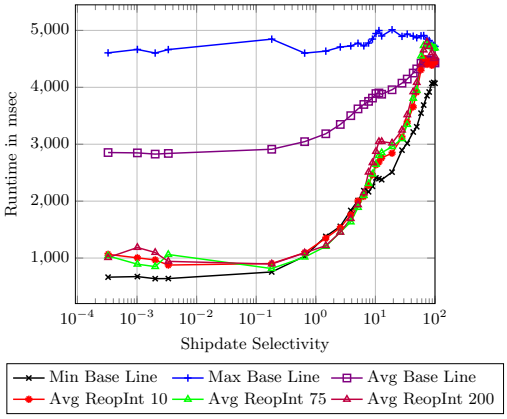


Figure 12: Q6 with varying Shipdate Selectivity.

However, the actual improvement fluctuates to some degree based on the start PEO and thus the time necessary to converge to the best PEO. Our approach improves execution time because we converge to the fastest PEO and react to changing selectivities and data properties during execution.

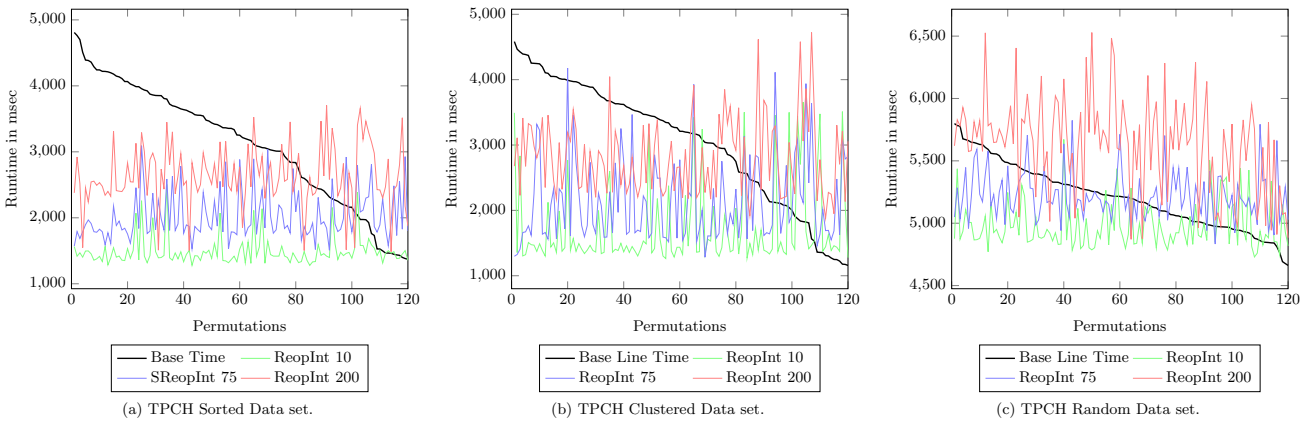
5.3 Selectivity Distribution

This section explores the impact of selectivity on our approach. We execute Q6 using different selectivities on shipdate and show the results in Figure 12. For each selectivity, we plot the minimum, maximum, and average runtime of the common execution pattern without progressive optimization (base line). Additionally, we plot the average runtime using progressive optimization and a reoptimization interval of 10, 75, or 200 vectors. With *ReopInt 10/75/200*, we apply our optimization sequence described in Section 4.4, each 10th, 75th, or 200th vector, respectively. Thus, we might execute different vectors using different PEOs.

For a selectivity below 0.1%, the average execution time using progressive optimization differs up to a factor of two from the minimal base line execution time. In this selectivity range, the position of the shipdate predicate is vital for the resulting query performance. Because the impact is so huge, the necessary optimization time transfer directly to a sub-optimal runtime. The most affected PEOs evaluate the shipdate predicate in the middle of the PEO. In this case, our optimization algorithm requires multiple steps to converge to the optimal PEO. In contrast, if the shipdate predicate is evaluated early or late in the PEO, our progressive optimization algorithm converges very fast to the optimal PEO.

In the selectivity range between 0.1% and 10%, the average runtime using progressive optimization mostly reach the minimal base line runtime. Thus, our optimization algorithm performs very efficiently in this selectivity range. For selectivities over 10%, our optimization algorithm slightly differs from the minimal base line runtime with the largest difference for very high selectivities. In general, large selectivities are hard to detect by our algorithm because the high number of branches not taken leads to a high number of possible selectivity distributions.

Overall, progressive optimization improves runtime up to a factor of three compared to average runtime and up to a factor of 4,5 compared to worst case runtime. Thus, we efficiently alleviate bad initial PEOs and make the overall query execute more robust.



(a) TPCH Sorted Data set. (b) TPCH Clustered Data set. (c) TPCH Random Data set.

Figure 13: Q6 on Different Value Distributions.

5.4 Sortedness

In this experiment, we explore the impact of sortedness on progressive optimization. We examine the runtime of Q6 on differently sorted data sets and present the results in Figure 13.

In Figure 13a, the data set is sorted on the shipdate column in ascending order. In general, shorter reoptimization intervals result in better runtimes for sorted data. The main reason for that is the point in time, at which the optimization algorithm detects that a better PEO exists and change to it. For Q6 with a lower and upper bound on the shipdate column, there are three different optimal PEOs during execution. In the first data partition, it is beneficial to evaluate the lower shipdate bound first because it has an effective selectivity of 0%. In the middle partition of the data set where shipdates fall in between the lower and upper shipdate bound, both shipdate predicates should be evaluated as late as possible in the PEO. Finally, in the last partition, the upper shipdate bound should be evaluated first to eliminate unnecessary overhead.

Based on these partitions, the optimal PEO changes during query execution. The larger the reoptimization interval, the later a transition between partitions will be detected. In the worst case, a transition is bypassed and an entire partition is executed using a sub-optimal PEO. This situation occurs for larger optimization intervals of 75 and 200 vectors and lead to increased runtimes. Finally, for faster initial PEOs (Permutation 80-120), this translates to slower runtimes for progressive optimization compared to the common execution pattern. However, using progressive optimization and a reoptimization interval of ten, we still introduce robust query execution with runtimes faster or at least as far as the common execution pattern.

In Figure 13c, the data set is sorted randomly. As a result, each predicate has an arbitrarily selectivity on each vector. Therefore, the underlying assumption of progressive optimization that we can predict future runtime based on sampling current vector execution, is no longer valid. The reoptimization interval of 10 leads to the best runtimes because it reacts most rapidly to changing value distributions. However, compared to the sorted data set, the runtimes are even increased for faster initial PEOs (Permutations 90-120). With larger optimization intervals, the improvements of progressive optimization decrease further. Using a reoptimization interval of 200, the runtime is almost always above base line execution.

In Figure 13b, we use knuth shuffling to redistribute the shipdate column. To introduced a clustered data set, we shuffle lineitems based on the shipdate column within the time frame of a month. This represents a middle ground between a sorted and random data set. Compared to a sorted data set, runtimes increase slightly for small reoptimization intervals and moderate for large reoptimization intervals. Compared to a random data set, the overall runtime is still improved.

Overall, the improvements of progressive optimization decrease for randomly distributed data sets. In particular, a decreased number of initial PEOs are improved. In general, a short reoptimization interval leads to the best results. However, in the next section, we present a method to detect the sortedness of a data set which could be exploited to decide if our progressive optimization should be applied and which optimization interval should be used.

5.5 Sortedness and Expensive Predicates

In the previous section we showed how important sortedness is if we try to choose the optimal PEO. In this experiment, we utilize performance counters to detect the sortedness of a data set. In Figure 14, we plot runtime and cache misses for a query using an expensive selection and a foreign key join. On the x-axis, we show different degrees of sortedness using knuth shuffle ranging from a sorted data set (1T) to a random data set (Mem). In between, the shuffle distance hit the size of a cache line (CL), L1, L2, or L3 cache. On these data sets, we run a query that either executes a selection or a foreign key join first and the other operator afterwards.

As shown in Figure 14a, there is a break even point for runtime. This point is reached if the shuffle distance exceeds twice the L1 cache size. For a sortedness below this point, it is cheaper to perform the join before applying the selection. The join is so cheap because such a sortedness introduces a highly local access pattern which induces only few cache misses. In contrast, if the sortedness spreads over this point, the join becomes more expensive and thus the selection should be applied first.

Note that, this kind of sortedness analysis can only be derived from performance counter. In particular, counting the number of qualifying tuples per vector is not sufficient. Therefore, measuring the number of cache misses (see Figure 14b) allows us to infer the sortedness. In this scenario, the trend of the runtime and the number of cache misses

correlate. Thus, we could derive sortedness and reorder the operations using progressive optimization.

5.6 Sortedness for Forgein Key Join

In this experiment, we use our optimization approach to optimize the join order in a QEP. In Figure 15, we join the lineitems table of the TPC-H benchmark with the order and part table in different orders. On the x-axis, we show the selectivity of both joins. Commonly, a query optimizer would join lineitems first with part because it is about eight times smaller than orders. However, as shown in Figure 15a, for all selectivities, joining orders first is always faster. The main reason for this is, that lineitems and orders are co-clustered. In contrast, the access pattern to the part table is random. This co-clusteredness leads to an improved accesses pattern with less cache misses as shown in Figure 15b.

In our approach, we exploit Equation 1 from Section 3.1 to determine if a join is executed on a co-clustered table pair. Using Equation 1, we can estimate the expected number of cache misses for a random access pattern. Then, we compare these values against the sampled cache accesses. If they match, we might reorder the join order; thus, eventually switching to a join order where a co-clustered join is executed first. In contrast, when sampling much less cache misses than expected, we gain knowledge that the we probably execute a co-clustered join first and do not have to reorder. It is important to note, that this kind of join order optimization can be exploited in our approach in addition to the branch not taken/cache miss sampling approach which we use for predicate only QEPs.

5.7 Overhead

In this experiment, we evaluate the overhead of our progressive optimization using performance counters against a counter-based approach, called *enumerator-based* approach in the following. An enumerator-based approach would insert explicit counter variables into the source code after each predicate evaluation to obtain the individual selectivities. In contrast, we use non-invasive performance counters to approximate these selectivities.

The overhead of progressive optimization is comprised of two components. First, we have to compare the exploitation of performance counters against the usage of explicit counter variables. In Figure 16, we measure the overhead for both variants for different predicate counts. As shown, for larger predicate counts, the enumerator-based approach almost doubles the runtime of the overall query. In contrast,

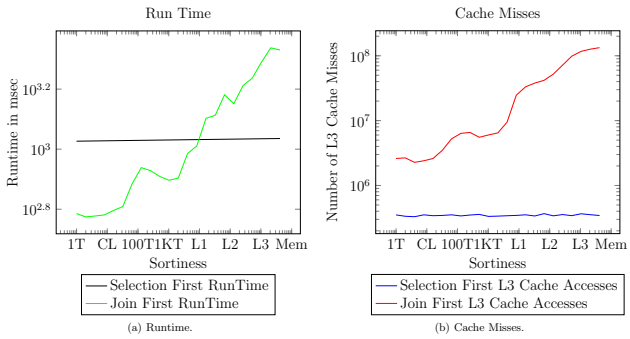


Figure 14: Exploitation of Sortedness.

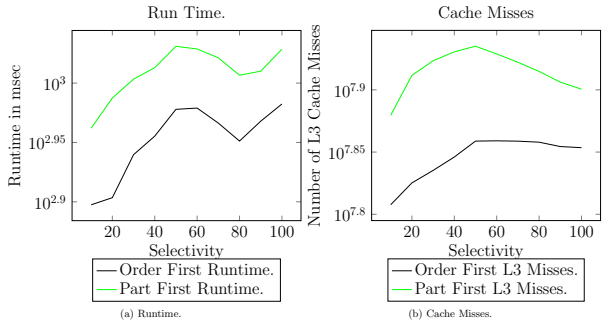


Figure 15: Foreign Key Join with different Orders.

performance counter do not impact the runtime. This observation follows Intel's statement that performance counters do not or only minimally impact the execution performance [7]. In the common case selectivities do not change between two optimization attempts and thus the enumerator-based approach would double the run-time during each optimization cycle. In contrast, performance counter introduce virtually no costs.

Second, we have to compare the algorithm to infer the individual predicate selectivity used by our approach against a similar algorithm for the enumerator-based approach. In our evaluation we showed, that optimization overhead contribute only minor to the total runtime. We assume, a comparable algorithm for an enumerator-based approach should perform similar. Finally, progressive optimization using performance counter rely solely on existing implementations. In contrast, an enumerator-based approach has to maintain implementations with and without counter variables for each operator.

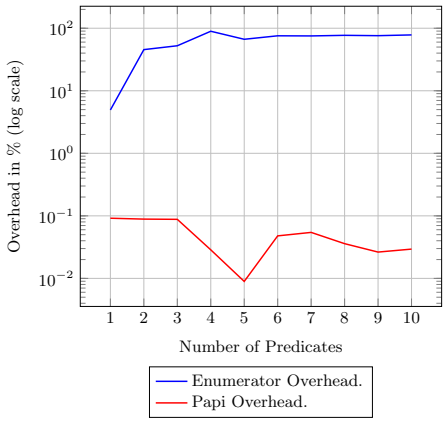


Figure 16: Overhead.

6. RELATED WORK

Previous work on progressive optimization by Markl et al. [14], Kabra et al. [8], and Babu et al. [3] validates cardinality estimates against actual values measured during runtime execution. If a significant disagreement is detected, the query execution might stop and a reoptimization process is triggered. Kache et al. [9] extends this approach for federated databases and Han et al. [6] for shared-nothing parallel databases. This approaches can be collectively termed as plan-switching approaches, as they involve run-time switching among complete query plans. In contrast to these approaches, we base our reoptimization decision on actual per-

formance counters which induce virtually no costs. Furthermore, we progressively optimize the current execution by inspecting the query vector-wise. This enables us to perform a more fine-grained optimization. Finally, we are able to exploit more properties than just the cardinality to reoptimize query plans and do not require any statistics over the data.

Stillger et al. [22] presents with the LEarning Optimizer LEO an approach to repair incorrect statistic and cardinality estimates. By monitoring previously executed queries, LEO computes adjustments based on the difference between optimizer estimates and actual measured costs. In contrast, our approach learns from the vector-wise processing of the same query to optimize future vector execution. Thus, we provide a feedback loop during run-time as opposed to a feedback loop among multiple query executions like in LEO.

Rducanu et al. [20], propose a micro adaptivity approach to learn the best implementation of a function during runtime. Therefore, they measure the run-time of different function implementations and apply a learning algorithm to choose the most promising implementation. In contrast, by sampling performance counters instead of run-time or even incremental tuple counters, we are able to learn properties of the data sets like sortedness or co-clusteredness of joins. Furthermore, we significantly reduce the overhead during run-time and provide a non-invasive approach.

Another research area discovers the best QEP based on a subset of possible best plans. Dutt et al. [5] propose to exploit a *bouquet* of plans from a set of optimal plans such that at least one of this plans is near-optimal. In contrast to our approach, they require more overhead during compile-time as well as during run-time. During compile time, they have to gather the bonquet of plans. In contrast, we create different orderings of operators during run-time using JIT compilation. During run-time, Dutt et al. [5] introduce explicit counters between operators. In contrast, we exploit performance counter which nearly induce no costs.

7. CONCLUSION AND FUTURE WORK

This paper provides the necessary cost models to enable performance counters for progressive optimization. Progressive optimization using performance counters avoids worst case predicate evaluation orders efficiently. Using progressive optimization, we improve runtime up to a factor of three compared to average runtime and up to a factor of 4,5 compared to worst case runtime. Thus, we efficiently alleviate slow initial PEOs and make the overall query execute more robust. At the same time, the optimization overhead could be restricted by fine tuning the termination criteria of the underlying non-linear optimization algorithm, the number of optimizations during execution, and the effort that is spent to find the best optimization result.

Our evaluation showed, that expect for a random data distribution, we almost always improve runtime compared to the common execution pattern through periodically reoptimizing sub-optimal PEOs. Finally, we showed that the impact of sortedness, skew, and correlation can be alleviated by our approach.

In future work, we will integrate other relational operators into our optimization approach. Additionally, if new performance counters become available through new processor technologies, we will implement them to improve our estimations.

8. REFERENCES

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 1988.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor : Where does time go ? *VLDB*, 1999.

[3] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. SIGMOD, 2005.

[4] P. Boncz, M. Zukowski, and N. Nes. Monetdb / x100 : Hyper-pipelining query execution. *CIDR, 2005*.

[5] A. Dutt and J. R. Haritsa. Plan bouquets: Query processing without selectivity estimation. SIGMOD, 2014.

[6] J. M. V. Han, Wook-shin Ng. Progressive optimization in a shared-nothing parallel database. *SIGMOD, 2007*.

[7] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2012.

[8] N. Kabra and D. J. Dewitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD, 1998*.

[9] H. Kache, V. Raman, S. Str, and V. Markl. Pop / fed : Progressive query optimization for federated queries in db2. *EDBT, 2006*.

[10] V. Leis, et al. The adaptive radix tree: Artful indexing for main-memory databases. *ICDE, 2013*.

[11] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree : A b-tree for new hardware platforms. *ICDE, 2013*.

[12] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowl. and Data Eng. 14*, 2002.

[13] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. *VLDB*, 2002.

[14] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. *SIGMOD, 2004*.

[15] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal 7*, 1965.

[16] T. Neumann. Efficiently compiling efficient query plans for modern hardware. VLDB, 2011.

[17] H. Pirk, A. Kemper, F. Funke, S. Manegold, U. Leser, M. Grund, T. Neumann, and M. Kersten. Cpu and cache efficient management of memory-resident databases. *ICDE, 2013*.

[18] V. Raman, et al. Db2 with blu acceleration: So much more than just a column store. *VLDB, 2013*.

[19] K. A. Ross. Selection conditions in main memory. *TODS*, 2004.

[20] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. *SIGMOD, 2013*.

[21] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. *DaMoN, 2011*.

[22] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2 s learning optimizer. *VLDB, 2001*.

[23] S. Zeuch and J.-c. Freytag. Selection on modern cpus. *IMDM*, 2015.