Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling

Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, Andrew Pavlo
Carnegie Mellon University
{pmenon,angom,lin.ma,tcm,pavlo}@cs.cmu.edu

ABSTRACT

Just-in-time (JIT) query compilation is a technique to improve analytical query performance in database management systems (DBMSs). But the cost of compiling each query can be significant relative to its execution time. This overhead prohibits the DBMS from employing well-known adaptive query processing (AQP) methods to generate a new plan for a query if data distributions do not match the optimizer's estimations. The optimizer could eagerly generate multiple sub-plans for a query, but it can only include a few alternatives as each addition increases the compilation time.

We present a method, called Permutable Compiled Queries (PCQ), that bridges the gap between JIT compilation and AQP. It allows the DBMS to modify compiled queries without needing to recompile or including all possible variations before the query starts. With PCQ, the DBMS structures a query's code with indirection layers that enable the DBMS to change the plan even while it is running. We implement PCQ in an in-memory DBMS and compare it against non-adaptive plans in a microbenchmark and against state-of-the-art analytic DBMSs. Our evaluation shows that PCQ outperforms static plans by more than 4× and yields better performance on an analytical benchmark by more than 2× against other DBMSs.

PVLDB Reference Format:
Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, Andrew Pavlo. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. PVLDB, 14(2): 101 - 113, 2021.
doi:10.14778/3425879.3425882

1 INTRODUCTION

In-memory DBMSs assume that a database's primary storage location is in DRAM, which means disk I/O is not a bottleneck during query execution. This has led to new research on improving OLAP query performance for in-memory DBMSs by (1) reducing the number of instructions that the DBMS executes to process a query, and (2) decreasing the cycles-per-instruction (CPI) [9].

One approach to reducing the DBMS's instruction count during query execution is a method from the 1970s [10]: just-in-time (JIT) *query compilation* [21, 39]. With this technique, the DBMS compiles queries (i.e., SQL) into native machine code that is specific to that query. Compared with the interpretation-based query processing approach that is used in most systems, query compilation results in faster query execution because it enables the DBMS to specialize both its access methods and intermediate data structures (e.g., hash tables). In addition, by optimizing locality within tight loops, query compilation can also increase the likelihood that tuple data is passed between operators directly in CPU registers [27, 28].

Although query compilation can accelerate the execution of a query plan, existing compilation techniques cannot overcome poor choices made by the DBMS's optimizer when constructing that plan. For example, the optimizer may choose the wrong operation orderings, its estimations of data structure sizes may be incorrect, and it may fail to optimize code paths for "hot" keys. These sub-optimal choices by the optimizer arise for several reasons, including: (1) the search spaces are exponential (hence the optimal plan might not even be considered), and (2) the cost models that optimizers use to estimate the quality of a query plan are notoriously inaccurate [23] because their estimates are based upon summarizations of the database's distributions (e.g., histograms, sketches, samples) that often fail to capture correlations and other variations.

One approach to overcoming poor choices by the optimizer is *adaptive query processing* (AQP) [12], which introduces a dynamic feedback loop into the optimization process. With this approach, the DBMS observes the behavior of the query during execution and checks whether the assumptions made by the optimizer match what the data actually looks like. If the system observes a large enough deviation from what the optimizer estimated, the DBMS can dynamically adapt by either: (1) causing the optimizer to select a different execution strategy for subsequent query invocations, or (2) halting the currently-executing query and having the optimizer generate a new plan (incorporating the information that had just been observed from the query execution so far) [6].

AQP offers performance advantages in interpreter-based DBMSs, but is ineffective in a compilation-based DBMS for two reasons. First, compiling a new query plan is expensive: often on the order of several hundreds of milliseconds for complex queries [20]. Second, the conditions of the DBMS's operating environment may change throughout the execution of a query. Thus, achieving the best performance is not a matter of recompiling once; the DBMS needs to continuously adjust throughout the query's lifetime. For example, the optimal predicate ordering can change between blocks within the same table if data distributions change. Lastly, while DBMSs often cache query plans as prepared statements and invoke them multiple times, each invocation of a cached plan will almost always have different concurrent queries and input parameters.

To overcome the gap between rigid compilation and fine-grained adaptivity, we present a system architecture and AQP method for JIT-based DBMSs to support **Permutable Compiled Queries** (PCQ). The key insight behind PCQ is our novel compile-once approach: rather than invoking compilation multiple times or pre-compiling multiple physical plans, with PCQ we compile a single

Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425882

physical query plan. But, we design this single plan so that the DBMS can easily permute it later without significant recompilation. For example, a query with five conjunctive filters has 120 potential orderings of its terms; our PCQ framework can dynamically switch between any of these 120 filter orderings (based upon changes in observed selectivity), achieving the high performance of compiled code while compiling each filter only once.

Our approach is the amalgamation of two key techniques. First, we use a hybrid query engine that combines pre-compiled vectorized primitives [9] with JIT query compilation with pipelines [28]. The DBMS generates compiled plans for queries on-the-fly, but those plans may utilize pre-compiled primitives. Second, we embed lightweight hooks in queries' low-level execution code that allow the DBMS to both (1) observe a pipeline's behavior during query execution, and (2) modify that pipeline while it is still running. The DBMS also uses metrics collected for one pipeline to optimize other pipelines before it compiles them.

We implemented our PCQ method in the **NoisePage** DBMS [4]. NoisePage is an in-memory DBMS that uses a compilation-based execution engine. Our experimental results show that PCQ improves performance by up 4× for plans generated from commercial optimizers. We also compare against existing in-memory OLAP systems (Tableau HyPer [3], Actian Vector [1]) and show that NoisePage with the PCQ framework achieves up to 2× better performance.

2 BACKGROUND

We first provide an overview of query compilation and adaptive query planning. We then motivate blending these techniques to support optimizations that are not possible in existing DBMSs.

2.1 Query Compilation

When a new query arrives, the DBMS's optimizer generates a plan tree that represents the data flow between relational operators. A DBMS that does not compile queries interprets this plan by walking the plan tree to execute the query. This interpretation means that the DBMS follows pointers and goes through conditional branches to process data. Such an execution model is an anathema to an in-memory DBMS where disk access is no longer a bottleneck. With query compilation, the system converts a plan tree into code routines that are "hard-coded" for that query. This reduces the number of conditionals and other checks during execution.

In general, there are two ways to compile queries in a DBMS. One method is for the DBMS to emit source code that it then compiles to machine code using an external compiler (e.g., gcc) [19, 21]. This approach is used in MemSQL (pre-2016) and Amazon Redshift. The other method is for the DBMS to generate an intermediate representation (IR) that it compiles using an embedded compiler library running in the same process (e.g., LLVM) [28]. This second approach obviates the need for the DBMS to invoke the compiler as an external process. This is the approach used in HyPer, MemSQL (post-2016) [32], Peloton [27], Hekaton, and Splice Machine.

Despite the performance benefits of compilation-based execution, one of the main issues with it is the compilation time itself. An interpretation-based DBMS can immediately start executing a query as soon as it has a query plan. A compilation-based DBMS, however, has to wait until the compilation step for a query finishes before it starts executing that query. If the system uses an external compiler, then this step can take seconds for each query [21]. But even a compiler running inside of the DBMS can take hundreds of milliseconds per query. The compilation time increases even more if the system's compiler employs heavyweight optimization passes.

There are three ways to reduce a query's compilation time. The first is to pre-compile as much as possible so that the compiler has less work to do. Another way is to compile a query's plan in stages rather than all at once. This reduces the start-up time before the DBMS begins executing the query. Decomposing the query in this manner, however, may reduce the efficacy of the compiler's optimizations because they only examine a subset of the plan. Lastly, the third way is to use an interpreter to start executing a query immediately while the query compiles in the background [20]. Such an interpreter processes the query using the same IR that the compiler converts into machine code. The DBMS can start execution immediately without waiting for the compilation to finish. Then when the compilation completes, the DBMS seamlessly replaces the interpreted execution with the compiled plan.

2.2 Adaptive Query Processing

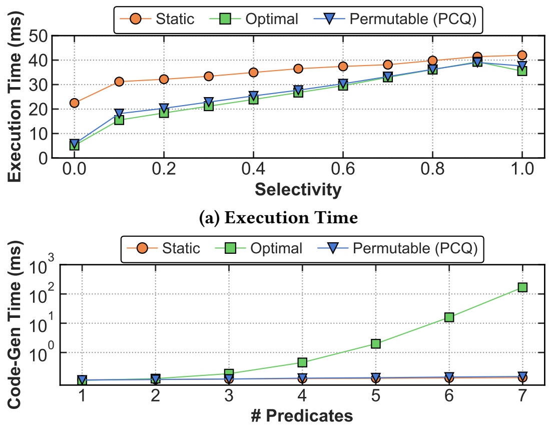
Since it is prohibitively expensive to examine the entire database to make decisions during query planning, some optimizers rely on cost models to approximate query plan costs. But when the DBMS executes the query plan, it may discover that these approximations were incorrect. For example, optimizers often underestimate the output cardinality of join operators, which leads to bad decisions about join orderings and buffer allocation sizes [23]. There are also other aspects of the DBMS's environment that an optimizer can never know, such as interference from other queries that affect memory channels and CPU caches.

AQP is an optimization strategy where the DBMS modifies a query plan to better tailor it to the target data and operating environment. Unlike in the"plan-first execute-second" approach, with AQP, the DBMS interleaves the optimization and execution stages such that they provide feedback to each other [6]. For example, while executing a query, the DBMS can decide to change its plan based on the data it has seen so far. This change could be that the DBMS throws away the current plan and go back to the query optimizer to get a new plan [26]. Alternatively, the DBMS could change yet-to-be-executed portions of the current plan (i.e., pipelines) [40].

There is a trade-off between the cost of context switching to the optimizer and restarting execution versus letting the query continue with its current plan. It is unwise to restart a query if its current plan has already processed a substantial portion of the data necessary for the query. To avoid restarting, some DBMS optimizers generate multiple alternative sub-plans for a single query. The granularity of these sub-plans could either be for an entire pipeline [11, 16] or for sub-plans within a pipeline [7]. The optimizer injects special operators (e.g., change [16], switch [7]) into the plan that contain conditionals to determine which sub-plan to use at runtime.

2.3 Reoptimizing Compiled Queries

Although there are clear benefits to AQP, generating a new plan or including alternative pipelines in a query is not ideal for compilation-based systems. Foremost is that compiling a new plan from scratch



(a) Execution Time

(b) Code-Generation Time

Figure 1: Reoptimizing Compiled Queries – PCQ enables near-optimal execution through adaptivity with minimal compilation overhead.

is expensive. But even if the DBMS's optimizer pre-computed all variations of a pipeline before compiling the query, including extra pipelines in a plan increases the compilation time. The DBMS could compile these pipelines in the background [20], but then it is using CPU resources for compilation instead of query execution.

There are also fine-grained optimizations where it is infeasible to use either of the two above AQP methods. For example, suppose the DBMS wants to find an ordering of predicates in a table scan such that the most selective predicates are evaluated first. Since the number of possible orderings is combinatorial, the DBMS has to generate a separate scan pipeline for each ordering. The number of pipelines is so high that the computation requirements to compile them would dominate the system. Even if the DBMS compiled alternative plans on-the-fly, it still may not adapt quickly enough if both the data and operating environment change during execution.

To help motivate the need for low-overhead AQP in compilation-based DBMSs, we present an experiment that measures the performance of evaluating a **WHERE** clause during a sequential scan on a single table (A) composed of six 64-bit integer columns (col1–col6) that has 10m tuples. The workload is comprised of a single query:

```
SELECT * FROM A
  WHERE col1 = δ₁ AND col2 = δ₂ AND ... AND col6 = δ₆
```

We generate each column's data and choose each filtering constant (δ_i) so that the overall selectivity is fixed, but each predicate term's selectivity changes for different blocks of the table. We defer the description of our experimental setup to sec. 5.

We first measure the time the DBMS takes to execute the above query using the best "static" plan (i.e., one with a fixed evaluation order chosen by the DBMS optimizer). We also execute an "optimal" plan that is provided the best filter ordering for each data block a priori. The optimal plan is as if the DBMS compiled all possible pipelines for the query and represents the theoretical lower bound execution time. Lastly, we also execute the query using permutable filters that the DBMS reorders based on selectivities.

The results in fig. 1a show that the static plan is up to 4.4× slower than the optimal plan when selectivity is low. As selectivity increases, the performance gap gradually reduces since more tuples must be processed. Our second observation is that PCQ is consistently within 10% of the optimal execution time across all selectivities. This is because it periodically reorders the predicate terms based on real-time data distributions.

Next, we measure the code-generation time for each of the three approaches as we vary the number of filter terms. In this experiment, we add an additional filter term on col1 to form a range predicate. The results in fig. 1b reveal that when there are fewer than three filter terms, the code-generation time for all approaches is similar. However, beyond three terms, the optimal approach becomes impractical as there are $O(n!)$ possible plans to generate. In contrast, the code-generation time for the permutable query increases by ~20% from one to seven terms.

Given these results, what is needed is the ability for a compilation-based DBMS to dynamically permute and adapt a query plan *without* having to recompile it, or eagerly generate alternative plans.

3 PCQ OVERVIEW

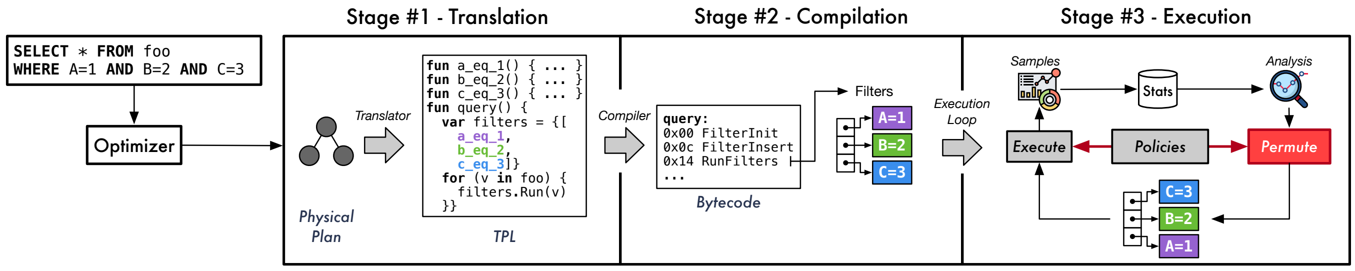
The goal of PCQ is to enable a JIT-based DBMS to modify a compiled query's execution strategy while it is running without (1) restarting the query, (2) performing redundant work, or (3) pre-compiling alternative pipelines. A key insight behind PCQ is to *compile once* in such a way that the query can be permuted later while retaining compiled performance. At a high-level, PCQ is similar to proactive reoptimization [7] as both approaches modify the execution behavior of a query without returning to the optimizer for a new plan or processing tuples multiple times. The key difference, however, is that PCQ facilitates these modifications for compiled queries without pre-computing every possible alternative sub-plan or pre-defining thresholds for switching sub-plans. PCQ is a dynamic approach where the DBMS explores alternative sub-plans at runtime to discover execution strategies that improve a target objective function (e.g., latency, resource utilization). This adaptivity enables fine-grained modifications to plans based on data distribution, hardware characteristics, and system performance.

In this section, we present an overview of PCQ using the example query shown in fig. 2. As we discuss below, the life-cycle of a query is broken up into three stages. Although we designed the framework for NoisePage's LLVM-based environment, it works with any DBMS execution engine that supports query compilation.

3.1 Stage #1 – Translation

After the DBMS's optimizer generates a physical query plan, the *Translator* converts the plan into a domain-specific language (DSL), called TPL, that decomposes the it into pipelines. TPL combines Vectorwise-style pre-compiled primitives [9] with HyPer's data-centric code generation [28]. Using TPL enables the DBMS to apply database-specific optimizations more easily than a general-purpose language (e.g., C/C++). Moreover, as we describe below, TPL enjoys low-latency compilation time.

Additionally, the Translator augments the query's TPL program with additional PCQ constructs to facilitate permutations. The first is *hooks for collecting runtime performance metrics* for low-level operations in a pipeline. For example, the DBMS adds hooks to the generated program in fig. 2 to collect metrics for evaluating **WHERE** clause predicates. The DBMS can toggle this collection on and off



```
SELECT * FROM foo
WHERE A=1 AND B=2 AND C=3
```

```
fun a_eq_1() { ... }
fun b_eq_2() { ... }
fun c_eq_3() { ... }
fun query() {
  var filters = {[
    a_eq_1,
    b_eq_2,
    c_eq_3]}
  for (v in foo) {
    filters.Run(v)
  }}
```

```
query:
0x00 FilterInit
0x0c FilterInsert
0x14 RunFilters
...
```

Figure 2: System Overview – The DBMS translates the SQL query into a DSL that contains indirection layers to enable permutability. Next, the system compiles the DSL into a compact bytecode representation. Lastly, an interpreter executes the bytecode. During execution, the DBMS collects statistics for each predicate, analyzes this information, and permutes the ordering to improve performance.

depending on whether it needs data to guide its decision-making policies on how to optimize the query's program.

The second type of PCQ constructs are parameterized runtime structures in the program that use *indirection to enable the substitution of execution strategies* within a pipeline. The DBMS parameterizes all relational operators in this way. This design choice follows naturally from the observation that operator logic is comprised of query-agnostic and query-specific sections. Since the DBMS generates the query-specific sections, it is able to generate different versions uses indirection to switch at runtime. We define two classifications of indirection. The first level is when operators are unaware or unconcerned with the specific implementation of query-specific code. The second level of indirection requires coordination between the runtime and the code-generator.

In the example in fig. 2, the Translator organizes the predicates in an array that allows the DBMS to rearrange their order. For example, the DBMS could switch the first predicate it evaluates to be on attribute `foo.C` if it is the most selective. Each entry in the indirection array is a pointer to the generated code. Thus, permuting this part of the query only involves lightweight pointer swapping.

3.2 Stage #2 – Compilation

In the second stage, the *Compiler* converts the DSL program (including both its hooks for collecting runtime performance metrics and its use of indirection to support dynamic permutation) into a compact bytecode representation. This bytecode is a CISC instruction set composed of arithmetic, memory, and branching instructions, as well as database-level instructions, such as for comparing SQL values with `NULL` semantics, constructing iterators over tables and indexes, building hash tables, and spawning parallel tasks.

In fig. 2, the query's bytecode contains instructions to construct a permutable filter to evaluate the **`WHERE`** clause. The permutable filter stores an array of function pointers to implementations of the filter's component. The order the functions appear in the array is the order that the DBMS executes them when it evaluates the filter.

3.3 Stage #3 – Execution

After converting the query plan to bytecode, the DBMS uses adaptive execution modes to achieve low-latency query processing [20]. The DBMS begins execution using a bytecode interpreter and asynchronously compiles the bytecode into native machine code using LLVM. Once the background compilation task completes, native function implementations are automatically executed by the DBMS.

During execution, the plan's runtime data structures use policies to selectively enable lightweight metric sampling. In fig. 2, the DBMS collects selectivity and timing data for each filtering term periodically with a fixed probability. It uses this information to construct a ranking metric that orders the filters to minimize execution time given the current data distribution. Each execution thread makes an independent decision since they operate on disjoint segments of the table and potentially observe different data distributions. All permutable components use a library of *policies* to decide (1) when to enable metric collection and (2) what adaptive policy to apply given new runtime metric data. The execution engine continuously performs this cyclic behavior over the course of a query. All policies account for the fact that execution threads may be concurrently executing native and bytecode implementations of query functions and observe varying runtimes.

NoisePage uses a push-based batch-oriented engine that combines vectorized and tuple-at-a-time execution in the same spirit as Relaxed Operator Fusion (ROF) [27]. Batch-based execution allows the DBMS to amortize overhead of PCQ indirection while retaining the performance benefits of JIT code. It also provides LLVM an opportunity to auto-vectorize generated code.

4 SUPPORTED QUERY OPTIMIZATIONS

We now present the optimization categories that are possible with PCQ. As described above, the DBMS generates execution code for a query in a manner that allows it to modify its behavior at runtime. The core idea underlying PCQ is that the generated code supports the ability to permute or selectively enable operations within a pipeline whenever there could be a difference in performance of those operations. These operations can be either short-running, fine-grained steps (e.g., a single predicate) or more expensive relational operators (e.g., joins). These optimizations are independent of each other and do not influence the behavior of other optimizations in either the same pipeline or other pipelines for the query.

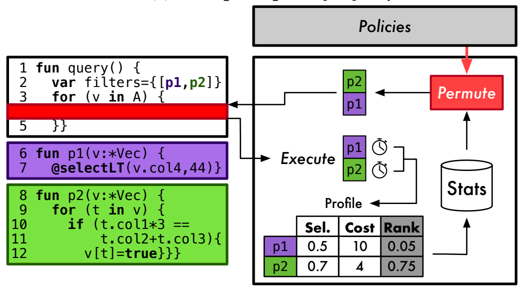
For each category, we describe what changes (if any) the DBMS's optimizer makes to a query's plan and how the Translator organizes the code to support runtime permutations. We also discuss how the DBMS collects metrics about that it uses for policy decisions.

4.1 Filter Reordering

The first optimization is the ability to modify the evaluation order of predicates during a scan operation. The optimal ordering strikes a balance between selectivity and evaluation time: applying a more

```
SELECT * FROM A WHERE col1 * 3 = col2 + col3 AND col4 < 44
```

(a) Example Input SQL Query



```
1 fun query() {
2   var filters={[p1,p2]}
3   for (v in A) {

5   }}
```

```
6 fun p1(v:*Vec) {
7   @selectLT(v.col4,44)}
```

```
8 fun p2(v:*Vec) {
9   for (t in v) {
10    if (t.col1*3 ==
11          t.col2+t.col3){
12        v[t]=true}}}
```

	Sel.	Cost	Rank
p1	0.5	10	0.05
p2	0.7	4	0.75

(b) Generated Code and Execution of Permutable Filter

Figure 3: Filter Reordering – The Translator converts the query in (a) into the TPL on the left side of (b). This program uses a data structure template with query-specific filter logic for each filter clause. The right side of (b) shows how the policy collects metrics and then permutes the ordering.

selective filter first will discard more tuples, but it may be expensive to run. Likewise, the fastest filter may discard too few tuples, causing the DBMS to waste cycles applying subsequent filters.

Preparation / Code-Gen: The first step is to prepare the physical plan to support reordering. The DBMS normalizes filter expressions into their disjunctive normal form (DNF). An expression in DNF is composed of a disjunction of summands, $s_1 \vee s_2 \vee \ldots s_M$. Each summand, s_i, is a conjunction of factors, $f_1 \wedge f_2 \wedge \ldots f_N$. Each factor constitutes a single predicate in the larger filter expression (e.g., `col4` < 44). The DBMS can reorder factors within a summand, as well as summands within a DNF expression. Thus, there are $R = M!N!$ possible overall orderings of a filter in DNF.

Decomposing and structuring filters as functions has two benefits. First, it allows the DBMS to explore different orderings without having to recompile the query. Re-arranging two factors incurs negligible overhead as it involves a function pointer swap. The second benefit is that the DBMS utilizes both code-generation and vectorization where each is best suited. The system implements complex arithmetic expressions in generated code to remove the overhead of materializing intermediate results, while simpler predicates fall back to a library of ~250 vectorized primitives.

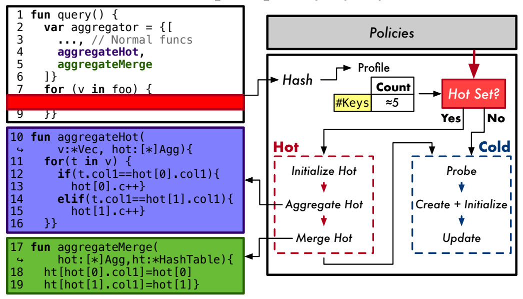
Since the **WHERE** clause in fig. 3a is in DNF, the query requires no further modification. Next, the Translator generates a function for each factor in the filter that accepts a tuple vector as input. In fig. 3b, `p1` (lines 6–7) and `p2` (lines 8–12) are generated functions for the query's conjunctive filter. `p1` calls on a builtin vectorized selection primitive, while `p2` uses fused tuple-at-a-time logic.

Lastly, line 2 in fig. 3b initializes a runtime data structure with a list of filter functions. This structure encapsulates the filtering and permutation logic. A sequential scan is generated over `A` on line 3 using a batch-oriented iteration loop, and the filter is applied to each tuple batch in the table on line 4.

Runtime Permutation: Given the array of filter functions created previously during code-generation, this optimization seeks to order them to minimize the filter's evaluation time. This process is illustrated in fig. 3b. When the DBMS invokes the permutable filter on an input batch, it decides whether to recollect statistics on each filter component. The frequency of collection and precisely what data to collect are configurable policies. A simple approach that we

```
SELECT col1, COUNT(*) FROM A GROUP BY col1
```

(a) Example Input SQL Query



```
1 fun query() {
2   var aggregator = {[
3     ..., // Normal funcs
4     aggregateHot,
5     aggregateMerge
6   ]}
7   for (v in foo) {

9   }}
```

```
10 fun aggregateHot(
 ↪      v:*Vec, hot:[*]Agg){
11   for(t in v) {
12     if(t.col1==hot[0].col1){
13       hot[0].c++}
14     elif(t.col1==hot[1].col1){
15       hot[1].c++}
16   }}
```

```
17 fun aggregateMerge(
 ↪      hot:[*]Agg,ht:*HashTable){
18   ht[hot[0].col1]=hot[0]
19   ht[hot[1].col1]=hot[1]}
```

	Count
#Keys	≈5

(b) Generated Code and Execution of Adaptive Aggregation

Figure 4: Adaptive Aggregations – The input query in (a) is translated into TPL on the left side of (b). The right side of (b) steps through one execution of PCQ aggregation.

use is to sample selectivities and runtimes randomly with a fixed probability p. We explore the effect of p in sec. 5.

If the policy chooses not to sample selectivities, the DBMS invokes the filtering functions in their current order on the tuple batch. Functions within a summand incrementally filter tuples out, and each summand's results are combined together to produce the result of the filter. If the policy chooses to re-sample statistics, the DBMS executes each predicate on all input tuples and tracks their selectivity and invocation time to construct a *profile*. The DBMS uses a predicate's *rank* as the metric by which to order predicate terms. The rank of a predicate accounts for both its selectivity and its evaluation costs, and is computed as $\frac{1-s}{c}$, where s specifies the selectivity of the factor, and c specifies the per-tuple evaluation cost. After rank computation, the DBMS stores the refreshed statistics in an in-memory statistics table. It then reorders the predicates using both their new rank values and the filter's permutation policy.

When the policy recollects statistics, the DBMS evaluates all filters to capture their true selectivities (i.e., no short-circuiting). This means the DBMS performs redundant work that impacts query performance. Therefore, policies must balance unnecessary work with the ability to respond to shifting data skew quickly.

4.2 Adaptive Aggregations

The next optimization is to extract "hot" group-by keys in hash-based aggregations and generate a separate code path for maintaining their values that do no probe the hash table. Hash aggregations are composed of five batch-oriented steps: (1) hashing, (2) probing, (3) key-equality check, (4) initialization, and (5) update. Parallel aggregations require an additional sixth step to merge thread-local partial aggregates into a global aggregation hash table. The Translator generates custom code for aggregate initialization, update, and merging because these are often computationally heavy and query-specific. The other steps rely on vectorized primitives.

We now present PCQ adaptive aggregations that exploit skew in the grouping keys.

Preparation / Code-Gen: The Translator first creates a specialized function to handle the hot keys. This function, `aggregateHot` on lines 10–16 in fig. 4, takes a batch of input tuples and an array of N aggregate payload structures for the extracted hot keys. Each element in the array stores both the grouping key and the running aggregate value. The policy determines the size of N. For east of illustration, we choose to extract two heavy-hitter keys. The Translator generates a loop to iterate over each tuple in the batch and checks for a key-equality match against one of the keys in the hot array. As N is a query compile-time constant, the Translator generates N conditional branches. Tuples that find a match update their aggregates according to the query; others fall through to the "cold" key code path.

Next, the Translator generates a merge function, `aggregateMerge` on lines 17–19, that takes a list of partially computed aggregates and merges them into the hash table. As before, because N is a compile-time constant, the Translator unrolls and inlines the merge logic for the N aggregates into the function.

Finally, in the main query processing function, the Translator creates the data structure (`aggregator`) on lines 2–6 and injects it with pointers to generated functions encapsulating each step in the aggregation, including the new functions to exploit key skew.

Runtime Permutation: Aggregation proceeds similarly as it would without any optimization, but with one adjustment. While computing the hash values of grouping keys in a batch, the DBMS also tracks an approximate distinct key count using HyperLogLog (HLL) [15]. Collecting this metric is inexpensive since HLLs have a compact representation and incur minimal computational overhead in comparison to the more complex aggregation processing logic. After hashing all tuples, if the HLL estimates fewer than N unique grouping keys in the input batch, we follow the optimized pipeline.

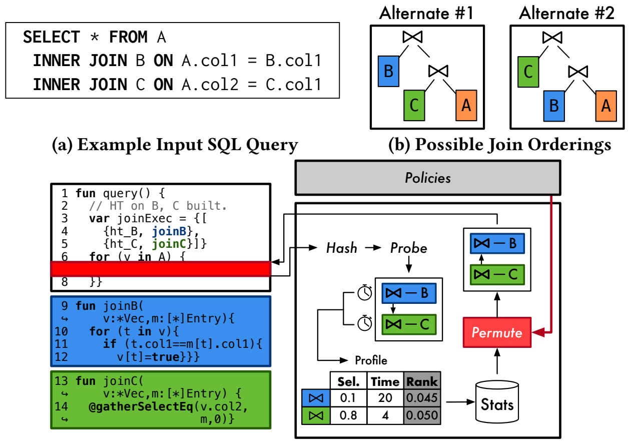
In the optimized flow, the DBMS first allocates an array of aggregate values. It initializes this array with the hottest keys in the current batch. The method for identifying these keys is defined by the system's configured policy. A simple policy is to use the first N unique keys in the batch. A more sophisticated option is to randomly sample from within the current batch until N unique keys are found. In this work, we use the former as we found it offers the best performance versus cost trade-off.

After initializing the hot aggregates array, the DBMS invokes the optimized aggregation function. On return, partially aggregated data is merged back into the hash table using the merging function. Since HLL estimations have errors, it is possible for some tuples to not find a match in the hot set. In this case, the batch is processed using the cold path as well. Thus, there is a risk of an additional pass, but the DBMS mitigates this by tuning the HLL estimation error. Supporting parallel aggregation requires neither a modification to the algorithm described earlier, or the generation of additional code. Each execution thread performs thread-local aggregation as before.

4.3 Adaptive Joins

A PCQ DBMS optimizes hash joins by (1) tailoring the hash table implementation based on runtime information and (2) reordering the application of joins in right- or left-deep query plans. We discuss data structure specialization before describing the steps required during code-generation and runtime to implement join reordering. We use the convention that the left input to a hash join is the build side, and the right input is the probe side.

Hash table construction proceeds in two phases. First, the DBMS materializes the tuples from the left join input into a thread-local memory buffer in row-wise format along with the computed hash



```
SELECT * FROM A
  INNER JOIN B ON A.col1 = B.col1
  INNER JOIN C ON A.col2 = C.col1
```

(a) Example Input SQL Query

(b) Possible Join Orderings

```
1  fun query() {
2    // HT on B, C built.
3    var joinExec = {[
4      {ht_B, joinB},
5      {ht_C, joinC}]}
6    for (v in A) {

8    }}
9  fun joinB(
 ↪      v:*Vec,m:[*]Entry){
10   for (t in v){
11     if (t.col1==m[t].col1){
12       v[t]=true}}}
13 fun joinC(
 ↪      v:*Vec,m:[*]Entry) {
14   @gatherSelectEq(v.col2,
 ↪                  m,0)}
```

	Sel.	Time	Rank
⋈	0.1	20	0.045
⋈	0.8	4	0.050

(c) Generated Code and Execution of Permutable Joins

Figure 5: Adaptive Joins – The DBMS translates the query in (a) to the program in (c). The right side of (c) illustrates one execution of a permutable join that includes a metric collection step.

of the join columns. The DBMS also tracks an approximate count of unique keys using an HLL estimator. Once the left join input is exhausted, the DBMS uses HLL to estimate the hash table size. If the estimated size is smaller than the CPU's L3 cache capacity, the DBMS constructs a *concise hash table* (CHT [31]); otherwise, it constructs a bucket-chained hash table with pointer-tagging [22]. With this, the DBMS is able to perfectly size the hash table, thereby eliminating the need to resize during construction. Furthermore, deferring the choice of table implementation to runtime allows the DBMS to tune itself according to the data distribution. In the second phase, each execution thread scans its memory buffers to build a global hash table. If a bucket-chained hash table was selected, pointers to thread-local tuples are inserted using atomic compare-and-swap instructions. If a CHT was selected, a partitioned build is performed as described in [31]. We now describe how to implement permutable joins using fig. 5 as the running example.

Preparation / Code-Gen: The DBMS's optimizer supports permutable joins in right-deep query plans containing consecutive joins, as in fig. 5a. The system designates one table as the "driver" that it joins with one or more tables (i.e., one per join). The DBMS may use either hash or index joins depending on the selected access method. The DBMS applies the joins in any order regardless of the join type (i.e., inner vs. outer) since each driver tuple is independent of other tuples in the table and intermediate iteration state is transient for a batch of tuples. In fig. 5b, the DBMS can join the tuples in `A` either against `C` or `B` first. The best ordering may change over the duration of a query on a per-block basis due to variations in data distributions. Our implementation in NoisePage has an additional requirement that the driver table contains all key columns required across all joins.

During code generation, the Translator first generates one key-check function per join. In fig. 5c, `joinB` (lines 9–12) and `joinC` (lines 13–14) are the key-check functions for joining tuples from `A` against tables `B` and `C`, respectively. These functions take in a vector of input tuples and a vector of potential join candidates, and then evaluates the join predicate for each tuple. As described earlier, the DBMS may implement these functions either by dispatching to vectorized primitives or using tuple-at-a-time logic directly in bytecode. In the example, `joinC` uses a built-in primitive to perform a fused gather and select operation with SIMD instructions.

Next, the Translator constructs a data structure (*joinExec* on lines 3–5) in the pipeline to manage the join and permutation logic. This structure requires three inputs for each join: (1) a pointer to the hash table to probe, (2) a list of attribute indexes forming the join key, and (3) a pointer to the join's key-check function. Finally, the Translator generates the scan code for `A` on lines 6–8 and the invocation of the join executor for each tuple batch on line 7.

Runtime Permutation: During execution, the DBMS first computes a hash value for each tuple in the input batch. Next, a policy decision is made whether to recollect statistics on each join. Assuming the affirmative, the DBMS then probes each hash table.

The probing process is decomposed into two steps. Since hash tables embed Bloom filters, the DBMS performs the combined lookup and filter operation using only the hash values computed in the previous step. The second step invokes each join's key-equality function to resolve false positives from the first step. The DBMS ensures that only tuples that pass previous joins are processed in the remaining joins. After completion, the system creates a profile that captures selectivity and timing information for each join step. Similar to filters, the DBMS saves the profile to its internal catalog and then permutes the join according to the policy.

5 EVALUATION

We now present an analysis of the PCQ method and corresponding system architecture. We implemented our PCQ framework and execution engine in the NoisePage DBMS [4]. NoisePage is a PostgreSQL-compatible HTAP DBMS that uses HyPer-style MVCC [29] over the Apache Arrow in-memory columnar data [25]. It uses LLVM (v9) to JIT compile our bytecode into machine code.

We performed our evaluation on machine with 2 × 10-core Intel Xeon Silver 4114 CPUs (2.2GHz, 25 MB L3 cache per-core, with AVX512) and 128 GB of DRAM. We ensure that the DBMS loads the entire database into the same NUMA region using `numactl`. We implemented our microbenchmarks using the Google Benchmark [2] library which runs each experiment a sufficient number of iterations to get a statistically stable execution times.

We begin by describing the workloads that we use in our evaluation. We then measure PCQ's ability to improve the performance of compiled queries. We execute these first experiments using a single thread to minimize scheduling interference. Lastly, we present a comparison of NoisePage on multi-threaded queries with PCQ against two state-of-the-art OLAP DBMSs.

5.1 Workloads

We first describe the three workloads that we use in our evaluation:

Microbenchmark: We created a synthetic benchmark to isolate and measure aspects of the DBMS's runtime behavior. The database contains six tables (`A`–`F`) that each contain six 64-bit signed integer columns (`col1`–`col6`). Each table contains 3m tuples and occupies 144 MB of memory. For each experiment that uses this benchmark, we vary the distributions and correlations of the database's columns' values to highlight a specific component. The workload contains three query types that each target a separate optimization from sec. 4: (1) a scan query with three predicates, (2) an aggregation query with groupings, and (3) a multi-way join query.

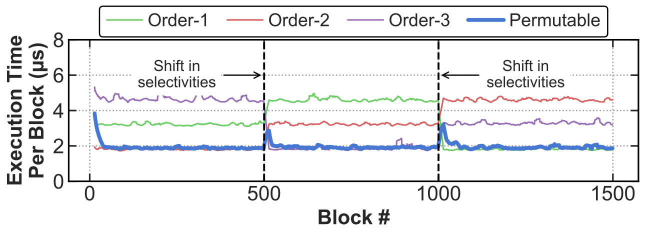


Figure 6: Performance Over Time – Execution time of three static filter orderings and our PCQ filter during a sequential table scan.

TPC-H: This is a decision support system workload that simulates an OLAP environment [37]. It contains eight tables in 3NF schema. We use a scale factor of 10 (~10 GB). To better represent real-world applications, we use a skewed version of the TPC-H generator [5]. We select nine queries that cover the TPC-H choke-point categories [8] that vary from compute- to memory/join-intensive queries. Thus, we expect our results to generalize and extend to the remaining queries in the benchmark.

Star Schema Benchmark (SSB): This workload simulates a data warehousing environment [30]. It is based on TPC-H, but with three differences: (1) it denormalizes the two largest tables (i.e., `LINEITEM` and `ORDERS`) into a single new fact table (i.e., `LINEORDER`), (2) it drops the `PARTSUPP` table, and (3) it creates a new `DATE` dimension table. SSB consists of thirteen queries and is characterized by its join complexity. We use a scale factor of 10 (~10 GB) using the default uniformly random data generator.

5.2 Filter Adaptivity

We begin with evaluating PCQ's ability to optimize and permute filter ordering in response to shifting data distributions. We use the microbenchmark workload with a **SELECT** query that performs a sequential scan over a single table:

```
SELECT * FROM A
 WHERE col1 < 1000 AND col3 < 1000 AND col3 < 3000
```

The constant values in the **WHERE** clause's predicates enable the data generators in each experiment to target a specific selectivity.

Performance Over Time: The first experiment evaluates the performance of PCQ filters during a table scan as we vary the selectivity of individual predicates. We populate each column such that one of the predicates has a selectivity of ~2% while the remaining two have 98% selectivity each. We alternate which predicate is the most selective over disjoint sections of the table. That is, for the first 500 blocks of tuples, the predicate on `col1` is the most selective. Then for the next 500 blocks, the predicate on `col2` is the most selective. Thus, each predicate is optimal for only $\frac{1}{3}$ of the table.

We execute this query with PCQ's permutable filters configured using a 10% sampling rate policy (i.e., the DBMS will collect metrics per block with a 10% probability). We also execute the query using three "static" orderings that each evaluate a different predicate first. These static orderings represent how existing JIT compilation-based DBMSs execute queries without permutability.

The results in fig. 6 show the processing time per block during the scan. Each of the static orderings is only optimal for a portion of the table, while PCQ discovers new optimal orderings after

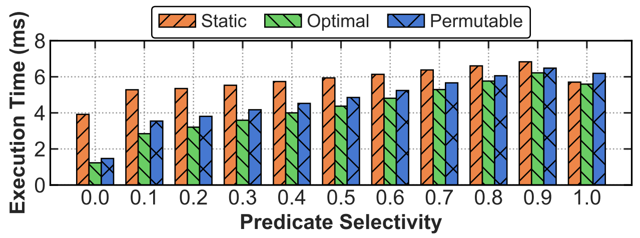


Figure 7: Varying Predicate Selectivity – Performance of the static, optimal, and permutable orderings when varying the overall query selectivity.

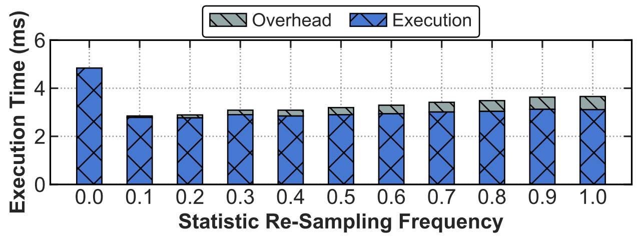


Figure 8: Filter Permutation Overhead – Performance of the permutable filter when varying the policy's re-sampling frequency and fixing the overall predicate selectivity to 2%.

each transition. During the transition periods after the distribution changes at blocks #500 and #1000, PCQ initially incurs some jitter in performance as it executes the previously superior ordering. Within ten blocks of data, PCQ re-samples selectivity metrics and permutes itself to the optimal ordering. Overall, the PCQ query is ~2.5× faster than any of the static plans.

Varying Predicate Selectivity: We next analyze the permutable filter optimization across a range of selectivities to measure its robustness. For this experiment, we modify the table's data distribution for the above query such that the filters' combined selectivity varies from 0% (i.e., no tuples are selected) to 100% (i.e., all tuples are selected). As before. the DBMS uses a 10% sampling rate policy. We compare against a "static" ordering as chosen by the DBMS's query optimizer based on collected statistics. We also execute an "optimal" configuration where we provide the DBMS with the best ordering of the filters on a per-block basis. This optimal plan represents the upper bound in performance that the DBMS could achieve without the re-sampling overhead.

The results in fig. 7 show that PCQ is competitive (within 20%) of the optimal configuration across all selectivities. Our second observation is that both optimal and PCQ consistently outperform the static ordering provided by the DBMS below 100% selectivity. At 0%, PCQ and optimal are 2.7× and 3.6× faster than static, respectively. This is because each is able to place the most selective term first in the evaluation order. As the filter selectivity increases, the execution times of all configurations also increase since the DBMS must process more tuples . At 100% selectivity, the PCQ filter performs the worst because it suffers from sampling overhead; if all tuples are selected, adaptivity is not required. Finally, all orderings perform better at 100% selectivity than at 90% because the DBMS has optimizations that it only enables when vectors are full.

Filter Permutation Overhead: As described in sec. 4.1, there is a balance between the DBMS's metric collection frequency and its impact on runtime performance. To better understand this trade-off, we next execute the **SELECT** query with different re-sampling frequencies. We vary the frequency from 0.0 (i.e., no sampling) to 1.0 (i.e., the DBMS samples and re-ranks predicates after accessing each block). We fix the combined selectivity of all the filters to 2% and vary which filter is the most selective at blocks #500 and #1000 as in fig. 6. The query starts with the Order-3 plan from fig. 6 as this was the static ordering with the best overall performance. We instrument the DBMS to measure the time spent in collecting the performance metric data versus query execution.

The results, shown fig. 8, demonstrate the non-linear relationship between metric collection frequency and performance. Disabling sampling removes any overhead, but incurs a ~1.7× slow-down compared to permutable filters because the DBMS cannot react to fluctuations in data distributions. Sampling on every block (i.e., 100% sampling rate) adds 15% overhead to execution time. The DBMS performs best with the 0.1 sampling rate and thus, we use this setting for the remaining experiments.

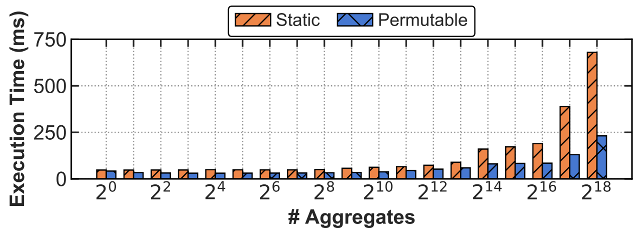


Figure 9: Varying Number of Aggregates – Performance of the adaptive aggregation as we vary the total number unique aggregate keys.

5.3 Aggregation Adaptivity

We next evaluate PCQ's ability to exploit skew when performing hash aggregations. Unless otherwise specified, the experiments in this section use the microbenchmark workload with a **SELECT** query that performs a hash aggregation over a single table:

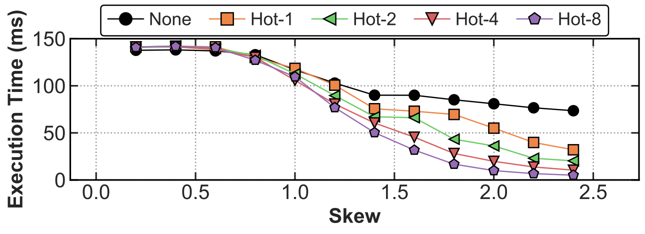
```
SELECT col1, SUM(col2), SUM(col3), SUM(col4)
  FROM A GROUP BY col1
```

We modified the workload's data generator to skew the grouping key (col1) to highlight a specific component of the system.

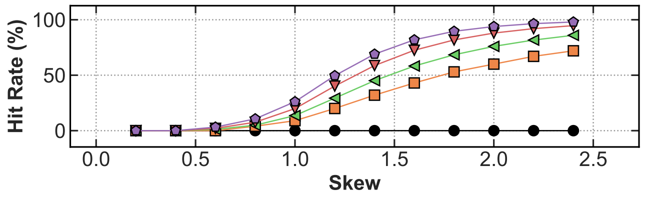
Varying Number of Aggregates: We first measure the performance of PCQ aggregations as we vary the total number of unique aggregate keys in the benchmark table. We populate the grouping key column (col1) with values from a random distribution to control the number of unique keys per trial. The data for the columns that are used in the aggregation functions (col2–col4) are chosen randomly (uniform) from their respective value domains.

We configured the PCQ framework to use five heavy-hitter keys. The choice of five is tunable for the DBMS, but fixed in this experiment. We also execute a static plan that fuses the table scan with the aggregation using a data-centric approach [28]. The static plan represents how existing JIT-based DBMSs execute the query.

The results are shown in fig. 9. When the number of aggregates is small (i.e., <16k keys), the hash table fits in the CPU cache and PCQ outperforms the static configuration. When there are fewer than five keys, PCQ routes updates through the "hot" path yielding a 1.5× improvement. Beyond this threshold, PCQ falls back to its hybrid vectorized and JIT implementation, outperforming the static



(a) Performance



(b) Hit Rate

Figure 10: Varying Aggregation Skew – Performance of PCQ's adaptive aggregation when increasing skew in aggregate keys with a fixed number of keys. (a) shows the total execution time to perform the aggregation. (b) shows the percentage of input tuples that hit a heavy-hitter branch.

plan by 1.6×. PCQ fairs well even at high cardinality because (1) it performs data-independent random accesses into the hash table and (2) both pre-compiled and generated aggregation steps are auto-vectorized. The benefits of data-independent memory access and auto-vectorization are most pronounced when the hash table exceeds the CPU's LLC. fig. 9 shows that this occurs at ~256k keys where PCQ is 3× faster than the static plan.

Varying Aggregation Skew: The DBMS must be careful when deciding how many heavy-hitter keys to extract into specialized JIT code for permutable aggregations. Extracting more keys (1) introduces the possibility of branch mispredictions that increase runtime, and (2) generates larger functions that increase compilation time.

To explore the relationship between the size of the heavy-hitter key set and performance, we execute the same **SELECT** query as before, but fix the total number of unique grouping keys to 200k. We use a skewed Zipfian distribution for the grouping keys and execute the query using PCQ in configurations that extract zero to eight heavy-hitter keys from the aggregation hash table. We measure both the query execution time and the percentage of tuples that hit one of the conditional branches for an extracted key.

The results in fig. 10a show that the configurations are within 3% of each other for low skew values. None performs the best since the others introduce untaken branch instructions due to the uniformity in the key distribution. As skew increases, the versions that extract keys perform better. The benefit of this optimization plateaus with increasing skew as the DBMS hits the memory bandwidth limits of the system. None uses the bucket-chained hash table while other versions update aggregates stored in plain arrays. At a skew level of 2.4, the Hot-8 configuration is 18× faster than None.

fig. 10b shows the percentage of tuples that match a hot key in the optimized aggregation function. With low skew (i.e., below 1.0), 10% of the tuples take a heavy-hitter branch; the remaining suffer the branch misprediction and fall back to the cold key path. Cost mispredictions are the reason why the optimized plans perform worse at a lower skew. At a higher skew the optimized versions absorb more updates that bypass the hash table, resulting in fewer cycles-per-tuple. At skew level 1.6, Hot-1 incurs a 45% hit rate, while Hot-8's rate is 82%. At the highest skew (2.4), the min/max hit rates are 72% and 98%, respectively; this explains the performance improvements in the optimized plans.

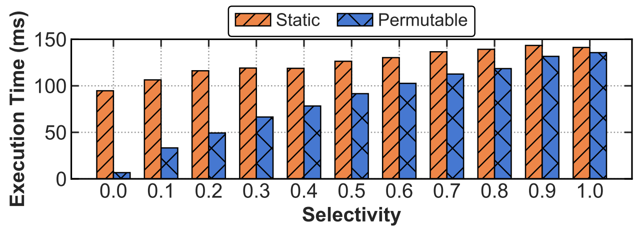


Figure 11: Varying Join Selectivity – Execution time to perform three hash-joins while varying overall join selectivity.

5.4 Join Adaptivity

We evaluate PCQ's ability to optimize hash join operations in response to changing data distributions. Each experiment constructs a right-deep join tree that builds hash tables in separate pipelines and probes them in the final pipeline. The experiments customize the data generation for each join key to target a specific join selectivity across any pair of tables, along with the overall selectivity.

Varying Join Selectivity: This experiment performs two inner hash joins between three microbenchmark workload tables:

```
SELECT * FROM A
  INNER JOIN B ON A.col1 = B.col1
  INNER JOIN C ON A.col2 = C.col1
```

We tailor the data generation for the above join attributes to achieve a desired selectivity from 0% (i.e., no tuples find join partners) to 100% (i.e., all tuples find join partners). We execute the join using a static ordering that reflects the join order provided by the DBMS. Moreover, the implementation uses a fused tuple-at-a-time processing model as would be generated by HyPer's JIT compilation-based engine [28]. We also execute the PCQ join with a 10% re-sampling rate policy. The PCQ variant starts with the same initial join ordering as provided to the static option.

fig. 11 shows that at 0% selectivity, the PCQ join performs ~14× better than the static join. This is because PCQ discovers and permutes the joins into their optimal order within ten blocks of processing the probe input. As the selectivity of the join increases, the need for permutability decreases since the DBMS must process more tuples. At 100% selectivity, PCQ betters the static plan since it vectorizes the hashing, probing, and key-equality steps.

Varying Number of Joins: We now evaluate PCQ's performance executing a multi-step join. For this experiment, we vary the number of join operations (i.e., one to five hash joins) in the query, but keep the overall query selectivity at 10%. Although permutability is unnecessary with only a single join, we include it here for completeness. The NoisePage engine elides permutable joins in such scenarios. We execute a similar **SELECT** query as in the previous experiment, but append additional join clauses and project in all table columns.

The results in fig. 12 show that PCQ performs 1.15× faster than the static plan even with a single join. This is because PCQ employs vectorized hash and probe routines, and benefits from LLVM's

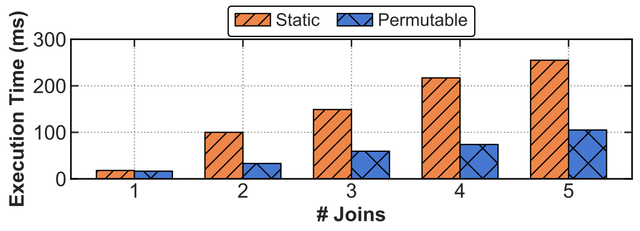


Figure 12: Varying Number of Joins – Execution time to perform a multi-step join while keeping the overall join selectivity at 10%.

auto-vectorization of the key-equality check function. Although the overall selectivity is constant, as the number of joins increase, PCQ outperforms the static plan by discovering the most selective joins and dynamically reordering them earlier in processing. PCQ is 3× faster than static when performing two joins, and 2.5× faster when performing greater than three joins.

5.5 System Comparison

Lastly, we compare NoisePage with and without PCQ against two state-of-the-art in-memory databases: Actian Vector (v9.2) and Tableau HyPer (v5.1). Vector [1] is a columnar DBMS based on MonetDB/x100 [9] that uses a vectorized execution engine comprised of SIMD-optimized primitives. We modified Vector's configuration to fully utilize system memory and CPU threads for parallel execution. HyPer [3] is a columnar DBMS that uses the LLVM to generate tuple-at-a-time query plans that are either interpreted or JIT compiled. The version of HyPer we use also supports SIMD predicate evaluation. After consulting with Tableau's engineers, we did not modify any configuration options for HyPer.

In this section we evaluate the TPC-H and SSB benchmarks. After loading the data into each system, we run their requisite statistics collection and optimization operations. We warm each DBMS by running the workload queries once before reporting the average execution time over five consecutive runs. We make a good faith effort to ensure the DBMSs execute equivalent query plans by manually inspecting them. We note, however, that the DBMSs include additional optimizations that are not present in all systems. For NoisePage, we use the query plan generated by HyPer's optimizer.

5.5.1 Skewed TPC-H. We first evaluate the TPC-H benchmark using Microsoft's skewed data generator [5], using a skew of 2.0 (i.e., high-skew). The results are shown fig. 13. We also show the effect of each optimization in table 1. Each cell shows the relative speedup of enabling the associated optimization atop all previous optimizations. Numbers close to 1.0 mean the optimization had little impact, while large numbers indicate greater impact. Gray (i.e., blank) entries signify that the optimization was not applied.

Q1: This query computes five aggregates over four group-by keys in a single table. Increased skew affects the distribution among the four grouping keys. The hottest grouping key pair receives 49% of the updates when there is no skew, and 86% with significant skew. NoisePage's PCQ aggregation optimization is triggered resulting in a 1.7× improvement since the bulk of processing time is spent performing the aggregation. Although NoisePage with PCQ is 4.8× faster than Vector, it is 1.2× slower than HyPer. We believe this is due to HyPer's use of fixed-point arithmetic which is faster than the floating-point math used in NoisePage.

Q4: This query computes a single aggregate over five group-by keys (triggering the PCQ aggregation optimization), and contains a permutable filter on `ORDERS`. The selectivity of the range predicate on `o_orderdate` is 0.08% with high skew. NoisePage with PCQ flips the range predicate and applies the aggregation optimization resulting in a 2× improvement over both NoisePage without PCQ and commercial systems. table 1 shows that the bulk of the benefit is attributed to the optimized aggregation.

Q5: This query joins six tables, but contains only two permutable joins. The final aggregation computes one summation on two group-by keys, which triggers the PCQ aggregation optimization. This query also contains vectorizable predicates that are supported by all DBMSs. In NoisePage, the benefit of permutable filters is modest, while the optimized aggregation leads to a 1.33× improvement over the baseline. The two permutable joins are never rearranged, hence there is no improvement from PCQ joins. Overall, NoisePage with PCQ is 3× faster than HyPer and 5× faster than Vector.

Q6: The performance of Q6 depends on the DBMS's implementation of the highly selective (0.05%) filter over `LINEITEM`. We note that increased skew does not affect the ordering of the `LINEITEM` predicate. Thus, NoisePage's PCQ permutable filter adds minor overhead resulting in 4% slowdown over the baseline. This is a direct result of resampling with a fixed probability, and can be remedied by using a more advanced sampling policy. All systems leverage SIMD filter evaluation with comparable performance.

Q7: This is a join-heavy query where HyPer chooses a bushy join plan that is 4× slower than a right-deep plan. Although no tuples reach the final aggregation, PCQ flips the application order of the range predicate on `l_shipdate` resulting in a 1.2× improvement.

Q11: This query also contains five joins, but none are permutable. It also contains two separate aggregations, but whose cardinalities never trigger the PCQ optimizations. Finally, it contains multiple vectorizable predicates, but all have single terms making permutation unnecessary. Thus, Q11 represents a query where none of the PCQ optimizations are tripped. We include it to show that PCQ incurs negligible overhead, and to serve as an example of where an optimizer can assist in identifying better plans in the presence of data skew. NoisePage (with an without PCQ) offers comparable performance to HyPer, and is 4× faster than Vector.

Q16: This query has a right-deep join pipeline using `PARTSUPP` as the driver, a multi-part filter on `PART` and a hash aggregation. The cardinality of the aggregation exceeds the optimization threshold (i.e., five). PCQ reorders the `PART` filters, yielding a boost of almost 1.2×. Next, PCQ reorders the join to use SIMD gathers due to the size of the build table, which improves performance by 1.2×. NoisePage with PCQ is 7.4× and 3× faster than HyPer and Vector, respectively. HyPer chooses a worse plan at high-skew: it decides on a left anti-join rather than a right anti-join. We believe that HyPer's performance would improve with a better plan.

Q18: Like Q16, this query also contains a right-deep join pipeline using `ORDERS` as the driver. Additionally, there is an aggregation, but whose cardinality exceeds the optimization's threshold. PCQ reorders the joins in order to utilize SIMD gathers on the smaller table resulting in a 1.19× improvement over the baseline. Interestingly, HyPer chooses a worse query plan at high skew, using

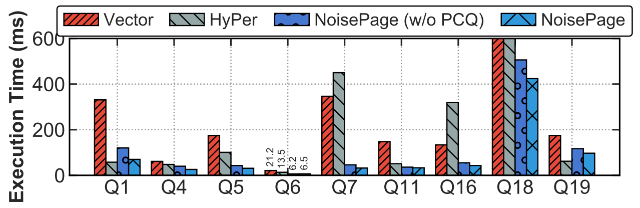


Figure 13: System Comparison on Skewed TPC-H – Evaluation of NoisePage, HyPer, and Vector on the *skewed* TPC-H benchmark.

Query	+Filters (§4.1)	+Aggregations (§4.2)	+Joins (§4.3)
Q1	–	1.71	–
Q4	1.05	1.54	–
Q5	1.08	1.33	1.00
Q6	0.96	–	–
Q7	1.02	1.40	1.00
Q11	–	1.02	–
Q16	1.18	1.00	1.00
Q18	–	1.00	1.19
Q19	1.21	–	–

Table 1: TPC-H Speedup – The speedup achieved when incrementally applying each PCQ optimization to TPC-H queries.

a right-semi join instead of a left semi-join, resulting in a 2.6× slowdown compared to PCQ.

Q19: This query contains an inner join between PART and LINEITEM followed by a complex disjunctive filter and a static aggregation. NoisePage with PCQ reorders the predicate on LINEITEM to improve performance by 1.2× over the baseline. HyPer performs the best, completing 1.2× and 2.5× quicker than NoisePage and Vector, respectively. We attribute NoisePage's degradation to costly transformations between internal representations for "selected" tuples when utilizing both tuple-at-a-time and vectorized filter logic.

5.5.2 Star Schema Benchmark. This experiment evaluates all systems on the Star Schema Benchmark [30]. The overall results are shown fig. 14 along with the benefit breakdown in table 2. The thirteen SSB queries are grouped into four categories. Each category contains structurally equivalent queries, but differ in their filtering and aggregating terms. Thus, we discuss the results by groups since the behavior of one query generalizes to all queries in the same category. Unlike the previous evaluation with TPC-H, we execute NoisePage with PCQ using a random initial plan to demonstrate the benefit of our approach; NoisePage without PCQ uses the optimal plan generated by HyPer.

Q1.*: All queries in this category contain a single join between the smallest and largest tables in the database, and contain selective multi-part filters on both tables. Since there is a single join, PCQ joins yield no benefit. However, PCQ rearranges some of the filtering terms resulting in a minor performance benefit. HyPer performs the best, running 1.7× and 3.7× faster than NoisePage and Vector, respectively. This is because it performs SIMD vectorized filter evaluation on compressed data, achieving a better overall CPI.

Q2.*: These queries contain three joins and an aggregation. Although starting with a random join order, PCQ permutes joins during execution based on observed selectivities and runtime conditions resulting in a mean improvement of ~1.5× over the baseline. We observe a minor performance degradation when applying the

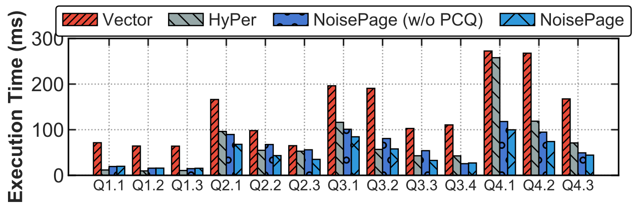


Figure 14: System Comparison on the Star-Schema Benchmark – Evaluation of NoisePage, HyPer, and Vector on the SSB.

Query	+Filters (§4.1)	+Aggregations (§4.2)	+Joins (§4.3)
Q1.1	1.00	1.00	1.00
Q1.2	1.02	1.00	1.00
Q1.3	1.06	1.00	1.00
Q2.1	0.96	1.00	1.32
Q2.2	0.99	1.00	1.56
Q2.3	1.00	1.00	1.60
Q3.1	1.00	1.00	1.20
Q3.2	1.01	1.00	1.42
Q3.3	1.03	1.00	1.69
Q3.4	1.00	1.00	0.92
Q4.1	1.03	1.00	1.19
Q4.2	1.02	1.00	1.33
Q4.3	1.02	1.00	0.98

Table 2: SSB Speedup – The speedup achieved when incrementally applying each PCQ optimization to SSB queries.

PCQ filter optimization due to the overhead of exploration. Since the optimal filter order is unchanged during the query's lifetime, exploring alternate orders is unnecessary. We believe a more sophisticated adaptive policy that adjusts sampling frequency avoids this problem. Overall, NoisePage with PCQ is 1.4× and 2.2× faster than HyPer and Vector, which use fixed query plans.

Q3.*: Similar to Q2, these queries contain three joins and an aggregation, but swaps in one different base table. Only one query (i.e., Q3.4) triggers the PCQ aggregation optimization. As in Q2, PCQ periodically explores the join order space to discover the optimal ordering resulting in an average performance improvement of 1.3× over the baseline. Since the majority of query processing time is spent performing joins, PCQ's aggregation optimization provides limited benefit. Finally, we note that PCQ joins are slower specifically in Q3.4. In this case, the DBMS periodically explores different join orderings (despite observing consistent optimal join rankings), but the overhead of this exploration outweighs the performance benefits. We believe better policy design can ameliorate this problem. Overall, NoisePage with PCQ results in an improvement of 1.3× over the baseline and HyPer, and 3.2× over Vector.

Q4.*: Queries in this category join all five tables in the database. In all but Q4.3, NoisePage with PCQ finds an optimal join and filtering ordering resulting in a ~1.26× improvement over the baseline. Q4.3 sees reduced performance for the same reason as in Q3.4: the PCQ policy forces exploration assuming the benefit is greater than the overhead. That assumption, however, is invalid in Q4.3. Although HyPer and Vector implement filters on compressed data, the bulk of processing time is spent execution joins. Hence, PCQ produces an average improvement of 1.2× over the baseline, and 1.9×, and 3.4× over HyPer, and Vector, respectively.

6 RELATED WORK

Deshpande et al. provides a thorough survey of the AQP methods up to the late 2000s [12]. The high-level idea with AQP is that the DBMS monitors the runtime behavior of a query to determine whether the optimizer's cardinality estimations exceed some threshold. It can then either (1) return to the optimizer to generate a new plan using updated estimates it collected during execution or (2) switch to an alternative sub-plan at an appropriate materialization point. The former is not desirable in a JIT code-gen DBMS because of the high cost of recompilation.

The two AQP methods from the latter category that are most relevant to our PCQ approach are parametric optimization [11, 16] and proactive reoptimization [7]. The parametric optimization method for the Volcano optimizer generates multiple plans for a pipeline and embeds them using *choose-plan* operators that allow the DBMS to change which pipeline plan to use during query execution based on the observed cardinalities. Similarly, proactive reoptimization introduced in the Rio optimizer added *switch* operators in plans that allow the DBMS to choose between different sub-plans within a pipeline [7]. Rio also supports collecting statistics during query execution. Plan Bouquets [14] generates a "parametric optimal set of plans" that it switches between at runtime, but also provides a worst-case performance bounds. All of these methods are similar to our approach except they target interpretation-based DBMS architectures. They also generate non-permutable plans that only support coarse-grained switching between sub-plans before the system executes them. PCQ, on the other hand, enables strategy switching within a pipeline while the DBMS is actively executing it. Perron et al. show that modern cost-based query optimizers continue to underperform for certain classes of queries [33]. Although their proposal targets the DBMS optimizer, PCQ solves many of the same issues during execution.

IBM developed at AQP technique for dynamically reordering joins in pipelined plans [24]. It targets OLTP workloads and does not generalize to analytical queries. More recently, SkinnerDB uses reinforcement learning to approximate optimal join ordering during query execution [38]. It requires, however, expensive pre-processing of data where it computes hash tables for all indexes and currently only supports single-threaded execution.

HyPer's adaptive compilation technique includes many of the building blocks that we use to build a PCQ-enabled DBMS [20]. First, it relies on an interpreter that operates on HyPer-specific bytecode, similar to NoisePage's interpreter. This bytecode is derived from LLVM IR rather than a DSL like TPL. HyPer only adapts its execution mode (i.e., interpreted vs. compilation), and does not modify the high-level structure of query plans, nor does it perform the low-level intra-pipeline optimizations that we described in sec. 4.

Another in-memory DBMS that supports adaptivity is Vector [9]. Instead of JIT compiling queries, Vector uses pre-compiled *primitives* that are kernel functions that perform an operation on a specific data type (e.g., an equality predicate on 32-bit integers). The DBMS then stitches the necessary primitives together to execute each query. Vector's "micro-adaptivity" technique compiles these primitives using different compilers (e.g., gcc, icc), and then uses a multi-armed bandit algorithm to select the best primitive at runtime based on performance measurements [34]. Since this approach only changes what compiler to use, it cannot accommodate plan-wide optimizations or adapt the query plan based on the observed data. Zeuch et al. developed a reoptimization approach using a cost-model based on the CPU's built-in hardware counters. Their framework estimates the selectivities of multi-table queries to adapt execution orderings.

A more recent adaptive approach for JIT compiled systems was proposed for Apache Spark [35]. This method provides dynamic speculative optimizations for compiling data file parsing logic. Grizzly [17] presents an adaptive compilation approach targeting stream processing systems. It initially generates generic C++ code with custom instrumentation to collect profiling information. The runtime uses this profiling information to recompile new optimized variants that it then monitors and verifies using hardware counters. Grizzly supports predicate reordering and domain-value specialization. PCQ supports more optimizations without recompiling plans.

One of the first implementations of reordering predicates was in Postgres from the early 1990s [18]. The authors instrumented the DBMS to collect completion times of predicates during query execution. They then modified Postgres's optimizer to reorder predicates to consider the trade-offs between selectivity and evaluation cost in future queries. This is the same high-level approach that IBM used in its Learning Optimizer (LEO) for DB2 [36]. The DBMS collects runtime information about queries and feeds this data back into the optimizer to improve its planning decisions.

Lastly, Dreseler et al. perform a deep-dive analysis of the TPC-H benchmark queries [13]. Their work groups the canonical chokepoint queries into one of three categories: plan-level, logical operator-level, and engine efficiency. Their conclusion is that predicate placement and subquery flattening were the most relevant to query performance. PCQ supports the former in the execution engine, while the latter is handled by the DBMS optimizer.

7 CONCLUSION

This work presented PCQ, a query processing architecture that bridges the gap between JIT compilation and AQP. With PCQ, the DBMS structures generated code to utilize dynamic runtime structures with a layer of indirection that enables the DBMS to safely and atomically switch between plans while running the query. To amortize the overhead of switching, generated code relies on batch-oriented processing. We proposed three optimizations using PCQ that improve different relational operators. For scans, we proposed an adaptive filter that efficiently discovers an optimal ordering to reduce execution times. For hash-based aggregations, we proposed a dynamic optimization that identifies and exploits skew by extracting heavy-hitter keys out of the hash table. Lastly, we proposed an optimization for left- or right-deep joins that enables the DBMS to reorder their application to maximize performance. Our evaluation showed that NoisePage with PCQ enabled delivers up to 4× higher performance on a synthetic workload and up to 2× higher performance on TPC-H and SSB benchmark workloads.

ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, IIS-1718582, SPX-1822933), Google Research Grants, and the Alfred P. Sloan Research Fellowship program.

REFERENCES

[1] [n.d.]. Actian Vector. http://esd.actian.com/product/Vector.

[2] [n.d.]. Google Benchmark. https://github.com/google/benchmark.

[3] [n.d.]. HyPer. https://hyper-db.de.

[4] [n.d.]. NoisePage. https://noise.page.

[5] [n.d.]. Skewed TPC-H. https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[6] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR*. 238–249.

[7] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive Re-optimization. In *SIGMOD*. 107–118.

[8] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*.

[9] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*.

[10] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A history and evaluation of System R. *Commun. ACM* 24 (October 1981), 632–646. Issue 10.

[11] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. 150–160.

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

[13] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB* 13, 8 (2020), 1206–1220.

[14] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan Bouquets: Query Processing without Selectivity Estimation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. 1039–1050.

[15] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms (DMTCS Proceedings)*. 137–156.

[16] G. Graefe and K. Ward. 1989. Dynamic Query Evaluation Plans. *SIGMOD Rec.* 18, 2 (June 1989), 358–366.

[17] Philipp M Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation *(SIGMOD)*.

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

[19] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *PVLDB* 7, 10 (2014), 853–864.

[20] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. 197–208.

[21] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 613–624.

[22] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*. 743–754.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[24] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *2007 IEEE 23rd International Conference on Data Engineering*. 26–35. https://doi.org/10.1109/ICDE.2007.367848

[25] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wes McKinney, and Andrew Pavlo. 2019. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Under Submission*.

[26] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. 659–670.

[27] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment* 11 (September 2017), 1–13. Issue 1.

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[29] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems *(SIGMOD)*.

[30] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.

[31] R Barber G Lohman I Pandis, V Raman R Sidle, G Attaluri N Chainani S Lightstone, and D Sharpe. 2014. Memory-Efficient Hash Joins. *Proceedings of the VLDB Endowment* 8, 4 (2014).

[32] Drew Paroski. 2016. Code Generation: The Inner Sanctum of Database Performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html.

[33] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*.

[34] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *SIGMOD*. 1231–1242.

[35] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 754–767. https://doi.org/10.14778/3377369.3377382

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

[37] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). http://www.tpc.org/tpch/.

[38] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *CoRR* abs/1901.05152 (2019). arXiv:1901.05152 http://arxiv.org/abs/1901.05152

[39] Stratis D. Viglas. 2013. Just-in-time Compilation for SQL Query Processing. *PVLDB* 6, 11 (2013), 1190–1191.

[40] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *PVLDB* 10, 8 (2017), 889–900.