On Building Robustness into Compilation-Based Main-Memory Database Query Engines

Prashanth Menon

CMU-CS-21-106
May 2021

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Andrew Pavlo (Co-Chair)
Todd C. Mowry (Co-Chair)
Jonathan Aldrich
Thomas Neumann, Technische Universität München (TUM)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by Google, Intel ISTC-CC, and the National Science Foundation under grant numbers CNS-1065112, IIS-1423210, CNS-1423172, IIS-1718582, and CCF-1822933. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Code Generation, Query Compilation, Adaptive Query Processing, Vectorized Processing

To my family, here and yet to arrive.

Abstract

Relational database management systems (DBMS) are the bedrock upon which modern data processing intensive applications are assembled. Critical to ensuring low-latency queries is the efficiency of the DBMSs query processor. Just-in-time (JIT) query compilation is a popular technique to improve analytical query processing performance. However, a compiled query cannot overcome poor choices made by the DBMSs optimizer. A lousy query plan results in lousy query code. Poor query plans often arise and for many reasons. Although there is a large body of work exploring how a query processor can adapt itself at runtime to compensate for inadequate plans, these techniques do not work in DBMSs that rely on compiling queries.

This dissertation presents multiple effective, practical, and complementary techniques to build adaptive query processing into compilation-based engines with negligible overhead. First, we propose a method that intelligently blends two otherwise disparate query processing approaches (compilation and vectorization) into one engine. This necessary first step allows operators to optimize themselves using a combination of software memory prefetching and single instruction, multiple data (SIMD) vectorization resulting in improved performance. Next, we present a framework that builds upon our previous work to allow the DBMS to modify compiled queries without recompiling the plan or generating code speculatively. This technique enables more extensive groups of operators in a query to coordinate their optimization process. Finally, we present a method that decomposes query plans into fragments that can be compiled and executed independently. This not only reduces compilation overhead but enables the DBMS to learn properties about data processed in an earlier phase of the query to hyper-optimize the code it generates for later phases.

Collectively, the techniques proposed in this dissertation enable any compilation-based DBMS to achieve dynamic runtime robustness without succumbing to any of its overheads.

Acknowledgments

I was extremely fortunate to work with not one but two outstanding advisors over the course of this dissertation. Andrew (Andy) Pavlo and Todd Mowry are not only exceptionally brilliant researchers but also incredibly humble, patient, and supportive. Andy is imbued with a frenetic energy that manifests itself through his focused work ethic. Todd possesses an uncanny ability to formalize my rough, vague, nascent ideas and distill their essence to surface its novelty. They both taught me how to identify interesting research problems and how to pursue a research objective doggedly. They were patient with me as I navigated new domains and provided encouragement when I needed it. Andy taught me everything from advanced topics in databases to how to judge the quality of a taco to deciphering prison tattoos. Todd taught me that databases are not the center of the universe (sorry, Andy) and incorporate ideas across disciplines to strengthen one's work. Ultimately, Andy and Todd made me a better researcher, and I will always be grateful for the experience working closely with them over my Ph.D.

I would also thank Jonathan Aldrich and Thomas Neumann for serving on my thesis committee. Jonathan sat through many of my practice talks and provided invaluable feedback. Despite being outside the database community, Jonathan quickly understood the 60-year history of databases, the problems I was trying to address, and my proposed solutions, all while asking incisive questions and offering crucial tweaks to make the work more approachable to a broader audience. Thomas is German. Not only does Thomas possess extensive knowledge across all database topics, but he navigates from high-level to low-level architectural detail with immense ease. His feedback on this work only made it more robust.

I overlapped with many talented researchers and students during my time at CMU: Dana Van Aken, David Andersen, Joy Arulraj, Matt Butrovich, Christos Faloutsos, Phil Gibbons, Anuj Kalia, Michael Kaminsky, Jack Kosaian, Marcel Kost, Viktor Leis, Tianyu Li, Hyeontaek Lim, Wan Shen Lim, Lin Ma, Amadou Ngom, Sivaprasad Sudhir, Jungmin Seo, Yingjun Wu, Jean Yang, Huanchen Zhang. Marcel spent a summer with us in the CMU database group from Karlsruhe Institute of Technology (KIT) and worked with me on the precursor to what would be the critical query execution engine of the NoisePage DBMS, and the foundation for this work. Amadou and Siva aided me in building out the foundation of the PCQ project in Chapter 6.

I am also thankful to everyone that has passed through the CMU database group. Our group meetings are always lively, fun, energetic, sometimes contentious, but always in the pursuit of the best system we can design. Despite the pizza lunches, which never sat well with me, I always enjoyed the design sessions we engaged in and the discussions that ensued. I also want to thank Deb

Cavlovich, Catherin Copetas, Jessica Packer, Karen Lindenfelser, Joan Digney, and all the other excellent administrative staff for simplifying the lives of all graduate students.

I owe a great debt to my family. I am grateful to my mother and father for always being my champion, for their unwavering support in my pursuit of higher education, for always grounding me, and for always lending a non-judgemental and compassionate ear when I need it. I want to thank both my brothers for creating the competitive but loving environment I have grown in. I am also thankful to their respective spouses for adding culture to our discussions outside of The Simpsons alone.

Finally, graduate school was memorable primarily because of my best friend and wife, SM. We met four months after beginning my Ph.D., and I wholly attribute my life outside of research to her. Through all the successes and failures, highs and lows, advancements and setbacks, joys and frustrations that accompany a Ph.D., SM has had my back with nothing but grace, love, and unconditional support. None of this would have been possible were it not for her.

Contents

List of Figures

List of Tables

Chapter 1

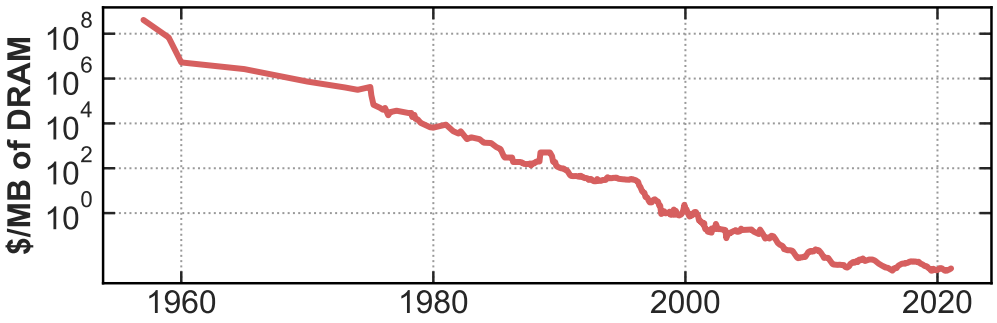
Introduction

Modern data processing applications have markedly evolved over the past decade. What distinguishes this new class of data-intensive applications from their predecessors is (1) the unprecedented scale at which they ingest new information [87] and (2) the complexity of the analysis they perform using a combination of historical and real-time data [27, 119]. The ability to ask complex questions about data immediately after ingestion is useful in many application domains, including Internet advertising (e.g., what advertisement to show someone based on their browsing history?), real-time monitoring systems (e.g., is an incoming packet from a potential attacker?), and financial services (e.g., is this new credit card purchase fraudulent?). Database management systems (DBMS) underpin many of these applications and are the primary workhorse responsible for executing complex queries over large shifting volumes of data with low latency. Optimizing the DBMS's performance has a direct impact on the utility of the applications they support.

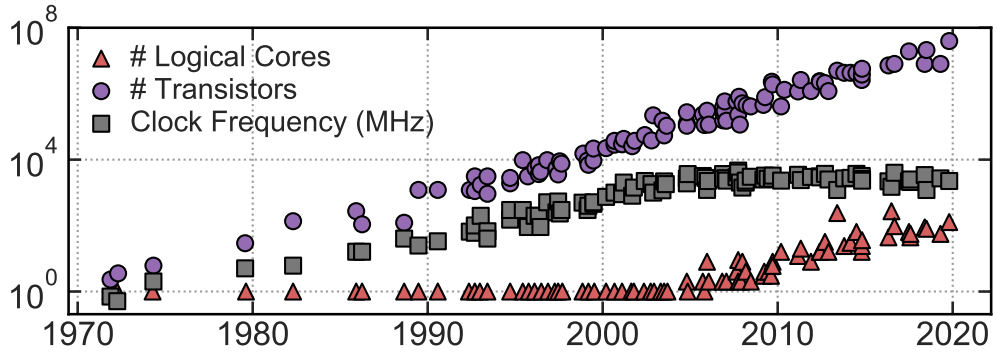
1.1 The Old Guard

The first relational DBMSs emerged in the early 1970s as research prototypes in IBM's System R [22] and University of California's INGRES [142]. These systems were designed for the workload demands and hardware available in their time. For instance, their primary workloads were interactive online transaction processing (OLTP) applications involving a human operator manually issuing commands against the DBMS (e.g., airline reservations [26]). A typical server-class machine in the 1970s had a single CPU core with one hardware thread, a small volatile main memory, and larger (but slow) persistent storage. Lack of enough memory to store an application's entire data set forced the DBMS to rely on a buffer pool to cache encoded disk pages [22]. The disparity of access speeds between main memory and disk and the interactive nature of transactions forced the DBMS to employ a concurrency control mechanism to simultaneously execute multiple transactions to utilize the system fully [117]. A row-oriented layout was the prevalent storage model since it kept all information for a database entity (e.g., a customer) together, aligning with how OLTP applications access their data (e.g., updating a customer's name).

Beginning in the mid-1990s, however, businesses sought to use their databases to ask complicated analytical questions and explore historical, current, and predictive views of their operations [14]. To achieve this, organizations periodically bulk loaded data from their OLTP systems



(a) Dollar Cost of DRAM [97]



(b) CPU Trends [76]

Figure 1.1: Modern Hardware Trends – From (a) we observe that DRAM manufacturing costs have consistently fallen from the early 1960s. In (b) we observe that although Moore's law [106] remains correct even in today's ecosystems [45, 54], clock frequencies have plateaued due to physical limitations in the CPUs ability to dissipate heat [32]. To compensate and continue to offer improved performance, CPU manufacturers rely on increasing core counts.

into a separate database through an extract-transform-load (ETL) process [16]. Business analysts ran their complex online analytics processing (OLAP) workloads on these isolated DBMSs (data warehouses) without interfering with transactional business operations. To accommodate this new class of read-only OLAP applications while minimizing cost, database vendors added extensions (e.g., materialized views, compression, and bitmap indexes) but built them atop the same fundamental DBMS architecture from the 1970s (i.e., row-oriented storage and concurrency control).

The five decades following System R and INGRES witnessed a radical shift in both the hardware landscape and the workloads a DBMS executes. We discuss them separately here.

Changes in Hardware: Modern server systems come equipped with tens of CPU cores, hundreds of gigabytes (or even terabytes) of main memory, and fast solid-state drives (SSD) or persistent memory devices. From Figure 1.1a we see that memory prices have fallen by several orders of magnitude from the 1970s, making it feasible for modern DBMSs to store most (if not all) of an application's working data set in main memory [143]. In Figure 1.1b we observe that while CPU transistor counts are growing following Moore's Law [106], limits on the CPU's ability to dissipate heat have forced CPU manufacturers to trade increased CPU frequency

for increased CPU core counts. Multi-core hardware and main-memory processing present a vastly different operating environment than what the first DBMS designers had available.

Changes in Workload: Modern OLTP applications increasingly interacted with the DBMS using simpler non-interactive interfaces. These new workloads often read and modify only a few records resulting in a small footprint and lifetime, and were write-heavy and repetitive. OLTP workloads also differ from OLAP workloads. OLAP workloads are far less predictable since analysts issue them in an ad-hoc manner as they explore the data set. Since OLAP workloads analyze historical trends, they typically read and process more data than their OLTP counterparts. Finally, analyzing data itself is a read-oriented undertaking, whereas OLTP applications are write-heavy. Traditional DBMSs at the time endeavoured to support both workloads using the same DBMS architecture from the 1970s optimized only for OLTP applications.

These changes prompted researchers in the mid-2000s to revisit the architecture, algorithms, and design principles of existing databases. The consensus they reached at the time was that (1) specialized systems explicitly designed for transaction processing or analytics could handily outperform legacy systems [140, 141], and (2) each specialized system required a "clean-slate" design to achieve optimal performance on modern hardware [67]. What followed is a flurry of research rearchitecting both OLTP and OLAP DBMSs from the ground up. Well-known OLTP systems include H-Store [75] (commercialized as VoltDB), Microsoft Hekaton [48], Oracle TimesTen [86], Calvin [147], Silo [149], and Shore-MT [73]. Similarly, the analytics space was dominated by column stores pioneered by C-Store [139] (commercialized as Vertica) and MonetDB/X100 [29] (commercialized as Vectorwise), but include SybaseIQ [94], MonetDB [31], Quickstep [116], Amazon Redshift (formerly ParAccel) [65], Snowflake [42], and IBM DB2 BLU [125].

Although the DBMS market's bifurcation into specialized OLTP and OLAP systems has yielded success, the ETL process persists. That is, there is a delay between when the OLTP DBMS initially ingests new data and when it is available for analysis in the OLAP DBMS. Since the mid-2010s, the explosion in data volumes and increasing demand for faster and deeper insights have given rise to a growing set of business applications that cannot afford to pay this ETL lag and require real-time data analysis [119]. In response, researchers have explored the design of hybrid transaction and analytical processing (HTAP) database systems [27, 57, 111]. An HTAP (also referred to as operational analytics) DBMS supports both transactional and analytical workloads. Well-known HTAP systems include SAP HANA [52, 58, 89], Oracle In-Memory [85], HYRISE [63], HyPer [77], BatchDB [95], and SingleStore [135].

It is in an HTAP environment that we explore the topics in this dissertation. Specifically, we seek to improve the *performance* and *robustness* of OLAP queries in an HTAP DBMS. We discuss these topics individually in the following sections.

1.2 Improving OLAP Performance

The OLAP performance of a DBMS is dictated by its efficiency in executing SQL query plans. Historically, most DBMSs process queries by implementing a form of the Volcano iterator model [60]. In this approach, the DBMS structures query plans as a tree of relational operators wherein each operator pulls and processes a single row (tuple) at a time from their children in the tree. Such an

interpretation-based approach provides a clean, composable, and testable abstraction that is easy for humans to reason about. However, previous work has shown that the CPU inefficiencies of the iterator model are the bottleneck for in-memory systems due to excessive use of virtual function calls, pointer chasing, and branch misprediction [29]. 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) [29, 55].

One popular technique to reduce the overhead of interpretation is vectorization, pioneered in MonetDB/X100 [29]. Like the Volcano-style iteration described previously, vectorization also pulls data from leaves to the root through an iterator interface. However, each invocation of the iterator fetches a block (i.e., vector) of 1k–2k tuples rather than just one tuple. This simple enhancement (1) amortizes the iteration overhead across all tuples in the vector and (2) maximizes computation on tuple data while in the CPU's cache. Relational operators rely on a set of core highly-optimized *primitives* that implement simple operations (e.g., addition or hash computation) on one or more type-specific vectors of data. Although vectorization has been shown to improve CPI during query processing, there are still overheads related to evaluating complex expressions. Vectorization alone is ill-suited in an HTAP DBMS that also executes OLTP transactions since these workloads process a small number of tuples. Fewer tuples result in smaller vectors that cannot hide the overhead of the Volcano processing model [78].

One approach to reducing the DBMS's instruction count during query execution is a method from the 1970s [34] that is now seeing a revival: just-in-time (JIT) *query compilation* [84, 126, 151]. With this technique, the DBMS compiles queries (i.e., SQL) into native machine code that is specific (i.e., "hard-coded") to that query. Compared with the interpretation-based query processing approach 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 passes between operators directly in CPU registers [108]. Previous work has shown that query compilation is a promising technique to optimizing both OLTP and OLAP workloads [55, 78, 108, 135]. However, the static nature of compiled plans which confers the largest performance benefits is also one of its largest weaknesses. We now describe why it is critical for a compilation-based DBMS to be flexible during query processing.

1.3 Robust Query Processing

A critical component in any DBMS is the query optimizer. A DBMS query optimizer's job is to find a "good" physical execution plan to use for a query. The more sophisticated optimizers use a combination of statistics (e.g., cardinality estimates) and a cost model to compare different, but semantically equivalent plans. In theory, if the estimates and cost model are accurate, and given infinite time to explore the set of equivalent plans, the DBMS optimizer will optimally choose the best query plan. However, previous work has shown that DBMS statistics are frequently wrong because they rely on simplifying assumptions that are invalid on real-world databases, such as uniformity and independence [91]. Incorrect statistics lead to bad decisions about join orders resulting in sub-optimal and sometimes catastrophic query plans. There is substantial evidence showing that

optimizer errors accrue exponentially in the size of a query [72]. As the demand for real-time analytics increases, it is incumbent on the DBMS to mitigate these errors.

One approach to compensate for poor choices by the optimizer is *adaptive query processing* (AQP) [46]. 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 used by most OLAP DBMSs, with AQP, the DBMS interleaves the optimization and execution stages such that they provide feedback to each other [23]. 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 [96]. Alternatively, the DBMS could change yet-to-be-executed portions of the current plan (i.e., pipelines) [160]. In the case of the latter, the system must ensure that the re-optimized query does not generate duplicate results nor miss existing results from the previous execution.

AQP is available in several commercial DBMSs, including Oracle and Microsoft SQL Server. 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. 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 [40, 61] or for sub-plans within a pipeline [24]. The optimizer injects special operators (e.g., change [61], switch [24]) into the plan that contain conditionals to determine which sub-plan to use at runtime. For example, such a conditional could examine the output cardinality of an operator below it in the plan tree, and then decide whether to execute a join using a hash or nested-loop algorithm.

Although there is a rich body of research on how the DBMS adapts [24], none are suitable in compilation-based DBMSs. Compilation and AQP have fundamentally conflicting goals: compilation strives to specialize the query as much as possible by removing generic logic to allow the compiler to generate the most efficient code. In contrast, AQP relies on generic logic with multiple data paths for a query to achieve runtime flexibility.

1.4 Thesis Statement

This thesis seeks to address the challenge of building robustness into compilation-based query engines. We use *robustness* to describe the DBMSs ability to change parts (or all) of a query plan mid-execution to best suit runtime conditions. We contend that neither compilation nor vectorization is wholly optimal, but rather an intelligent orchestration of both techniques is required for a DBMS to adapt to (1) concurrent workloads on the system and (2) skewed data distributions that affect query performance.

This thesis aims to provide evidence to support the following statement:

Thesis Statement: *Combining the design principles from compilation-based and vectorized query processing engines enables an analytical database management system to optimize its execution environment, adapt to the changing data distributions, and improve overall query performance with minimal overhead.*

1.5 Summary of Goals & Contributions

While query compilation is necessary to achieve good performance on contemporary hardware, it is not sufficient given today's workloads and those we foresee in the future. This dissertation's overall goal is to improve the state-of-the-art in robust query execution in compilation-based DBMSs by addressing the issues enumerated above. Our work integrates ideas and techniques spanning across the database and compiler research communities. Although our efforts were developed and evaluated in two research HTAP DBMSs, we hope that the ideas laid out in this document serve as a guide for readers as they build adaptive compilation-based query engines.

The contributions in this dissertation are as follows:

A Hybrid Query Engine (Chapter 3) We present a query processing model that blends query compilation and vectorized processing in one engine. The DBMS can selectively fuse subsets of operators to efficiently pipeline data and vectorize other operators to exploit inter-tuple parallelism [101].

Adaptive Compiled Queries (Chapter 4) We present a method to generate and compile code in a manner that allows the DBMS to modify compiled plans with negligible overhead [102]. The DBMS then uses this framework to implement several optimizations.

Incremental Code Generation (Chapter 5) We present a method that decomposes query plans into fragments and interleaves their code generation and execution. The DBMS relies on this decoupling to learn properties about data a query reads in early parts to tailor code it generates for later parts.

1.6 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents background exposition on query execution techniques in modern main-memory DBMSs. Chapter 3 presents an initial design of a hybrid query engine that combines query compilation and vectorization, laying the foundation for the remainder of our work. Chapter 4 presents an adaptive query processing technique tailored for compilation-based DBMSs that can dynamically customize execution to data distributions. Chapter 5 introduces a new approach to code generation that enables the DBMS to safely customize generated code to the data a query reads without incurring compilation overhead. Chapter 6 presents a discussion of the related work and Chapter 7 discusses possible areas for future work. We conclude the dissertation in Chapter 8.

Chapter 2

Background

This chapter provides an overview of the two prevailing techniques for query execution in modern in-memory DBMSs: query compilation and vectorized query processing. To aid in our discussion, we use the Q19 query from the TPC-H benchmark [146]. A simplified version of this query's SQL is shown in Figure 2.1. We omit the query's `WHERE` clauses for simplicity; it is not important for our exposition.

2.1 Query Processing

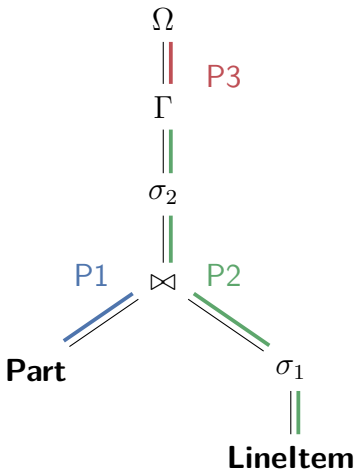
A query goes through several preparation stages before execution by the query processor. The DBMS first parses the SQL text into an abstract syntax tree, performs semantic analysis, and translates it into a tree of relational algebra operators. The DBMS optimizer then optimizes this tree using a cost model to generate a physical execution plan it believes is "best" to execute given an existing workload and available system resources. It is the query processor's responsibility to execute this physical plan using existing data structures within the DBMS (e.g., row- or column-oriented tables, B-Tree indexes) or creating new structures (e.g., hash indexes).

One of the most popular query execution models is the Volcano iterator model [60]. In this model, all operators implement a generic iterator interface composed of three functions: (1) `open` which initializes the operator, (2) `getNext` which requests the operator to produce the next valid output tuple in its stream and return it to the caller, and (3) `close` which instructs the operator to deinitialize and end its stream. Being a tree structure, each non-leaf operator has one or more child iterators depending on the semantics of the operation. Such a design provides a clean, composable, and testable abstraction that is easy for humans to understand. It enables the DBMS to compose arbitrary combinations of operators without knowing (or caring) how each operator behaves internally (i.e., the interface hides implementation details).

Query processing proceeds by repeatedly invoking the `getNext` function on the root operator to produce tuples satisfying the query. On each invocation, the root operator calls `getNext` on its child operators. These invocations cascade down the query plan tree recursively until reaching a leaf operator, typically a table or index scan operation. This access pattern is referred to as a pull-based operator model since operators *pull* tuples from their children and deliver them to their parents (i.e., from the leaves to the root). Operators are implemented generically since they must process

```
SELECT SUM(...) AS revenue
 FROM LineItem
 JOIN Part ON l_partkey = p_partkey
WHERE (CLAUSE1) OR (CLAUSE2) OR (CLAUSE3)
```

(a) An abbreviated version of TPC-H's Q19



(b) Query Plan with Pipelines

Figure 2.1: TPC-H Q19 – Figure 2.1a is an abbreviated version of TPC-H's Q19. The three clauses are a sequence of conjunctive predicates on attributes in both the `LINEITEM` and `PART` tables. Figure 2.1b shows the generated physical query plan for TPC-H's Q19 with annotated pipelines.

any data type and operation supported by the DBMS. A DBMS's expression evaluation system best exemplifies this general-purpose behavior. A simple expression adding the values in two columns, $a + b$, is typically implemented using an expression tree with the addition operator as the root. The addition needs to support all combinations of input data types a and b. For instance, the ANSI SQL-92 [100, 157] standard allows adding integers and variable precision numerics together. To enable this, operators use dynamic type information to *interpret* the bytes within a tuple, cast them to the appropriate type at runtime, and ensures proper error handling (e.g., overflow logic). Generically implementing operators obviates the need to write and compile operators for all possible combinations of data types the DBMS supports.

To help illustrate this approach, consider physical plan for TPC-H Q19 shown in Figure 2.1b. It contains scan of tables `PART` and `LINEITEM`, a filter both directly over `LINEITEM` and after the hash join of the base tables, concluding in a hash aggregation. Let us focus on σ_2. Processing is initiated by invoking `getNext` on the aggregation which is forwarded to σ_2. The `getNext` function in σ_2 sits in a loop iterating over the results of its child operator (i.e., the subtree rooted at the hash join), retrieving one tuple at a time, and applying the filter. When σ_2 receives a tuple that satisfies the predicate term, it immediately returns to the caller. The aggregation receives this tuple and either updates an existing aggregate in its hash table, or creates a new one. When done, it requests another valid tuple from its child σ_2. This process repeats until all operators have exhausted their inputs or an exception occurs.

2.2 Query Compilation

Although the iterator model has many benefits, it was designed in an era where I/O was the primary bottleneck in query processing. Previous work has shown that Volcano-style iteration introduces CPU inefficiencies when deployed in a main-memory DBMS [29]. These inefficiencies

```
1  // Join Hash-Table
2  HashTable ht;
3  // Scan Part table, P
4  for (auto &part : P.GetTuples()) {
5    ht.Insert(part);
6  }
7  // Running Aggregate
8  Aggregator agg;
9  // Scan LineItem table, L
10 for (auto &lineitem : L.GetTuples()) {
11   if (PassesPredicate1(lineitem)) {
12     auto &part = ht.Probe(lineitem);
13     if (PassesPredicate2(lineitem, part)) {
14       agg.Aggregate(lineitem, part);
15 } } }
16 // Return the final aggregate.
17 return agg.GetRevenue();
```

P1

P2

P3

Figure 2.2: Query Compilation Example – The generated code for TPC-H Q19 (Figure 2.1) using a tuple-at-a-time processing model.

arise for many reasons. First, the DBMS query driver must invoke the `getNext` function to produce each valid output tuple. These calls cascade through the entire tree to produce one tuple. Although manageable for OLTP queries that process few tuples, these functions may be called billions of times in OLAP queries. Moreover, the `getNext` function is usually virtual, making it more expensive and less predictable than a regular function invocation. Second, since each operator is implemented using generic logic, it typically relies on type-based dispatch to jump to code routines to interpret the bytes within a tuple. Indirection and poor instruction locality degrade the CPUs branch predictor leading to poor CPU performance.

An alternative query processing approach is query compilation. With query compilation, the system converts a query plan tree into a series of code routines that are "hard-coded" just for that query. This greatly reduces the number of conditionals and other checks that the DBMS performs during query execution.

There are multiple ways to compile queries in a DBMS. One method is to generate C/C++ code that is then compiled to native code using an external compiler (e.g., `gcc`). Another method is to generate an intermediate representation (IR) that is compiled into machine code by a runtime engine (e.g., `LLVM`). Lastly, staging-based approaches exist wherein a query engine is partially evaluated to automatically generate specialized C/C++ code that is compiled using an external compiler [80]. Each of these approaches has different software engineering and compilation trade-offs.

In addition to the variations in how the DBMS compiles a query into native code, there are several techniques for organizing this code [108, 114, 136, 151]. One naïve way is to create a separate routine per operator. Operators then invoke the routines of their children operators to pull up the next data (e.g., tuples) to process. Although this is faster than interpretation, the CPU can still incur expensive branch mispredictions because the code jumps between routines [108].

A better approach (used in HyPer [108] and SingleStore [115]) is to employ a push-based model that reduces the number of function calls and streamlines execution. To do this, the DBMS's optimizer first identifies the *pipeline breakers* in the query's plan. A pipeline breaker is any operator that explicitly spills any or all tuples out of CPU registers into memory. In a push-based model, child operators produce and *push* tuples to their parents, requesting them to consume the tuples. A tuple's attributes are loaded into CPU registers and flow along the pipeline's operators from the start of one breaker to the start of the next breaker, at which point it must be materialized. Any operator between two pipeline breakers operate on tuples whose contents are in CPU registers, thus improving data locality.

We now illustrate this approach using the same TPC-H Q19 plan shown earlier in Figure 2.1. Figure 2.1b shows the plan annotated with its three pipelines **P1**, **P2**, and **P3**. In this work, we use Ω to denote consumption of the plan's output. The DBMS's optimizer generates one loop for every pipeline. Each loop iteration begins by reading a tuple from either a base table or an intermediate data structure generated from a child operator (e.g., a hash-table used for a join). The routine then processes the tuple through all operators in the pipeline, storing a (potentially modified) version of the tuple into the next materialized state for use in the next pipeline. Combining the execution of multiple operators within a single loop iteration is known as *operator fusion*. Fusing together pipeline operators in this manner obviates the need to materialize tuple data to memory, allowing the engine to instead pass tuple attributes through CPU registers, or the cache-resident stack.

Returning to our example, Figure 2.2 shows the colored blocks of code that correspond to the identically-colored pipelines in the query execution plan in Figure 2.1b. The first block of code (**P1**) performs a scan on the `PART` table and the build-phase of the join. The second block (**P2**) scans the `LINEITEM` table, performs the join with `PART`, and computes the aggregate function. The code fragments demonstrate that pipelined operations execute entirely using CPU registers and only access memory to retrieve new tuples or to materialize results at the pipeline breakers. Further, it shows that the DBMS can achieve good code locality since the generated code is compact and has tight loops.

2.3 Vectorized Processing

The code in Figure 2.2 achieves better performance than interpreted query plans because it executes fewer CPU instructions, including costly branches and function calls. However, it processes data in a tuple-at-a-time manner, which makes it difficult for the DBMS to employ optimizations that operate on multiple tuples at a time [15].

Generating multiple tuples per iteration has been explored in previous systems. MonetDB's bulk processing model materializes the entire output of each operator to reduce the number of function calls in the system. The drawback of this approach is that this is bad for cache locality unless the system stores tables in a columnar format and each query's predicates are selective [162].

Instead of materializing the entire output for each operator, the X100 project [29] for MonetDB that formed the basis of VectorWise (now Actian Vector) generates a vector of results (typically 100–10k tuples). This approach is employed by both research DBMSs ([41, 116, 123]) and commercial DBMSs ([1, 37, 42, 85, 125]). Modern CPUs are inherently well-suited to vectorized processing. Since loops are tight and iterations are often independent, out-of-order CPUs can execute multiple it-

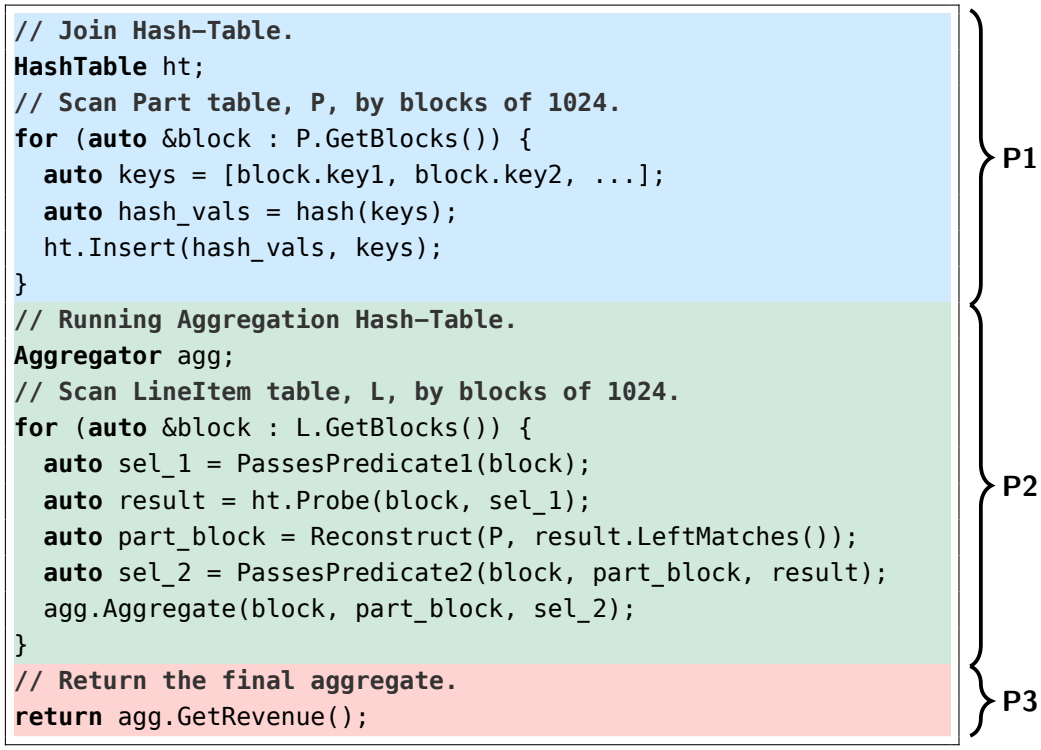


Figure 2.3: Vectorization Example – An execution plan for TPC-H Q19 from Figure 2.1 that uses the vectorized processing model.

erations concurrently, fully leveraging its deep pipelines. A vectorized engine relies on the compiler to automatically detect loops that can be converted to SIMD, but modern compilers are only able to optimize simple loops involving computation over numeric columns. Only recently has there been work demonstrating how to manually optimize more complex operations with SIMD [121, 122].

Figure 2.3 shows pseudo-code for TPC-H Q19 using the vectorized processing model. First, **P1** is almost identical to its counterpart in Figure 2.2, except tuples are read from `PART` in blocks. Moreover, since vectorized DBMSs employ late materialization, only the join-key attribute and the corresponding tuple ID is stored in the hash-table. **P2** reads blocks of tuples from `LINEITEM` and passes them through the predicate. In contrast to tuple-at-a-time-processing, the `PassesPredicate1` function is applied to all tuples in the block in one iteration. If the predicate is conjunctive, this loop is run *for each component* of the conjunction. All of the predicate filter functions produce an array of the positions of the tuples in the block that pass the predicate (i.e., *selection vector*). This vector is woven through each call to retain only the valid tuples. The system then probes the hash-table with the input block and the selection vector to find join match candidates. It uses this result to reconstruct the block of tuples from the `PART` table. The penultimate operation (filter) uses both blocks and the selection vector from the join to generate the final list of valid tuples.

Vectorized processing leverages both modern compilers and modern CPUs to achieve good performance. It also enables the use of explicit SIMD vectorization; however, most DBMSs do not employ SIMD vectorization throughout an entire query plan. Many systems instead only use SIMD in limited ways, such as for internal sub-tasks (e.g., checksums). Others have shown how to use

SIMD for all of the operators in a relational DBMS but they make a major assumption that the data set is small enough to fit in the CPU caches [121], which is usually not possible for real applications. But this means that if the data set does not fit in the CPU caches, then the processor will stall because of memory accesses and then the benefit of vectorization will be minimal.

Chapter 3

Relaxed Operator Fusion

As described earlier, the decreasing cost of DRAM has enabled modern high-performance DBMSs to store the bulk of their working data set in main memory. In this setting, the primary bottleneck in query execution is CPU efficiency particularly in the form of cache misses to memory and computational throughput. Modern CPUs support instructions that help on both of these fronts: (1) software *prefetch* instructions can move blocks of data from memory into the CPU caches before they are needed, thereby hiding the latency of expensive cache misses [38, 81]; and (2) *SIMD* instructions can exploit vector-style data parallelism to boost computational throughput [121].

A key challenge for both software prefetching and SIMD vectorization is that neither technique works well in a tuple-at-a-time processing model used by existing compilation-based engines. In order to successfully hide cache miss latency with prefetching, the software must prefetch a number of tuples ahead (to overlap the cache miss with the processing of other tuples). To bundle together SIMD-width vectors for SIMD processing, the software needs to extract data parallelism across chunks of tuples at a time. While both software prefetching and SIMD vectorization require the ability to look across multiple tuples at a time, there are key differences and subtleties in how they interact with each other. For example, SIMD vector instructions require that data be packed together contiguously, whereas prefetching needs to generate a set of addresses (to be prefetched) ahead of time, but it does not require either those addresses (or the data blocks that they point to) to be arranged contiguously. In addition, since the relative sparsity of the data that is being processed tends to increase in the higher levels of a query plan tree, this also changes relative trade-offs between software prefetching and SIMD vectorization.

Although query compilation, vectorization, software prefetching, and SIMD have been studied in the past (in isolation) for specific DBMS operators, to the best of our knowledge, no DBMS has successfully combined all techniques into a single query processing engine. Part of the reason is that most systems that employ query compilation generate tuple-at-a-time code that avoids tuple materialization. However, both vectorization and prefetching requires a vector of input tuples to successfully exploit data-level parallelism. Recent work has explored implementing SIMD scans in a query compiling DBMS, but it requires reverting to interpreted scans that feed its results into compiled code tuple-at-a-time [88] or is unable to stage data suitably to use prefetching [41]. DBMSs based on vectorized processing rely on out-of-order CPUs to exploit data-level parallelism, but this often is not possible for complex operators, such as hash-joins or index probes. Hence, we contend

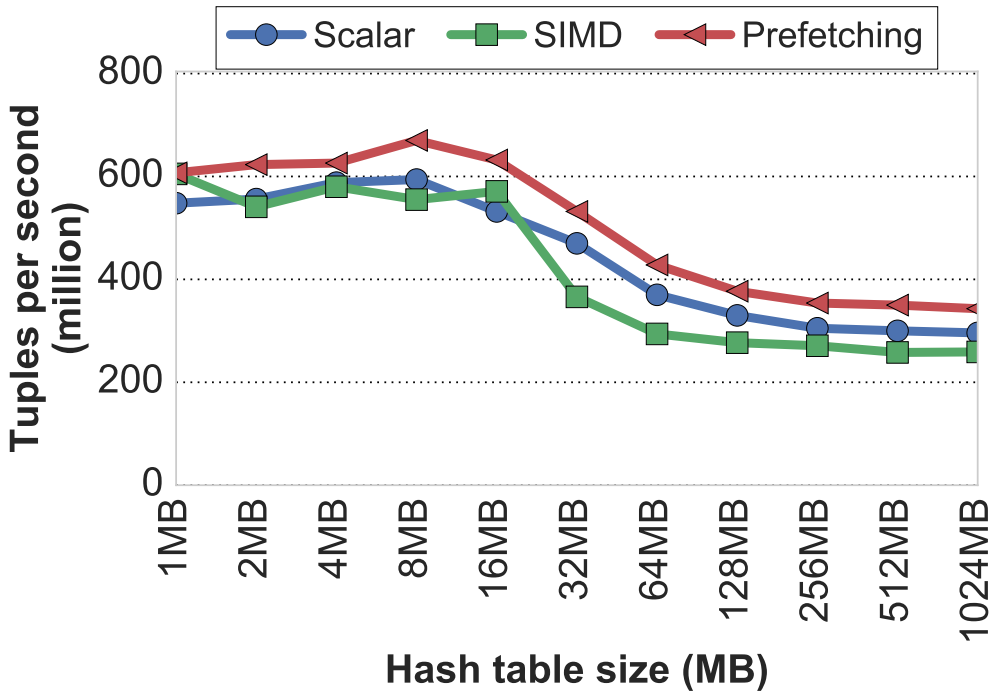


Figure 3.1: Microbenchmark – Effect of hash-table size on join performance

that what is needed is a hybrid model that is able to tactfully materialize state to support both tuple-at-a-time processing, vectorized processing, and prefetching.

3.1 Motivating Example

To help demonstrate this point, we execute a microbenchmark that performs a hash-join between two tables, each containing a single 32-bit integer column (as in [121]). We implemented three different approaches: (1) a scalar tuple-at-a-time join, (2) a SIMD join using the vertical vectorization technique described in [121], and (3) a tuple-at-a-time join modified to use group-prefetching [38].

We measure the overall throughput in output tuples per-second as we vary the size of the join's hash-table from 1 MB to 1024 MB. We keep the size of the probe table (A) fixed to 100m tuples, totaling ~382 MB of raw data. We vary the size of the build table (B) such that it produces a hash-table of the desired size in the experiment. The hash-table uses open-addressing with linear probing, a 50% fill-factor, and the 32-bit finalizer from MurmurHash3 [20] as the hash function. We choose this to simulate real-world implementations and because it requires minimal computation (three bit-shifts, two multiplications, and three XORs). Each hash-table bucket stores a 4-byte key and a 4-byte payload, and the hash-table is organized as an array-of-structs. The values in both tables are uniformly distributed, and each tuple in A matches with at most one tuple in B. The join produces a combined table with 100m tuples with a total size of ~381 MB. We executed our test on 20 hardware threads (hyper-threading is disabled) with 25 MB of shared L3 CPU cache. We defer the full description of our environment until Section 5.4.

From the results shown in Figure 3.1, we see that SIMD probes utilizing the vertical vectorization technique performs worse than tuple-at-a-time probes with prefetching, even when the hash-table is cache-resident. This is because vectorization requires recomputing hash values on hash and key collisions. The second observation is that both tuple-at-a-time techniques with and without prefetching outperform the SIMD version when the hash-table does not fit in cache. This is because the join shifts from a compute-bound operation to a memory-bound operation for which SIMD does not help. Lastly, we see that tuple-at-a-time processing with prefetching is consistently better than all approaches, often by up to 1.2×.

The main takeaway from the microbenchmark results is that tuple-at-a-time processing with prefetching outperforms SIMD for hash-joins regardless of hash-table sizes. Prefetching requires looking across a vector of tuples to exploit inter-tuple parallelism, but fully pipelined compiled plans avoid any materialization. Moreover, data becomes more sparse and memory accesses become more random as we move up the query plan tree, making prefetching that much more important. At the leaves of the tree, the DBMS can rely on the hardware prefetcher; this is not true higher up.

Neither wholly vectorized nor wholly compiled query plans are optimal. What is needed is the ability for the DBMS to tactfully materialize tuples through prefetching at various points in the query plan — to enable vectorization and exploit inter-tuple parallelism — and otherwise fuse operators to ensure efficient pipelining. To this end, we now present our hybrid engine architecture, **Relaxed Operator Fusion** (ROF), that blends query compilation and vectorization into a single engine.

3.2 Overview

The primary goal of operator fusion is to minimize materialization. We contend that strategic materialization can be advantageous as it can exploit inter-tuple parallelism inherent in query execution. Tuple-at-a-time processing by its nature exposes no inter-tuple parallelism. Thus, to facilitate strategic materialization, one could *relax* the requirement that operators within a pipeline be fused together. With this, the DBMS instead decomposes pipelines into *stages*. A stage is a partition of a pipeline in which all operators are fused together. Stages within a pipeline communicate solely through cache-resident vectors of tuple IDs. Tuples are processed sequentially through operators in any given stage one-at-a-time. If the tuple is valid in the stage, its ID is appended into the stage's output vector. Processing remains within a stage while the stage's output vector is not full. If and when this vector reaches capacity, processing shifts to the next stage in the pipeline, where the output vector of the previous stage serves as the input to the current. Since there is always exactly one active processing stage in ROF, we ensure both input and output vectors (when sufficiently small) will remain in the CPU's caches.

ROF is a hybrid between pipelined tuple-at-a-time processing and vectorized processing. There are two key distinguishing characteristics between ROF and traditional vectorized processing; with the exception of the last stage iteration, ROF stages always deliver a full vector of input to the next stage in the pipeline, unlike vectorized processing that may deliver input vectors restricted by a selection vector. Secondly, ROF enables vectorization across *multiple* sequential relational operators (i.e., a stage), whereas conventional vectorization operates on a single relational operator, and often times *within* relational operators (e.g., vectorized hash computation followed by a vectorized hash-table lookup).

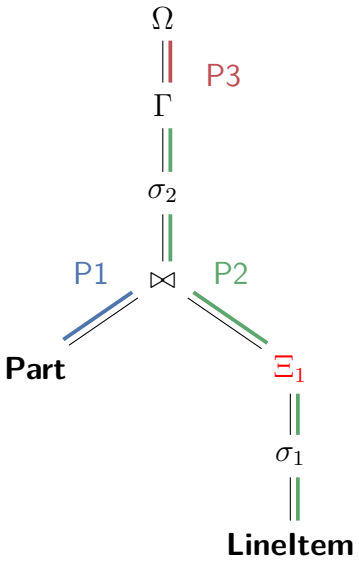


Figure 3.2: Example Query Plan Using ROF – The physical query plan for TPC-H Q19 using our relaxed operator fusion model.

To help illustrate ROF's staging, we first walk through an example. We then describe how to implement ROF in an in-memory DBMS.

3.2.1 Example

Returning again to our running TPC-H Q19 example, Figure 3.2 shows a modified query plan using our ROF technique to introduce a single stage boundary after the first predicate (σ_1). The Ξ operator denotes an output vector that represents the boundary between stages. Thus, pipeline **P1** in Figure 3.2 is separated into two stages.

The code generated for this modified query plan is shown in Figure 3.3. In the first stage (lines 13–20), tuples are read from the `LINEITEM` table and passed through the filter to determine their validity in the query. If a tuple passes through the filter (σ_1), then its ID is appended to the stage's output vector (Ξ). When this vector reaches capacity, or when the scan operator has exhausted tuples in `LINEITEM`, the vector is delivered to the next stage.

The next stage in the pipeline (lines 22–30) uses this vector to read valid `LINEITEM` tuples for probing the hash-table and finding matches. If a match exists, both components are passed through the secondary predicate (σ_2) to again check the validity of the tuple in the query. If it passes this predicate, it is aggregated as part of the final aggregation operator.

We first note that one loop is still generated per-pipeline (lines 11–31). A pipeline loop contains the logic for all stages contained in the pipeline. To facilitate this, the DBMS splits pipeline loops into multiple inner-loops, one for each stage in the pipeline. In this example, lines 13–20 and 22–30 are for the first and second stages, respectively. The DBMS fuses together the code for the operators within a stage loop. This is seen in Figure 3.3 as line 26 corresponds to the probe, line 27 to the second predicate, and line 28 to the final aggregation. In general, there are the same number of inner-loops per pipeline loop as there are stages, and the number of stage output vectors (Ξ) is equal to one less than the number of stages.

```
#define VECTOR_SIZE 256

HashTable join_table;  // Join Operator Table
Aggregator aggregator; // Running Aggregator

oid_t buf[VECTOR_SIZE] = {0}; // Stage Vector
int buf_idx = 0; // Stage Vector Offset
oid_t tid = 0; // Tuple ID

// Pipeline P2
while (tid < L.GetNumTuples()) {
  // Stage #1: Scan LineItem, L
  for (buf_idx = 0; tid < L.GetNumTuples(); tid++) {
    auto &lineitem = L.GetTuple(tid);
    if (PassesPredicate1(lineitem)) {
      buf[buf_idx++] = tid;
      if (buf_idx == VECTOR_SIZE) break;
    }
  }
  // Stage #2: Probe, filter, aggregate
  for (int read_idx = 0; read_idx < buf_idx; read_idx++) {
    auto &lineitem = L.GetTuple(buf[read_idx]));
    auto &part = join_table.Probe(lineitem);
    if (PassesPredicate2(lineitem, part)) {
      aggregator.Aggregate(lineitem, part);
} } }
```

Figure 3.3: ROF Staged Pipeline Code Routine – The example of the pseudo-code generated for pipeline P2 in the query plan shown in Figure 3.2. In the first stage, the code fills the stage buffer with tuples from table LINEITEM that satisfy the scan predicate. Then in the second stage, the routine probes the join table, filters the results of the join, and aggregates the filtered results.

Lastly, the code maintains both a read and write position for each output vector. The write position tracks the number of tuples in the vector; the read position tracks how far into the vector a given stage has read. A stage has exhausted its input when either (1) the read position has surpassed the amount of data in the materialized state (i.e., data table or hash-table) or (2) the read and write index are equal for the input vector. Therefore, a pipeline is complete only when all of its constituent stages are finished. If a stage accesses external data structures that are needed in subsequent stages, ROF requires a companion output vector that stores data positions/pointers that it is aligned with the primary TID output vector.

Our ROF technique is flexible enough to model both tuple-at-a-time processing and vectorized processing, and hence, subsumes both models. The former can be realized by creating exactly one stage per pipeline. Since a stage fuses all operators it contains and every pipeline has only one stage,

pipeline loops contain no intermediate output vectors. Vectorized processing can be modeled by installing a stage boundary between pairs of operators in a pipeline.

Staging alone does not provide many benefits; however, it facilities two optimizations not possible otherwise: SIMD vectorization, and prefetching of non-cache-resident data.

3.2.2 Vectorization

SIMD processing is generally impossible when executing a query tuple-at-a-time. Our ROF technique enables operators to use SIMD by introducing a stage boundary on their input, thereby feeding them a vector of tuples. The question now becomes whether to also impose a stage boundary on the output of a SIMD operator. With a boundary, SIMD operators can efficiently issue selective stores of valid tuple IDs into their output vectors. With no boundary, the results of the operator are pipelined through all operators that follow in the stage. This can be done using one of two methods. Both methods assume the result of a SIMD operator resides in a SIMD register. In the first method, the operator breaks out of SIMD code to iterate over the results in the individual SIMD lanes one-at-a-time [33, 159]. Each result is pipelined to the next operator in the stage. In the second method, rather than iterate over individual lanes, the operator delivers its results in a SIMD register to the next operator in the stage. Both methods are not ideal. Breaking out of SIMD code unnecessarily ties up the registers for the duration of the stage. Delivering the entire register risks under-utilization if not all input tuples pass the operator, resulting in unnecessary computation.

Given this, we greedily force a stage boundary on all SIMD operator outputs. The advantages of this are (1) SIMD operators always deliver a 100% full vector of only valid tuples, (2) it frees subsequent operators from performing validity checks, and (3) the DBMS can generate tight loops that are exclusively SIMD. We now describe how to implement a SIMD scan using these boundaries.

Implementation: In contrast to scalar selection where the result of applying a predicate against a tuple is a single boolean value indicating the validity of the tuple, the result of a SIMD application of a predicate is a bit-mask. Processing n elements in parallel produces a bit-mask stored in a SIMD register where all bits of each of the n elements are either 0 or 1 (to indicate the validity of the associated tuple). To determine which tuples are valid using the bit-mask, the DBMS could employ partial vectorization and iterate over the bits in the mask to extract one bit at a time. But this requires breaking out of the SIMD code and has $O(n)$ complexity.

ROF uses a different approach that is wholly in SIMD code. The technique leverages a precomputed, cache-resident index to lookup permutation masks that are used to shuffle SIMD elements into valid and invalid components. To illustrate how a DBMS uses these bit-masks to efficiently determine valid tuples, we walk through an example. The illustration in Figure 3.4 is performing a SIMD scan over a 4-byte integer column `colattr_A` and evaluating the predicate `colattr_A` < 44. The DBMS first loads as many attribute values as possible (along with their tuple IDs) into the widest available SIMD register. Next, it applies the predicate to produce a bit-mask. In the example, the tuples with IDs (1, 3, 7) fail to pass the predicate. To correctly write out only the valid IDs, the DBMS shuffles the tuple ID SIMD register so that the valid and invalid tuple IDs are stored contiguously, effectively partitioning the register. To achieve this, the DBMS invokes the `movemask` instruction to convert the bit-mask into an integer number that it uses as an index into a *permutation table*. This is a precomputed table that maps a given bit-mask value to a 8-byte bit-mask that corresponds to the

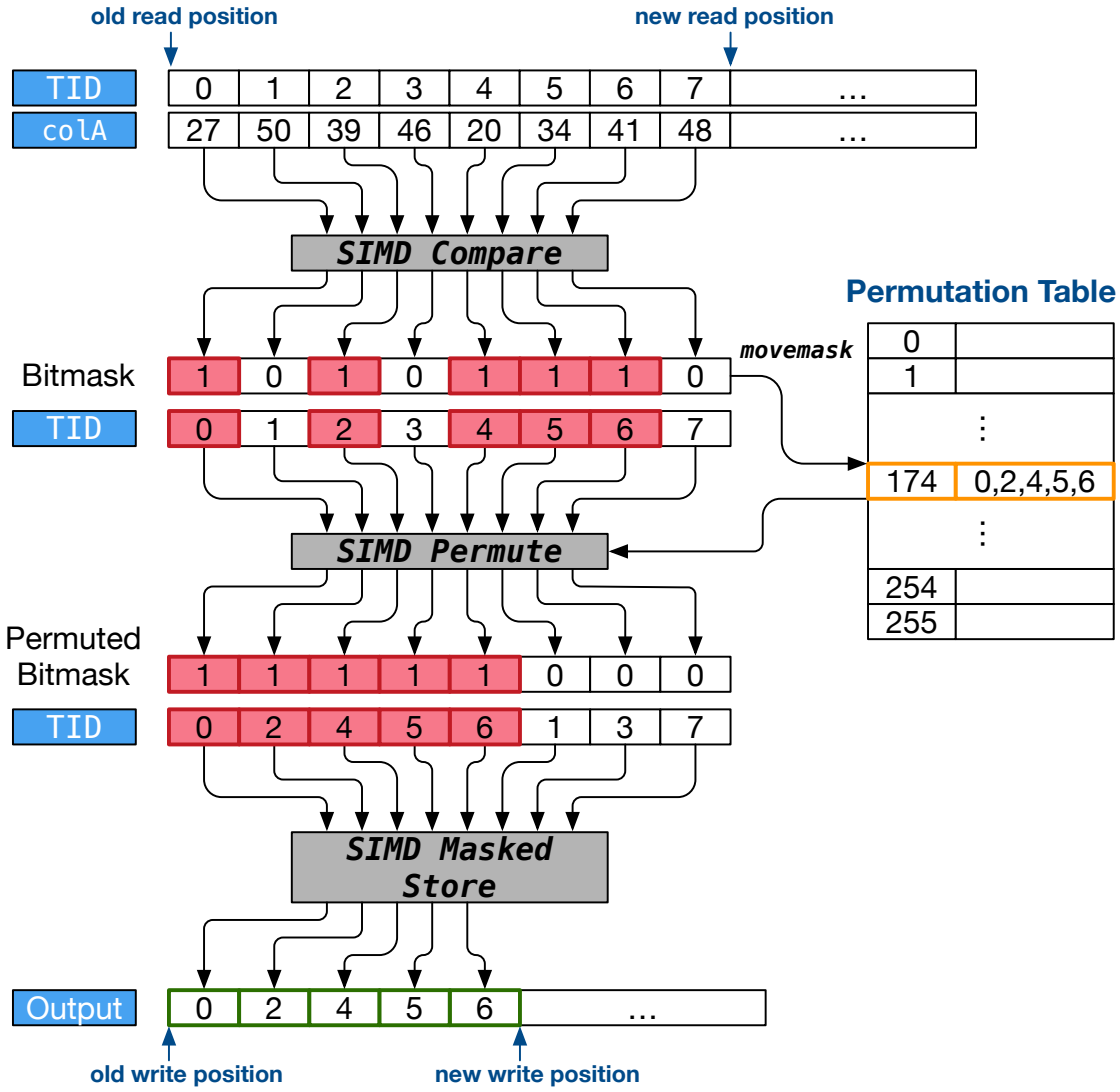


Figure 3.4: SIMD Predicate Evaluation Example – An illustration of how to use SIMD to evaluate the predicate for TPC-H Q19.

correct re-arrangement of elements in the SIMD register to partition it into valid and invalid parts. In the example, the bit-mask's value is 174, which corresponds to the bit-mask (0,2,4,5,6). Applying this permutation bit-mask moves elements in positions 0, 2, 4, 5, and 6 to the first five elements in the register. We apply this permutation to both the original bit-mask and the tuple ID counter. The DBMS then writes the modified tuple ID counter to the output vector at the current write position using a masked store with the modified bit-mask as the selection mask. The DBMS then increments the new write position by the number of valid tuples (using the `popcnt` instruction), loads a new vector of values, and increments the tuple ID vector by eight.

The permutation table stores an 8-byte value (i.e., the permutation bit-mask) for each possible input bit-mask. A SIMD register storing n elements can produce 2^n masks. With AVX2 256-bit registers operating on eight 4-byte integers, this results in $2^8 = 256$ possible bit-masks. Thus, the

size of the largest permutation table is at most $2^8 \times 8 = 2$ KB, small enough to fit in the CPU's L1 cache. For data types that are smaller than 4-bytes, we cast upwards to 4-byte values.

3.2.3 Prefetching

Aside from the regular patterns of sequential scans, the more complex memory accesses within a DBMS are beyond the scope of what today's hardware or commercial compilers can handle. Therefore, we propose new compiler passes for automatically inserting prefetch instructions that can handle the important irregular, data-dependent memory accesses of an OLAP DBMS.

The DBMS must prefetch sufficiently far in advance to hide the memory latency (i.e., overlapping it with useful computation), while at the same time avoiding the overhead of unnecessary prefetches [107]. Limiting the scope of prefetching to within the processing of a single tuple in a pipeline is inefficient because by the time the DBMS knows how far up the query plan the tuple will go, there is not sufficient computation to hide the memory latency. On the other hand, aggressively prefetching all of the data needed within a pipeline can also hurt performance due to cache pollution and wasted instructions. These challenges are exacerbated as the complexity within a pipeline increases, since it becomes increasingly difficult to predict data addresses in advance.

Our ROF model avoids all of these problems. The DBMS installs a *stage boundary* at the input to any operator that requires random access to data structures that are larger than the cache. This ensures that prefetch-enabled operators receive a full vector of input tuples, enabling it to overlap computation and memory accesses since these tuples can be processed independently. To estimate the size of any constructed or used data structures, the DBMS planner uses statistics and the structure of the query plan. Hash joins and hash-based aggregations are two classes of important operators that benefit significantly from prefetching. Index-based joins are another operator that can potentially benefit from prefetching. If the planner discovers that the join index is larger than the size of the CPU cache, it installs a stage boundary at the input to facilitate prefetching. If the join index is based on a traditional B+Tree, the engine generates group-prefetching (GP) [38] code since each input tuple will, on average, perform an equal number of random memory accesses (i.e., one for each level of the tree). If the index has a more irregular access pattern (e.g., BwTree or a skip-list), then the engine should generate asynchronous memory access chaining (AMAC) [81] code to support early termination of probes.

Implementation: To help ground our discussion of how to implement prefetching in ROF, we briefly discuss the design of the hash-table used in both hash-joins and hash-based aggregations, found in Figure 3.5. We use an open-addressing hash-table design with linear probing for hash and key-collisions. Previous work has shown this design to be robust and cache-friendly [129]. We use MurmurHash3 [20] as our primary hash function. This is differs from previous work [121] that prefers to use computationally simpler (and therefore faster) hash functions, such as `multiply-add-shift`. We want to use a general-purpose hash function that can (1) work on multiple different non-integer data types, (2) provide a diverse hash distribution, and (3) execute fast. MurmurHash3 satisfies these requirements and is used in many popular systems [2, 3, 8].

Our hash-table, shown in Figure 3.5, is laid out as a contiguous array of buckets. Buckets begin with an 8-byte `status` field that indicates (1) if this bucket is empty, (2) if it is occupied by a single key-value pair, or (3) if it is occupied and there are duplicate values for the key. Duplicate values

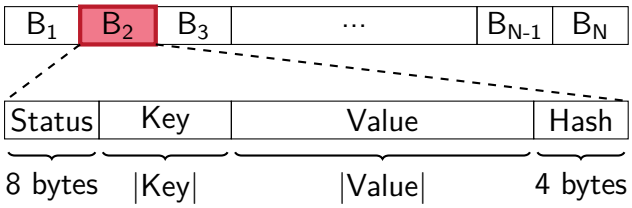


Figure 3.5: Hash-Table Data Structure – An overview of the DBMS's open-addressing hash-table used for joins (with B_N buckets).

are stored externally in a contiguous memory space, and the `status` field is re-purposed to store a pointer to this memory location. The key and value data are stored next in the bucket, followed by the hash value. We store the `status` and key value near the beginning of the bucket to ensure we can read both with one memory-load; this is obviously not possible if the key exceeds the size of a cache-line (minus 8 bytes). This is important since the `status` field is read on *every* hash-table access for both insertions and probes, whereas the key is needed to resolve key-collisions. The hash value is used only during table resizing to avoid recomputation. Since resizing is far more infrequent than insertions and probes, storing the hash value at the end does not impact overall join or aggregation performance.

The hash-table's design is amenable to both software and hardware prefetching. Since joins and aggregations operate on tuple vectors, software prefeteching will speed up the initial hash-table probe. Secondly, the hardware prefetcher kicks in to accelerate the linear probing search sequence for hash collisions. By front-loading the `status` field and the key, the DBMS tries to ensure that at most one memory reference to a bucket is necessary to check both if the bucket is occupied and if the keys match.

Although we can employ any software prefetching technique with ROF, we decide to use GP for multiple reasons. Foremost is that GP requires a simpler code structure and is faster to generate and compile than AMAC [38, 81]. GP also naturally provides synchronization boundaries between code stages for a group to resolve potential data races when inserting duplicate key-value pairs. Finally, using an open-addressing hash-table with linear probing means that all tuples have exactly one random access into the hash-table during probes and insertions (with the exception of duplicate-handling which requires two). Since all tuples in a group have the same number of random accesses even in the presence of skew, AMAC does not improve performance over GP.

3.2.4 Query Planning

A DBMS's optimizer has to make two decisions per query when generating code with the ROF model: (1) whether to enable SIMD predicate evaluation and (2) whether to enable prefetching.

During optimization, Peloton's planner takes a greedy approach and installs a boundary after every scan operator if the scan has a SIMD-able predicate. Determining whether a given predicate can be implemented using SIMD instructions is a straightforward process that uses data-type and operation information already encoded in the expression tree. As we will show in Section 5.4, using SIMD when evaluating predicates during a scan never degrades performance.

The planner can also employ prefetching optimizations using two methods. In the first method, the query planner relies on database- and query-level statistics to estimate the sizes of all interme-

diate materialized data structures required by the query. For operators that require random access to data structures whose size exceeds the cache size, the planner will install a stage boundary at the operator's input to facilitate prefetching. This heuristic can backfire if the collected statistics are inaccurate (see Section 3.3.3) and result in a minor performance degradation. An alternative approach is for the query planner to *always* install a stage boundary at the input to any operator that performs random memory accesses, but generate two code paths: one path that does prefetching and one that does not. The query compiler generates statistics collection code to track the size of intermediate data structures, and then uses this information to guide the program through either code path at runtime. In this way, the decision to prefetch is independent of query planning. We note that this approach will result in a code explosion as each branch requires a duplication of the remaining query logic; this process can repeat for each prefetching operator. ROF remedies this by installing a stage boundary at the operator's output, thereby duplicating only the operator's logic rather than the entire query plan.

3.3 Experimental Evaluation

We now present a brief analysis of our ROF query processing model. For this evaluation, we implemented ROF in the Peloton in-memory DBMS [11]. Peloton is an HTAP DBMS that used interpretation-based execution engine for queries. We modified the system's query planner to support JIT compilation using LLVM (v3.7). We then extended the planner to also generate compiled query plans using our proposed ROF optimizations.

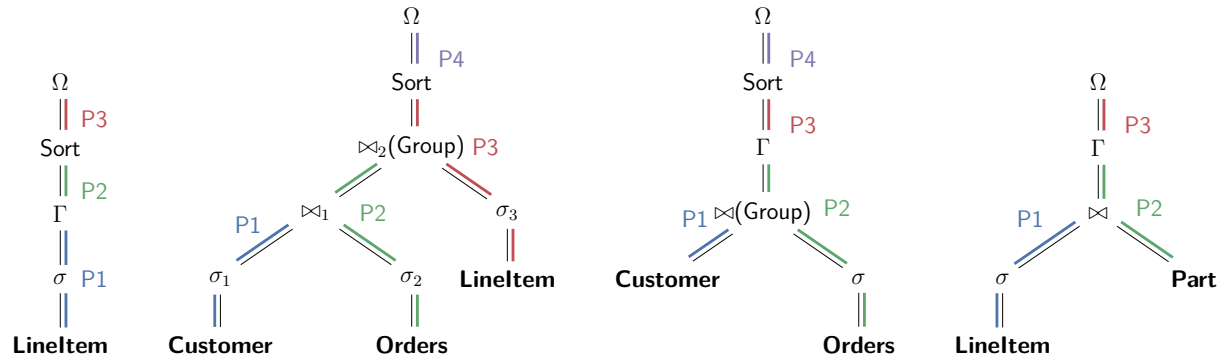
We performed our evaluation on a machine with a dual-socket 10-core Intel Xeon E5-2630v4 CPU with 25 MB of L3 cache and 128 GB of DRAM. This is a Broadwell-class CPU that supports AVX2 256-bit SIMD registers and ten outstanding memory prefetch requests (i.e., ten line-fill buffer (LFB) slots) per physical core. We also tested our ROF approach with an older Haswell CPU and did not notice any changes in performance trends.

In this section, we first describe the workload that we use in our evaluation. We then present a high-level comparison of the performance of our ROF query processing model versus a baseline implementation that only uses query compilation. Next, we provide a detailed breakdown of the query plans to explain where the optimizations of the ROF model have their greatest impact. We then select the optimal query plan for each TPC-H query and measure the sensitivity of the results to two important compiler parameters for ROF (i.e., vector size and prefetching distance). To prevent cache coherence traffic from interfering with our measurements, we limit the DBMS to only use a single thread per query and do not execute multiple queries at the same time for these initial experiments. We then evaluate the DBMS's performance with ROF when using multiple threads and finish with a comparison of the absolute performance of Peloton with ROF against two state-of-the-art OLAP DBMSs.

We ensure that the DBMS loads the entire database into the same NUMA region using `numactl`. We run each experiment ten times and report the average measured execution time over all trials.

3.3.1 Workload

We use a subset the TPC-H benchmark in this evaluation [146]. TPC-H is a decision support system workload that simulates an OLAP environment where there is little to prior knowledge of



(a) Q1 (b) Q3 (c) Q13 (d) Q14

Figure 3.6: TPC-H Query Plans with Pipelines – The high-level query plan for the subset of the TPC-H queries that we evaluate in our experiments, and do deep-dive analysis on. Each plan is annotated with their pipelines [108].

the queries. It contains eight tables in 3NF schema. We use a scale factor of 10 in each experiment (∼10 GB). Peloton is still early in development; we plan to run larger scale experiments are it matures.

Although the TPC-H workload contains 22 queries, we select eight queries that cover all TPC-H choke-point query classifications [28] that vary from compute- to memory/join-intensive queries. Thus, we expect our results to generalize and extend to the remaining TPC-H queries. An illustration of the pipelined plans for these queries is shown in Figure 3.6; the plan for Q19 is shown in Figure 2.1b.

3.3.2 Baseline Comparison

For this first experiment, we execute the TPC-H queries using the data-centric approach used in HyPer [108] and other DBMSs that support query compilation. We deem this as the baseline approach. We then execute the queries with our ROF model. This demonstrates the improvements that are achievable with the prefetching and vectorization optimizations that we describe in Section 5.2.

Figure 3.7 shows the performance of our execution engine (which implements a data-centric query compilation engine [108]) both with and without our ROF technique enabled. As we see in the figure, our ROF technique yields performance gains ranging from 1.7× to 2.5× for seven of eight queries.

3.3.3 Optimization Breakdown

To better understand how ROF impacts performance, we now present case studies for a subset of the TPC-H queries we evaluated in Figure 3.7. For the sake of space, we only discuss in detail five of the eight TPC-H queries we evaluate. The results we draw extend to the remaining queries. Figures 3.8 to 3.12 show the execution time for each query broken down by the time spent in each pipeline in the original query plans from Figure 3.6 as we incrementally apply ROF to additional operators within the tree. The leftmost bar is the baseline execution (without ROF), the next bar

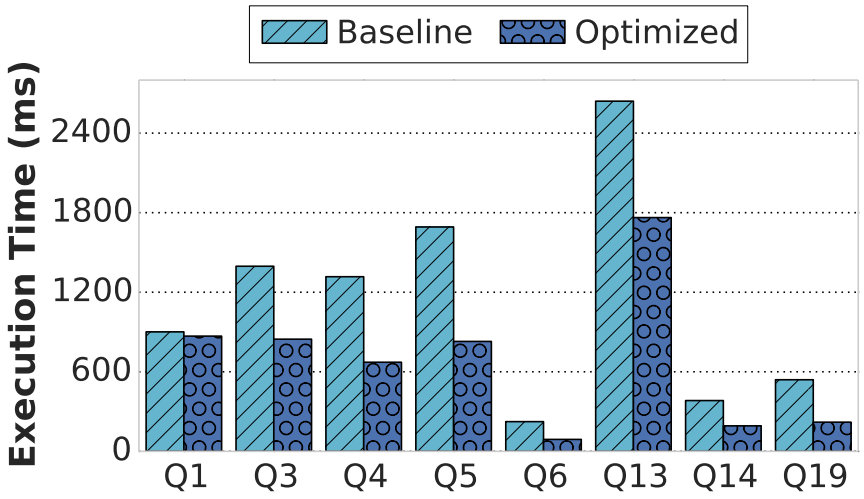


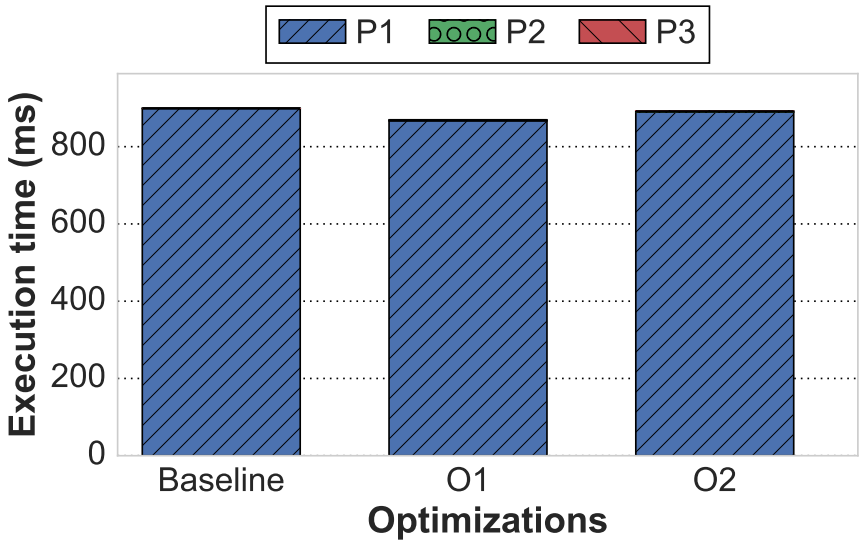
Figure 3.7: Baseline Comparison – Query execution time when using regular query compilation (baseline) and when using ROF (optimized).

applies ROF to one operator, the bar after that includes both plus applying ROF to any additional operators. These optimizations are cumulative and each one is orthogonal to the previous. We describe the details of these optimizations in a table below each graph.

Q1 Case Study: ROF yields only a marginal improvement (1.04×) over the baseline for this query. The bulk of Q1's execution time is spent in **P1**, which performs a selection and an aggregation. **P2** and **P3** are not easily visible in Figure 3.8: the time spent materializing aggregated data into a memory heap (in preparation for sorting) and the sorting itself accounts for less than 0.3% of the execution time.

For our first optimization (**O1**), the planner converts the predicate on the `LINEITEM` table into a SIMD scan. It adds a stage boundary after σ in **P1**, and converts the scalar predicate evaluation into a SIMD check. But because this predicate matches almost the entire table (i.e., 98% selectivity). At such high-selectivity, the benefit data-level parallelism (i.e., reduced CPI) from using SIMD predicate evaluation is offset by the overhead of (redundant) data copying of valid IDs into an output vector. Thus, in this case SIMD yields only a marginal reduction in latency. Moreover, the scan portion of **P1** accounts for only 4% of the overall time; the remaining 96% of the time is in the aggregation (Γ). This means that applying SIMD to the scan (assuming a maximum theoretical speedup of 8× using AVX2 256-bit registers) can only achieve a speedup of at most 1.036×.

The second optimization (**O2**) uses the output of the SIMD scan stored in σ's output stage vector (Ξ) to prefetch hash-table entries in the build phase of the aggregation (Γ). But Figure 3.8 shows prefetching makes the query slower. Γ's hash-table is sufficiently small enough (just four entries) to fit in the CPU's L1 cache, obviating the need for prefetching. The overhead of the DBMS invoking the prefetch instructions is non-negligible and thus degrades performance. **O2** highlights the importance of accurate query statistics. Inaccurate statistics may lead the planner to incorrectly install an ROF stage boundary to enable prefetching resulting in reduced performance.



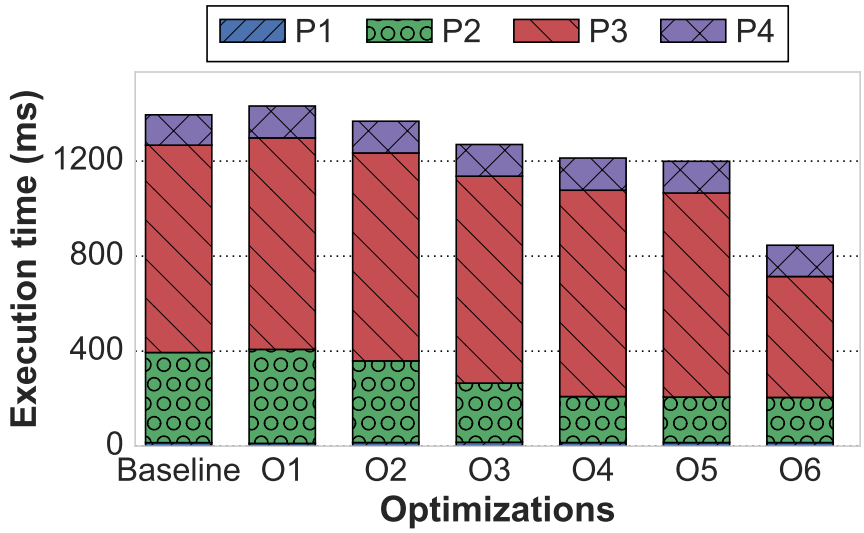
O1	**Modification:** **P1** $\Rightarrow$ (LINEITEM $\rightarrow \sigma \rightarrow \Xi \rightarrow \Gamma$) **Description:** Apply SIMD to predicate σ.
O2	**Modification:** — **Description:** Use Ξ to prefetch buckets on Γ build.

Figure 3.8: Q1 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q1 query plan shown in Figure 3.6a.

Q3 Case Study: This next query is the most complex one from the TPC-H workload that we evaluate. As such, it presents the largest number of opportunities to apply our ROF technique. The first observation from the measurements in Figure 3.9 is that the optimizations that apply SIMD predicate evaluation (i.e., **O1**, **O2**, and **O5**) all offer marginal performance improvements. This is because the majority of time spent in pipelines **P1**, **P2**, and **P3** are not scanning table data, but rather in the joins $\bowtie_1$ and $\bowtie_2$. Roughly 1% of time in **P1** is spent filtering CUSTOMER tuples, 3.7% of **P2** is on scanning and filtering the ORDERS table, and 7% of **P3** is on scanning the LINEITEM table. Although using SIMD with these operators provides some benefit, it does not address the main bottlenecks.

Instead, the more complex, memory-intensive operators will produce greater improvements. These are the optimizations that implement prefetching of hash-table buckets for either the build- or probe-phase of joins $\bowtie_1$ or $\bowtie_2$ (i.e., **O3**, **O4**, and **O6**). **O3** uses the staging vector written to by the SIMD predicate on σ_2 to prefetch hash-table buckets for the probe of $\bowtie_1$. This speeds up **P2** by 1.4×. **O4** improves upon this by introducing a second stage boundary after the join $\bowtie_1$. This second stage prefetches hash-table buckets during the build phase of the join $\bowtie_2$. **O4**'s prefetching improves **P2**'s execution time by 1.26× and for the overall query by 1.14×.

The last and most important optimization for Q3 is **O6** because this highlights the advantage of the ROF approach. While using SIMD for the predicate evaluation on LINEITEM (which is the largest table in the query) only increases performance by 1.02×, using the resulting output vector to perform prefetching on the probe of $\bowtie_2$ results in an improvement of 1.38×. In general, we find



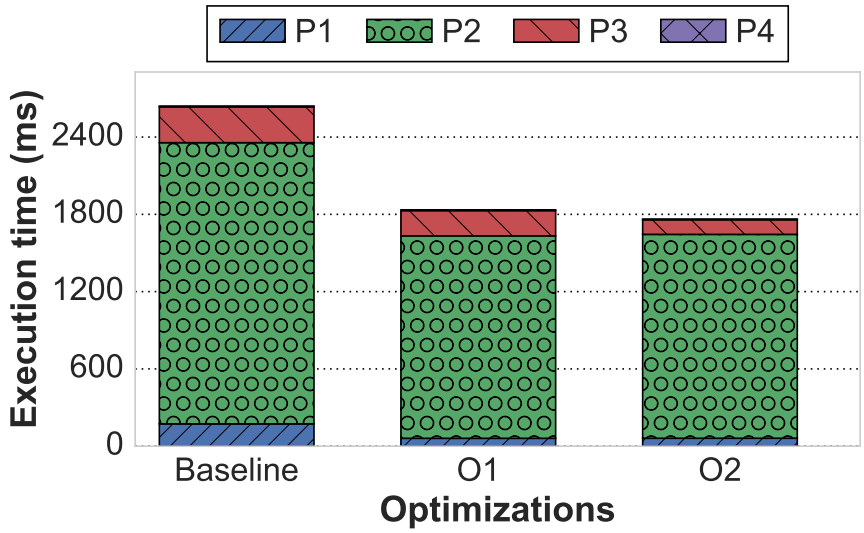
O1	**Modification: P1** $\Rightarrow$ (CUSTOMER $\rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow \bowtie_1$) **Description:** Apply SIMD to predicate σ_1 (CUTOMER).
O2	**Modification: P2** $\Rightarrow$ (ORDERS $\rightarrow \sigma_2 \rightarrow \Xi_2 \rightarrow \bowtie_1 \rightarrow \bowtie_2$) **Description:** Apply SIMD to predicate σ_2 (ORDERS).
O3	**Modification:** — **Description:** Use Ξ_2 to prefetch buckets during $\bowtie_1$ probe.
O4	**Modification: P2** $\Rightarrow$ (ORDERS $\rightarrow \sigma_2 \rightarrow \Xi_2 \rightarrow \bowtie_1 \rightarrow \Xi_3 \rightarrow \bowtie_2$) **Description:** Use Ξ_3 to prefetch buckets during build of $\bowtie_2$.
O5	**Modification: P3** $\Rightarrow$ (LINEITEM $\rightarrow \sigma_3 \rightarrow \Xi_4 \rightarrow \bowtie_2 \rightarrow Sort$) **Description:** Apply SIMD to predicate σ_3.
O6	**Modification:** — **Description:** Use Ξ_4 to prefetch buckets for $\bowtie_2$ probe.

Figure 3.9: Q3 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q3 query plan shown in Figure 3.6b.

that using ROF optimizations across this query plan improves by up to 1.61×, with much of this resulting from prefetching.

Q13 Case Study: This query presents another interesting data point because it demonstrates that ROF can still improve performance when only using prefetching without SIMD. Q13 contains a predicate on ORDERS that cannot be implemented using SIMD since it contains a non-trivial LIKE clause on a string column. However, by installing a stage boundary after the predicate (σ), Figure 3.10 shows that ROF still improves performance over the baseline.

The first optimization (**O1**) prefetches hash-table buckets for both the build- and probe-phases of the group-join ($\bowtie$). We implement the group-join operator from [105]. As described previously,



O1	**Modification: P2** $\Rightarrow$ (ORDERS $\rightarrow \sigma \rightarrow \Xi_1 \rightarrow \bowtie \rightarrow \Gamma$) **Description:** Use Ξ_1 to prefetch buckets for build and probe of $\bowtie$.
O2	**Modification: P2** $\Rightarrow$ (ORDERS $\rightarrow \sigma \rightarrow \Xi_1 \rightarrow \bowtie \rightarrow \Xi_2 \rightarrow \Gamma$) **Description:** Use Ξ_2 to prefetch buckets during build of Γ.

Figure 3.10: Q13 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q13 query plan shown in Figure 3.6c.

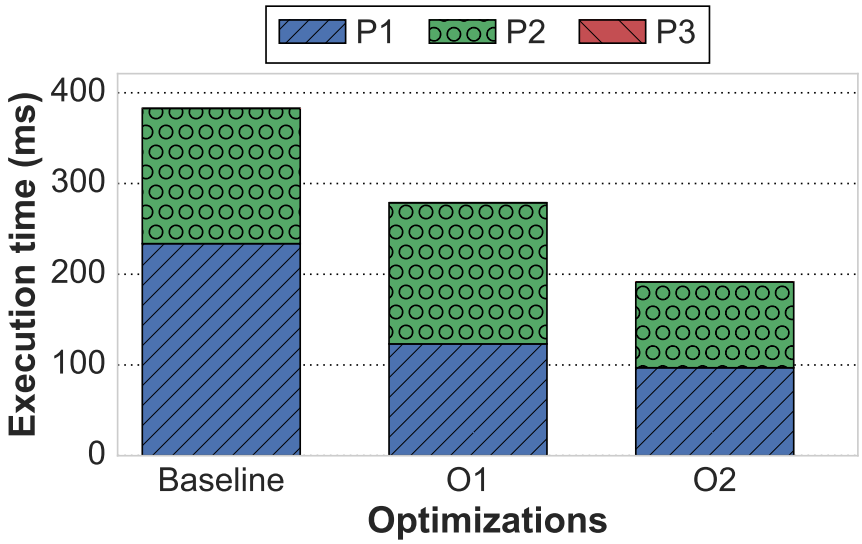
the DBMS installs a stage boundary (Ξ_1) after the predicate σ. The scan of ORDERS is entirely scalar, and its output is written to the stage's output vector. The DBMS is then able to prefetch on the build input since it is already being read from materialized state (i.e., the CUSTOMER table). **O1** results in a performance improvement of 1.34× over the baseline.

The second optimization (**O2**) prefetches hash-table buckets needed during the build phase of the aggregation (Γ). Once again the DBMS installs a stage boundary (Ξ_2) after the group-join ($\bowtie$). This optimization further improves performance over **O1** by 1.04×, resulting in an overall improvement of 1.5× compared to the baseline.

Q14 Case Study: The predicate on the LINEITEM table has only a 2% selectivity and, hence, it is an ideal candidate for being converted into a SIMD predicate. To do so, in **O1** the planner installs a stage boundary (Ξ) after the predicate over LINEITEM (σ). This improves **P1**'s execution time by 1.89× and Q14's overall time by 1.37×.

Next, the planner enables prefetching for both the build- and probe-phases of the join in **O2**. For **P1**, this is facilitated by re-using **O1**'s output vector (Ξ) for the SIMD predicate (σ). **P2** does not need an additional stage since the scan of the PART table is directly from the table. **O2** further improves performance by 1.45× from the previous optimization and by almost 2× over the baseline.

We do not apply any optimizations on the aggregation operator (Γ) because it is static (i.e., it always generates a single output tuple). In generated code, the aggregation is implemented as a simple counter. Hence, **P3** requires no computation and contributes effectively nothing to the query's



O1	**Modification: P1** $\Rightarrow$ (LINEITEM $\rightarrow \sigma \rightarrow \Xi \rightarrow \bowtie$) **Description:** Apply SIMD to predicate σ (LINEITEM).
O2	**Modification:** — **Description:** Use Ξ and PART to prefetch buckets during join $\bowtie$.

Figure 3.11: Q14 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q14 query plan shown in Figure 3.6d.

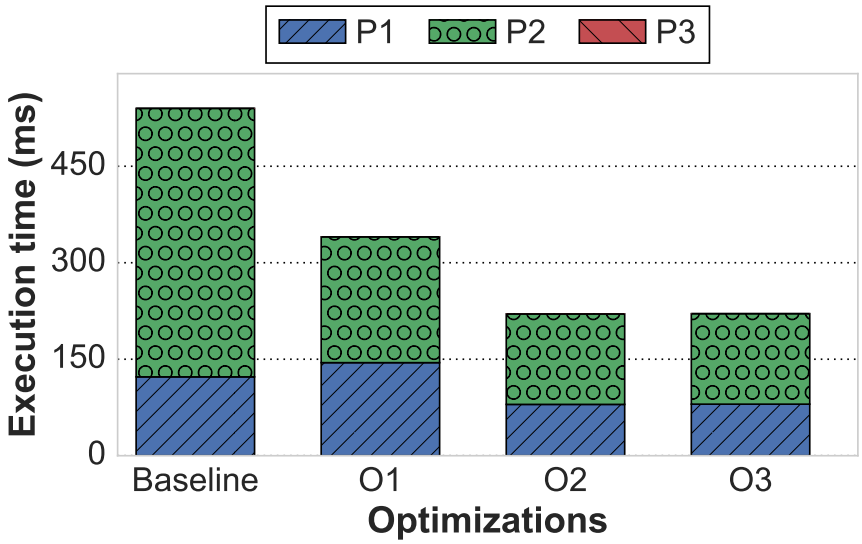
overall execution time of the query. We note that Peloton does not currently support generating index nested-loop join code and, instead, employs a slightly less performant hash-join between `Part` and `LineItem`.

Q19 Case Study: Like Q14, the predicate on the `LINEITEM` table in this query is selective; less than 4% of tuples make it through the filter. This predicate is extracted from a much larger disjunctive predicate that is applied to the results of the join ($\bowtie$). Again, this means that the predicate is a good candidate for SIMD evaluation. Thus, **O1** installs a stage boundary and output vector (Ξ_1), after the predicate on `LINEITEM` (σ_1). The results in Figure 3.12 show that **O1** improves the overall performance of the query by almost 1.6×.

The second optimization uses **O1**'s output vector to issue prefetch instructions for hash-table buckets during the hash-join probe. Similarly, **O2** also modifies **P1** to prefetch hash-table buckets to build $\bowtie$. Together these two optimizations further improve Q19's performance by almost 1.6× and almost 2.5× over the baseline implementation. We note that the contribution of **P3** is effectively zero because it is a static aggregation that performs no computation.

3.3.4 Sensitivity to Vector Width

In the previous experiments, we executed the queries using the optimal vector sizes and prefetch group size. We now analyze the sensitivity of the ROF model to these two configuration parameters. Our evaluation shows that these parameters are independent to each other and thus we will examine



O1	**Modification: P2** $\Rightarrow$ (LINEITEM $\rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow\bowtie\rightarrow \sigma_2 \rightarrow \Gamma$) **Description:** Apply SIMD to predicate σ_1 (LINEITEM).
O2	**Modification:** — **Description:** Use Ξ_1 to prefetch buckets during probe of $\bowtie$.
O3	**Modification: P2** $\Rightarrow$ (LINEITEM $\rightarrow \sigma_1 \rightarrow \Xi_1 \rightarrow\bowtie\rightarrow \Xi_2 \rightarrow \sigma_2 \rightarrow \Xi_3 \rightarrow \Gamma$) **Description:** Insert staging points between every pair of operators.

Figure 3.12: Q19 Case Study – The breakdown of the ROF optimizations applied to TPC-H Q19 query plan shown in Figure 2.1b.

them separately. We begin with measuring the effect of the stage output vector size to overall query performance. We select the optimal staged plan for the eight TPC-H queries and fix the prefetch group size to 16. We then vary the size of all the output vectors in each plan from 64 to 256k tuples.

The results in Figure 3.13 show that all but one of the queries (Q13) are insensitive to the size of the stages' output vectors. This is notable for Q14 and Q19 because we showed in the previous experiment that they both benefited greatly from SIMD vectorization. This is because their scans are highly selective (2% for Q14 on the ORDERS table and 4% for Q19 on the LINEITEM table), and so using SIMD shifts the main bottleneck to the next stage in their respective pipelines. For Q14, this is mainly the probe-side of the join, whereas it is the build-side of the join in Q19. The extra latency incurred due to accessing larger-than-cache hash-tables constitutes the largest time component of their execution plan, even though it is ameliorated through prefetching.

Q1 is also insensitive to the vector size, but for a slightly different reason. In this query, more than 98% of the tuples in the LINEITEM table qualify the predicate. As such, the primary bottleneck in the first pipeline is not in the SIMD scan, but in the aggregation. This aggregation does not benefit from larger vector sizes, which is why Q1 is not impacted by varying this configuration parameter.

For Q13, the primary bottleneck in the query plan is the evaluation of the LIKE clause on the colo_comment field of the ORDERS table. Since the DBMS cannot execute this predicate using SIMD, increasing the size of the predicate's output vector does not help. It instead uses this vector to prefetch

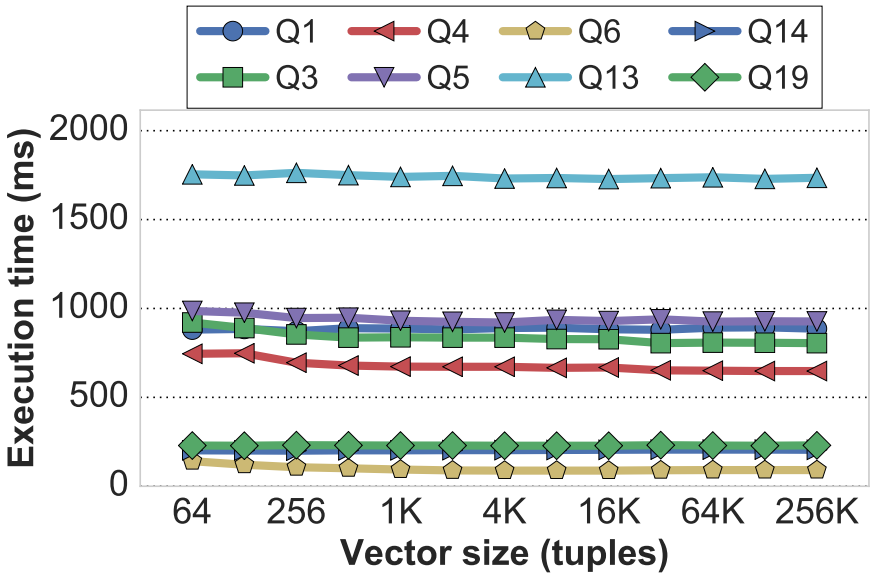


Figure 3.13: Sensitivity to Vector Width – The average execution time of the TPC-H queries when varying the maximum number of tuples stored in the ROF's stage output vectors.

hash-table buckets as part of the subsequent probe. But since the query plan uses a group size of 16, the system is already able to saturate the memory parallelism in the hardware and thus larger vector sizes do not help.

The only query whose performance gets better as the vector width increases is Q3. The results show that this query benefits slightly with larger vector sizes, but does not improve further when vector size exceeds 16k tuples. This is because the SIMD scan on the `LINEITEM` table remains in the SIMD code stage over a larger range of tuples. The predicate is roughly 54% selective, and so using larger vectors reduces the number of outer-most loop iterations. In general, larger vectors reduces the total number of outer-most loop iterations. This is helpful for scan queries with low-selectivity predicates. However, larger vectors don't improve performance for queries with joins. This is because modern CPUs support a limited number of outstanding memory references, making the query become memory-bound quicker than CPU-bound.

3.3.5 Sensitivity to Prefetching Distance

We next analyze the performance of the DBMS when varying the size of the ROF model's prefetch groups. We again use the best staged plans that we generated in Section 3.3.2 for each query. This time we fix the output vector size constant to use the optimal configuration for each query as determined in Section 3.3.4. We then vary the group sizes from zero (disabling prefetching) to 256 tuples.

The results in Figure 3.14 show that prefetch group size has a strong influence the performance of all queries, with the exception of Q1 and Q6. Q6's performance remains constant across prefetch groups because it contains only a sequential scan. In the previous section, we showed that Q1 is insensitive to the output vector size. But we now also see that Q1 is not affected by this other parameter as well. We attribute this to the fact that the only data structure that is prefetched in Q1 (i.e.,

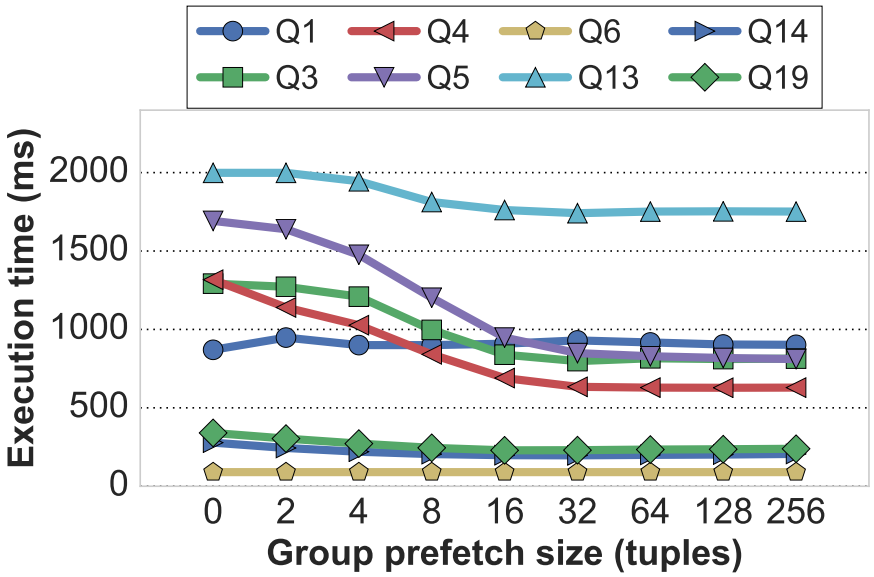


Figure 3.14: Sensitivity to Prefetching Distance – The execution time of the TPC-H queries when varying the ROF model's group prefetch size.

aggregate hash-table) fits in the CPU's L1 cache. In fact, Figure 3.14 shows that Q1's lowest execution time is when there is no prefetching at all. This indicates that the combination of the weak predicate and a small hash-table makes Q1 ill-suited for ROF. We note, however, that ROF does not degrade the performance of Q1 (as seen in Figures 3.7 and 3.8) since the predicate, though weak, can still use SIMD vectorization.

Our second observation is that all of the queries get faster with increasing group sizes up until 16 tuples. The CPU used in our evaluation supports a maximum of 10 outstanding L1 cache references (per core), and thus one would expect the optimal group size to be 10 since this should saturate the CPU's memory-level parallelism. These results, however, show that the optimal group size is 16. This is because the GP technique that we implemented is also limited by instruction count. Using larger group sizes enable fewer iterations of the outer-most loop, which reduces overall instruction count. Hence, groups larger than 16 do not improve performance because at that point the DBMS saturates the CPU's available memory parallelism.

3.3.6 Multi-threaded Execution

We now evaluate the performance of Peloton with ROF when executing queries using a multiple threads. We implemented a simplified version of the multi-threaded execution strategy employed in HyPer [90]. Each pipeline in the query plan is executed using multiple threads that each modify only thread-local state. At pipeline-breaking operators, a single coordinator thread coalesces data produced by each execution thread. As an example, during a building phase of a hash-join, each execution thread constructs a thread-local hash-table based on its assigned input partitions. The coordinator thread constructs a global hash-table whose keys and values are pointers into each execution thread's local storage. The constructed global hash-table is used in the subsequent pipeline to perform the probe-side of the join. Since the global hash-table is read-only after construction,

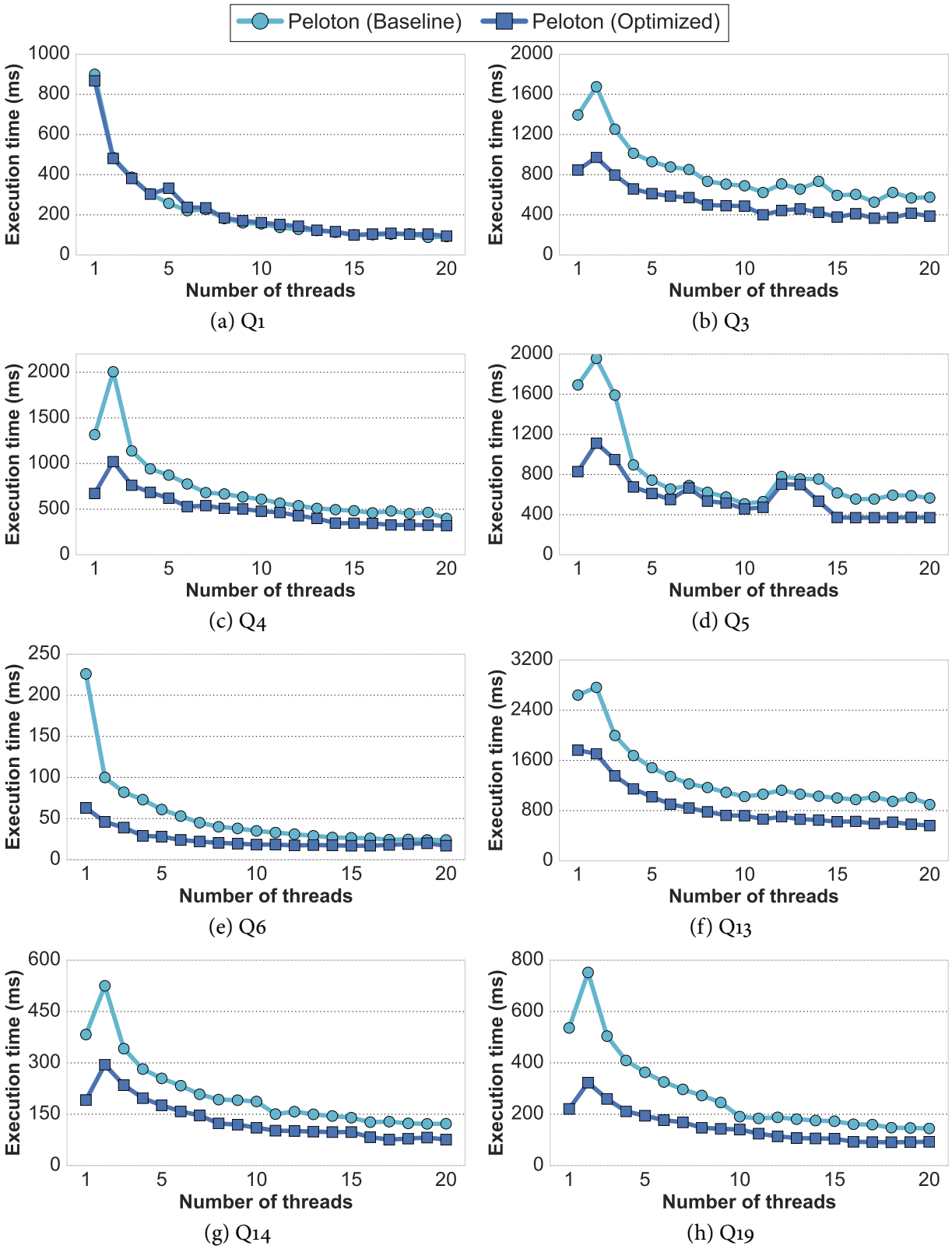


Figure 3.15: Multi-threaded Execution – The performance of Peloton when using multiple threads to execute queries with and without the ROF model.

execution threads need not synchronize during the probe phase. We note that Peloton does not adopt a NUMA-aware data placement/allocation policy.

We ran the eight TPC-H queries from Section 3.3.2, using each query's best staged plan. We vary the number of execution threads from one to the number of physical cores in the benchmark machine. We report averages over ten runs using a TPC-H SF10 database.

The results in Figure 3.15 demonstrate that using ROF with vectorization and prefetching complements multi-threaded query execution. We first note that the jump in execution time when moving from one thread to two threads for all the queries is due to the non-negligible bookkeeping and synchronization overhead that is necessary to support multi-threaded execution. This is independent of the ROF model. As described earlier, hash-joins end at a synchronization barrier on the build-side as execution threads wait for the coordinator thread to construct a global hash-table. For low thread counts, this overhead outweighs the benefit of multiple threads. But this cost is eliminated with the addition of more execution threads.

Figure 3.15a shows that ROF does not improve Q1 since it is a CPU-bound query with a high-selectivity predicate. This corroborates our previous results in Sections 3.3.3 to 3.3.5. Executing Q1 with multiple threads yields a consistent increase up to 20 cores at which point all CPUs are fully utilized and have saturated the memory bandwidth. We note that multi-threaded execution benefits both the unoptimized plan and the ROF plan leveraging SIMD predicate evaluation.

The other seven TPC-H queries exhibit similar speed-up with increasing thread counts. Although each execution thread constructs a small thread-local hash-table, the size of the coordinator thread's global hash-table will always exceed the CPU cache size. Since the join's probe-side is usually an order-of-magnitude larger than the build-side, Figures 3.15b and 3.15f to 3.15h shows that using ROF with prefetching improves performance by 1.5–1.61× over the baseline.

One final observation is the slight variance in execution times in Q3 and Q13 with more than 10 threads. This jitter is due to NUMA effects on our two CPU machine (10 cores per socket). Both Q3 and Q13 execute using a group hash-join, meaning that the DBMS uses the materialized hash-table during building and probing. Concurrent updates to the table are serialized using 64-bit compare-and-swap instructions. Hence, CPUs in different NUMA regions experience different latency when accessing these counters stored in the global hash-table. Q5 exhibits this effect as it requires two global hash-tables probes (i.e., four random memory accesses). Despite this, ROF is still improves performance by 1.5×.

3.3.7 System Comparison

Lastly, we compare Peloton with ROF against two state-of-the-art in-memory databases: Vector [1] and HyPer [77]. The former is a columnar DBMS based on MonetDB/x100. It uses a vectorized execution engine that supports both compressed tables and SIMD instructions when available. The latter is also a columnar DBMS, but uses LLVM (like Peloton) to generate compiled tuple-at-a-time query plans. For Peloton, we execute the queries both with and without our ROF model enabled; this corresponds to the "baseline" and "optimized" configurations from Section 3.3.2.

We deployed all of the DBMS using the same hardware and database as described in Section 4.3.1. To ensure a fair comparison, we disabled multi-threaded execution in all of the systems and made a good faith effort to tune each one for TPC-H. We note, however, that both Vector and HyPer include

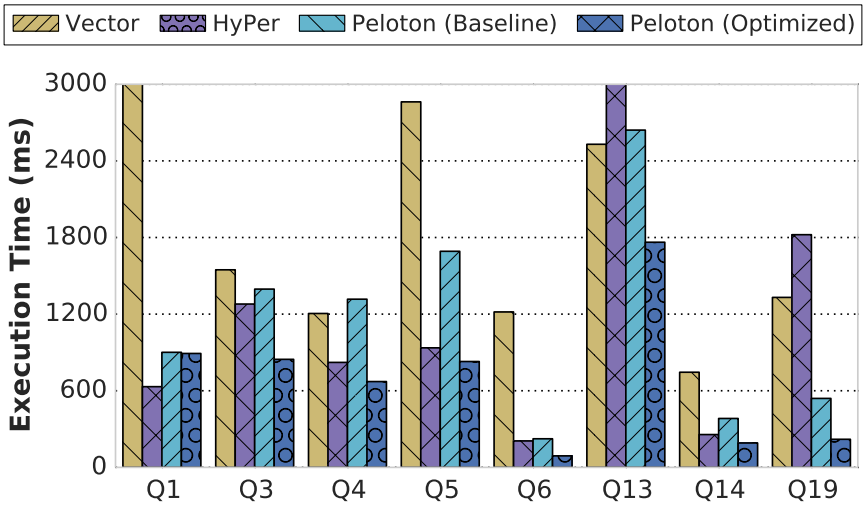


Figure 3.16: System Comparison – Performance evaluation of the TPC-H benchmark in Vector, HyPer, and Peloton with and without our ROF model.

additional optimizations that may not exist across all three DBMSs. Thus, we tried to ensure the query plans generated in each system are equivalent or at least do not differ too greatly. We warm up each DBMS first by executing all of the TPC-H queries without taking measurements.

The results of this experiment are shown in Figure 3.16. We now provide an analysis of the eight TPC-H queries:

Q1: HyPer performs the best in Q1, completing almost 1.38× faster than Peloton, and more than 6.5× faster than Vector. This is because HyPer uses fixed-point decimal arithmetic rather than more computationally expensive floating point arithmetic.

Q3: Peloton with our ROF technique outperforms Vector and HyPer by 1.8× and 1.5×, respectively. Additionally, we see that Peloton without our ROF technique is comparable to HyPer since they both use the same push-based compiled queries. Q3 contains two joins, both of which access a hash-table that does not fit in cache if the join is not partitioned using early materialization. This means that every access to the hash-table is a cache-miss. Our ROF technique hides this cache-miss latency by overlapping computation and memory accesses of different tuples. While (radix) partitioning-based joins is another strategy to solve this problem, previous work has shown that the subsequent overhead of gathering attributes necessary for operators following the join negates the benefits of gained by in-cache joins [133]. Second, Vector executes the `LINEITEM` predicate using a SIMD scan. This version of HyPer does not implement SIMD scans, though this is addressed in later work [88].

Q4: Peloton with our ROF technique outperforms both Vector and HyPer by 1.8× and 1.2×, respectively. Peloton without ROF has performance comparable to Vector, but is outperformed by HyPer. The primary reason for this is because HyPer uses CRC32 for hashing, implemented using SIMD (SSE4), whereas Peloton uses the more computationally intensive MurmurHash3. Thus, the

cache benefits afforded by prefetching with ROF are slightly offset by the higher instruction overhead (in comparison to HyPer) due to a more complex hash function. In general, Peloton with ROF improves performance over the baseline by more than 2×.

Q5: Peloton with ROF is roughly 3.4× faster than Vector and 1.1× faster than HyPer. Q5 stresses join performance as it contains a five-way join between the largest tables in the benchmark. Hence, prefetching plays an important role as materialized join-tables will exceed the size of cache. Peloton with ROF (and prefetching) improves baseline performance by over 2×, but only offers a small improvement over HyPer. As in Q4, this is due to the simpler CRC32 hash function employed by HyPer. The scan over `LINEITEM` contains no restrictions before probing the hash-table on `ORDERS`, hence the majority of time is spent performing hash computations in Peloton. This higher instruction count in comparison to HyPer offsets the benefits of prefetching.

Q6: Peloton with ROF outperforms Vector by 5.4× and HyPer by 2.3×. Q6 is a sequential scan with a highly selective predicate (1.2%). Hence, leveraging SIMD predicate evaluation yields significant performance improvements, more than 2× in Peloton. The version of HyPer we use does not include the SIMD optimizations [88], but we believe it will also enjoy similar benefits of SIMD.

Q13: In Q13, we see that Peloton without ROF performs worse than Vector, but when we enable ROF to remove the cache miss penalties incurred during the join it performs roughly 1.4× faster than Vector. We note here that the majority of time spent in executing this query is in the scan of the `ORDERS` table. This is because the scan involves a `LIKE` clause and thus the query's performance hinges on the performance this evaluation. We note that Peloton uses a simple comparison for the `LIKE` function that assumes clean input data. The slower results for Vector and HyPer suggest to us that they are likely using more sophisticated implementations that are able to handle problematic data better (e.g., broken UTF encodings).

Q14: This query contains a highly selective scan on `LINEITEM` that benefits from a SIMD implementation. Peloton and Vector are able to take advantage of this optimization, but HyPer (in this version) and Peloton without ROF must execute a scalar scan. We note that both Peloton and Vector implement Q14 with a hash-join, whereas HyPer uses an index nested-loop join. This is the reason why Peloton without ROF is slower than HyPer since the probe phase of the join will incur the additional overhead of duplicate chain traversal. But with the addition of SIMD scan and prefetching over the build- and probe-phase of the join, Peloton with ROF performs 3.9× and 1.35× faster than Vector and HyPer, respectively.

Q19: Similar to Q14, Q19 also contains a highly selective scan on `LINEITEM`. But Peloton uses dictionary-encoding for the filtered attributes because their cardinalities are sufficiently small. With dictionary-encoding enabled, Peloton with ROF converts the scalar scan over `LINEITEM` into vectorized scan using SIMD. Vector also automatically compresses strings. With the addition of prefetching and staging, Peloton with ROF executes this query 6× faster than Vector and 8× faster than HyPer.

3.4 Conclusion

We presented the relaxed operator fusion query processing model for in-memory OLAP DBMSs. With ROF, the DBMS introduces staging points in a query plan where intermediate results are temporarily materialized to cache-resident buffers. Such buffers enables the DBMS to employ various optimizations to exploit inter-tuple parallelism using a combination of vectorization and software prefetching. This allows a DBMS to support faster OLAP query execution and to support vectorization optimizations that were previously not possible when data sets exceed the size of CPU-level caches. We implemented our ROF model in the Peloton in-memory DBMS and showed that it reduces the execution time of OLAP queries by up to 2.2×. We also compared Peloton with ROF against two other in-memory DBMSs (HyPer and Actian Vector) and showed that it achieves 1.8× lower execution times.

Chapter 4

Permutable Compiled Queries

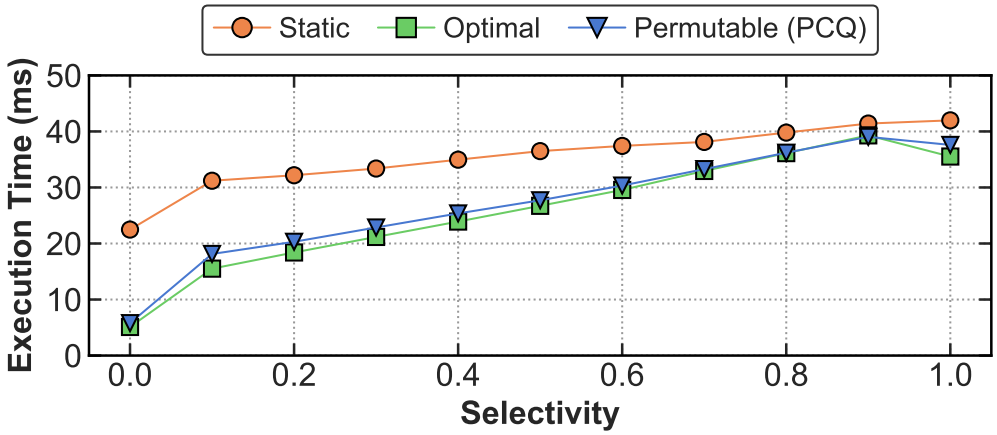
Although query compilation can accelerate the execution of a query plan, existing compilation techniques cannot overcome poor choices made by the DBMS's optimizer when constructing that plan. Sub-optimal choices by the optimizer arise for several reasons, including: (1) the search spaces are exponential (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 [91].

One approach to overcoming poor choices by the optimizer is *adaptive query processing* (AQP) [46], which introduces a dynamic feedback loop into the optimization process. While effective in interpreter-based DBMSs, AQP is infeasible in DBMSs that use JIT query compilation for two reasons. First, compiling a new query plan is expensive: often on the order of several hundreds of milliseconds for complex queries [82]. 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 may need to make adjustments repeatedly throughout the query's lifetime.

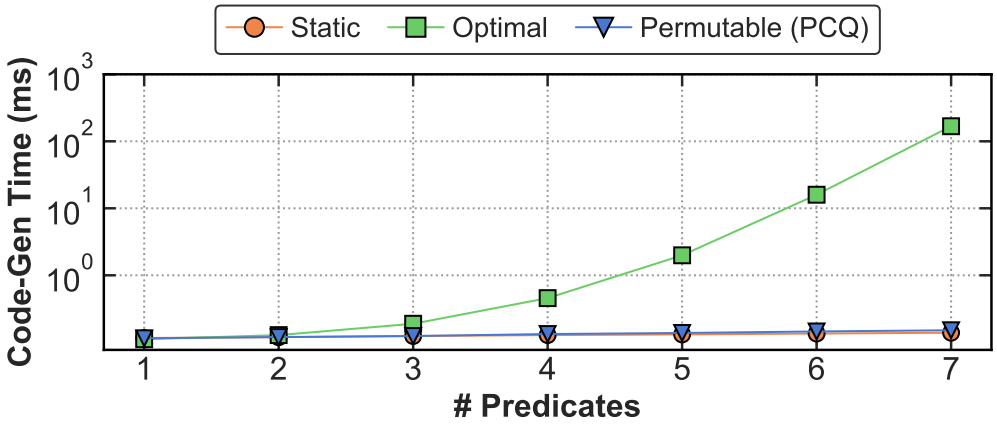
Although there are clear benefits to AQP, generating a new plan or including alternative pipelines in a query is not ideal for compilation-based systems. Foremost is that compiling a new plan from scratch is expensive. But even if the DBMS's optimizer pre-computed all variations of a pipeline before compiling the query, including extra pipelines in a plan increases the compilation time. The DBMS could compile these pipelines in the background [82], 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



(a) Execution Time



(b) Code-Generation Time

Figure 4.1: Reoptimizing Compiled Queries – PCQ enables near-optimal execution through adaptivity with minimal compilation overhead.

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 Section 5.4.

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.

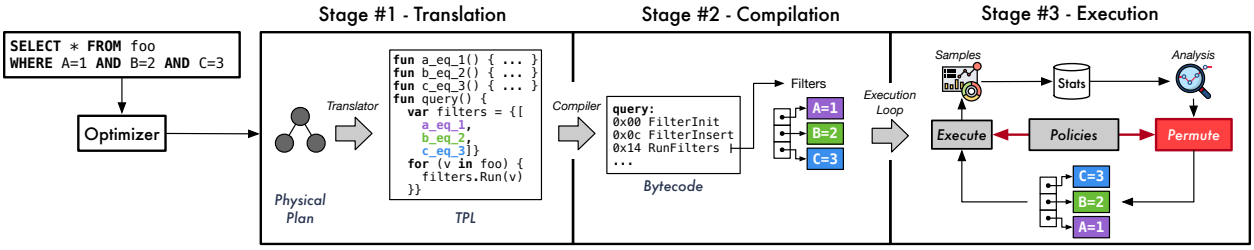


Figure 4.2: System Overview – The DBMS translates the SQL query into a DSL that contains indirection layers to enable permutability. Next, the system compiles the DSL into a compact bytecode representation. Lastly, an interpreter executes the bytecode. During execution, the DBMS collects statistics for each predicate, analyzes this information, and permutes the ordering to improve performance.

The results in Figure 4.1a 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 Figure 4.1b 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.

4.1 Overview

To overcome the gap between the rigid nature of compilation and fine-grained adaptivity, this chapter presents a system architecture and AQP method for JIT-based DBMSs to support **Permutable Compiled Queries** (PCQ). The key insight in PCQ is our novel compile-once approach: rather than invoking expensive compilation multiple times or pre-compiling multiple physical plans, with PCQ we compile a single physical query plan. But we design this single plan so that the DBMS can easily permute it later without significant recompilation to support fine-grained adaptivity.

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 [24] 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 dy-

namic 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 Figure 4.2. As we discuss below, the life-cycle of a query is broken up into three stages.

Stage #1 - Translator After the DBMS's optimizer generates a physical plan for the query, the *Translator* converts the plan into a domain-specific language (DSL), called TPL, that decomposes the plan into pipelines. Using TPL enables the DBMS to reason about and apply database-specific optimizations more easily than a general-purpose language (e.g., C/C++). As we will describe later, another benefit of using this restricted DSL is that it has low-latency compilation times.

TPL mixes Vectorwise-style pre-compiled primitives [29] with HyPer's JIT query compilation through pipelines [108]. It provides pre-compiled and vectorized builtin functions. The Translator generates these vectorized calls whenever possible (e.g., for vectorizable predicates), and reverts to tuple-at-a-time processing otherwise (e.g., for predicates involving complex arithmetic). Leveraging both techniques enables the DBMS to reduce or hide the overhead incurred in supporting PCQ, while achieving high performance.

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 Figure 4.2 to collect metrics for evaluating `WHERE` clause predicates. The DBMS can toggle this collection on and off 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 query in Figure 4.2, the Translator organizes the predicates in an array with hooks that allow the DBMS to rearrange their evaluation order. For example, the DBMS could choose to 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 the execution strategy for this part of the query only involves lightweight pointer swapping.

Stage #2 - Compilation In the second stage, the *Compiler* converts the DSL query 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 Figure 4.2, 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.

Stage #3 - Execution After converting the query plan to bytecode, the DBMS uses adaptive execution modes to achieve low-latency query processing [82]. The DBMS begins execution using a virtual machine interpreter. Simultaneously, a background thread compiles the bytecode into native machine code using LLVM. Once the background compilation terminates, the function pointers (shown in Figure 4.2) are atomically swapped to point to compiled code. Thus, different versions of a function may run simultaneously during query processing by different threads, or between consecutive invocations in a single thread. Although this mechanism is not strictly necessary to support PCQ, it reduces the latency of short running queries.

During execution, the plan's runtime data structures use policies to selectively enable lightweight metric sampling. In Figure 4.2, 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 achieve the smallest execution time given the current data distribution. Each execution thread makes an independent decision since they operate on distinct 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. Since JIT code is faster than bytecode, the DBMS may observe varying runtimes between invocations of the same function when it switches from interpreted mode to compiled mode assuming similar data characteristics.

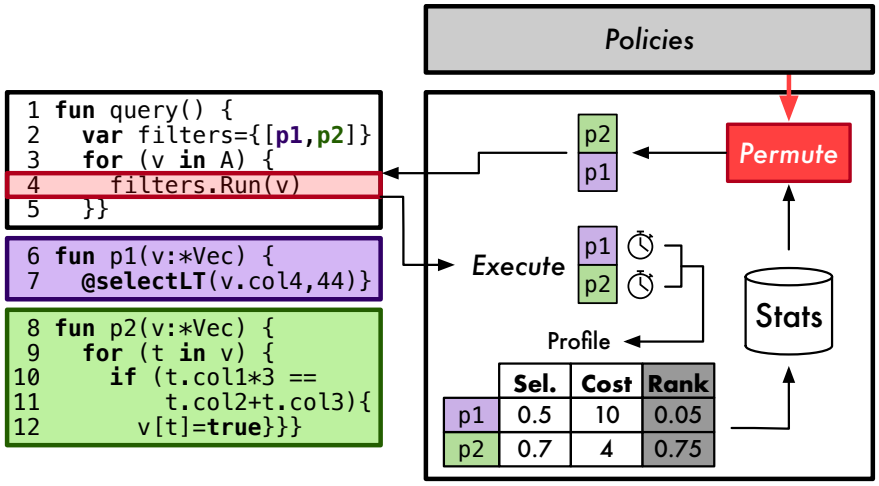
The DBMS uses a pull-based batch-oriented engine that combines vectorized and tuple-at-a-time execution in the same spirit as Relaxed Operator Fusion (ROF) [101]. Batch-based execution allows the DBMS to amortize the cost of expensive auxiliary operations related to PCQ, including those to collect statistics, perform permutations, and execute non-inlined function calls. It uses ROF to implement hash table operations for joins and aggregations with prefetch-enabled code paths. Likewise, the DBMS uses ROF to support vectorized predicate evaluation with SIMD.

4.2 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.

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

(a) Example Input SQL Query



(b) Generated Code and Execution of Permutable Filter

Figure 4.3: Filter Reordering – An example depicting PCQ permutable filters. The Translator converts the query in (a) to the program 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.

For each category, we describe the steps for supporting PCQ. We first discuss 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 each optimization that it uses for policy decisions.

4.2.1 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 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. We use Figure 4.3 as a running example through this discussion.

Preparation / Code-Gen: Consider the SQL query in Figure 4.3a. 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 Figure 4.3a 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 Figure 4.3b, `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 directly in `TPL`. We note that LLVM will automatically vectorize `p2` using SIMD instructions.

Lastly, line 2 in Figure 4.3b 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 Figure 4.3b. 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, and the one employed in this work, is to sample selectivities and runtimes randomly with a fixed probability p. There are more sophisticated policies, and we wish to pursue them as part of this thesis.

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.

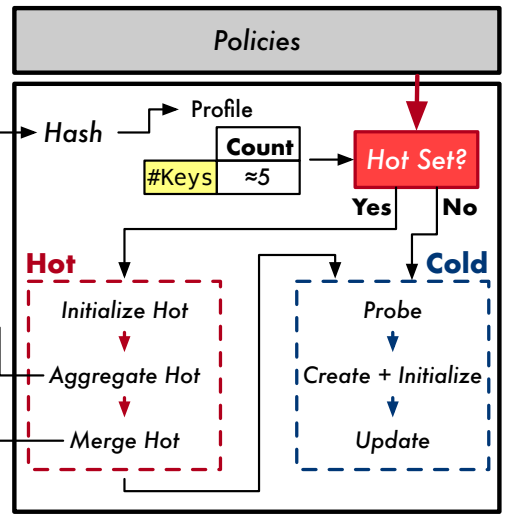
4.2.2 Adaptive Aggregations

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. Each step takes in a vector of

```
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) {
8     aggregator.Run(v)
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]}
```



(b) Generated Code and Execution of Adaptive Aggregation

Figure 4.4: Adaptive Aggregations – The input query in (a) is translated into TPL on the left side of (b). The generated TPL uses a runtime aggregation data structure that is templatized with query-specific logic in addition to new customized functions to handle heavy-hitter groups. The right side of (b) steps through one execution of a batched aggregation.

input tuples and potentially extra intermediate data vectors. 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.We use Figure 4.4 as a running example in this section.

Preparation / Code-Gen: The Translator first creates a specialized function to handle the hot keys. This function, `aggregateHot` on lines 10–16 in Figure 4.4, 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. The program then scans table `A` in batches on lines 7–9, and processes the aggregation for batch on line 8. After scanning all tuples, the algorithm adds the hot keys into the main hash table. For this, the Translator generates a merge function, `aggregateMerge` on lines 17–19, that takes a list of partial aggregates and adds them to the hash tables. As before, because N is a compile-time constant, the Translator unrolls and inlines the merge logic for the N aggregates into the function.

Since the array of heavy-hitter aggregates is provided uninitialized, the first step is to initialize them with what we believe to be the "hottest" keys in the current batch. This process also relies on the policies available to the DBMS. One simply policy is to use the first N unique keys in the batch as the "hot" set. Another option is to randomly sample from within the current batch until N unique keys are found. A more sophisticated policy is one that tracks the temperature of keys and inserts them in the array by decreasing key temperature.

Once the heavy-hitter set has been initialized with N keys, the translator generates a loop that iterates over each tuple in the batch and checks for a key equality match against one of the aggregate elements in the heavy-hitter set. This is done by generating N conditional branches in the body of the loop. Tuples that find a match update their aggregates according to the query. Those that don't will fall through to the normal process.

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) [53]. 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.

4.2.3 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 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 [113]); otherwise, it constructs a bucket-chained hash table with pointer-tagging [90]. 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 [113]. We now describe how to implement permutable joins using Figure 4.5 as the running example.

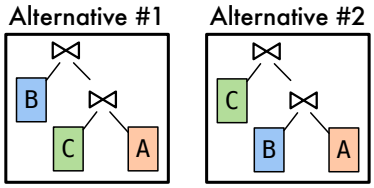
Preparation / Code-Gen: The DBMS's optimizer supports permutable joins in right-deep query plans containing consecutive joins, as in Figure 5.3a. The system designates one table as the "driver" that it joins with one or more tables (i.e., one per join). The DBMS may use either hash or index joins depending 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 Figure 4.5b, 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 Figure 4.5c, `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.

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

(a) Example Input SQL Query

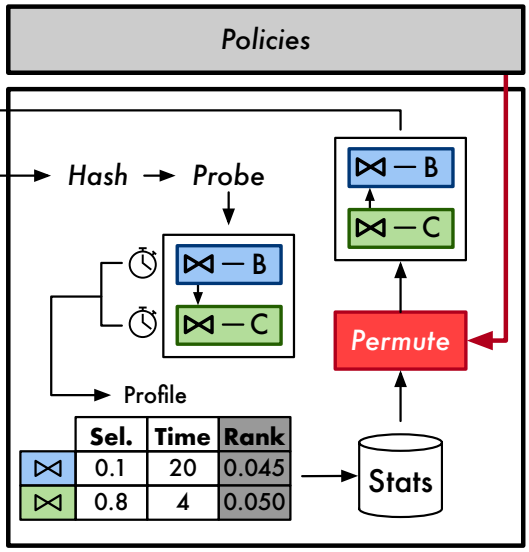


(b) Possible Join Orderings

```
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) {
7     joinExec.Run(v)
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)}
```



(c) Generated Code and Execution of Permutable Joins

Figure 4.5: 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.

Runtime Permutation: The objective of the DBMS is to dynamically order the joins so as to minimize the overall execution time. This process is depicted in Figure 4.5c. During execution, the DBMS first computes a hash value for each tuple in the input batch. The attribute indexes provided earlier during initialization identify the columns to hash and their order. 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.

4.3 Experimental Evaluation

We now present a brief analysis of the PCQ method and corresponding system architecture. We implemented our PCQ framework and execution engine in the NoisePage DBMS [9]. NoisePage is

a PostgreSQL-compatible HTAP DBMS that uses HyPer-style MVCC [110] over the Apache Arrow in-memory columnar data [93]. Each Arrow block of tuples is 1 MB allocated using `jemalloc`. 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) and 128 GB of DRAM. This processor supports AVX512 instructions and contains ten line-fill buffer (LFB) slots that each support one outstanding memory prefetch request. We implemented our microbenchmarks using the Google Benchmark [5] library; it runs each experiment a sufficient number of iterations to get a statistically stable execution time results.

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.

4.3.1 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 Section 4.2: (1) a scan query with three predicates, (2) an aggregation query with groupings, and (3) a multi-way join query.

TPC-H: This is a decision support system workload that simulates an OLAP environment [146]. 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 [12]. We select nine queries that cover the TPC-H choke-point categories [28] 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 [112]. 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.

4.3.2 Filter Adaptivity

In this experiment, we measure PCQ's ability to optimize and permute filter ordering in response to shifting data distributions. As the query executes, the selectivity of its predicates changes over

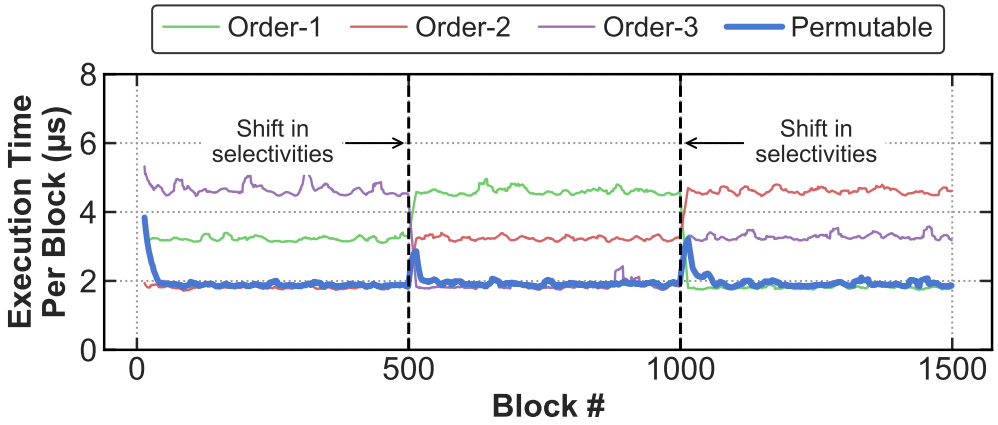


Figure 4.6: Performance Over Time – Execution time of three static filter orderings and the PCQ filter as we perform a sequential scan over a table.

time, thereby changing the optimal order. We use the microbenchmark workload with a `SELECT` query that performs a sequential scan over a single table:

```
SELECT * FROM A
WHERE col1 < C1 AND col3 < C2 AND col3 < C3
```

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 entire table.

We execute this query with the filter reordering optimization using a 10% sampling rate policy (i.e., the DBMS collects performance metrics per block with a probability of 0.1). 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 Figure 4.6 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 each transition. The performance of PCQ is suboptimal in the beginning because it is initially configured with a random ordering. However, within ten blocks of data, PCQ re-samples its selectivity metrics and permutes itself to the optimal ordering. During the transition periods after the distribution changes at blocks #500 and #1000, PCQ incurs some jitter in performance. This variance is because the DBMS is executing the previously superior ordering. After re-collecting statistics, however, the DBMS changes the ordering to find the new optimal configuration in about 10–15 blocks. As such, the PCQ query is ∼2.5× faster than any of the static orderings.

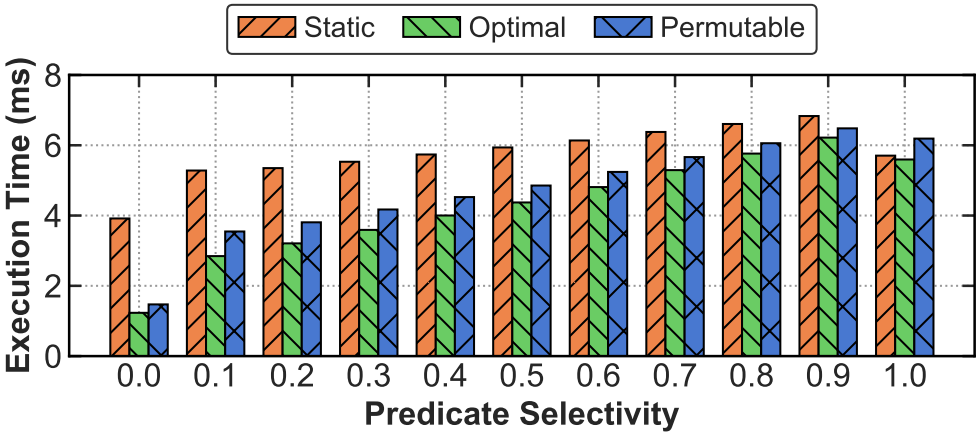


Figure 4.7: Varying Predicate Selectivity – Performance of the static, optimal, and permutable orderings when varying the overall query selectivity.

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 Figure 4.7 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 (i.e., the one yielding fewest output tuples) 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 Section 4.2, 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 Figure 4.6. The query starts with the Order-3 plan from Figure 4.6 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.

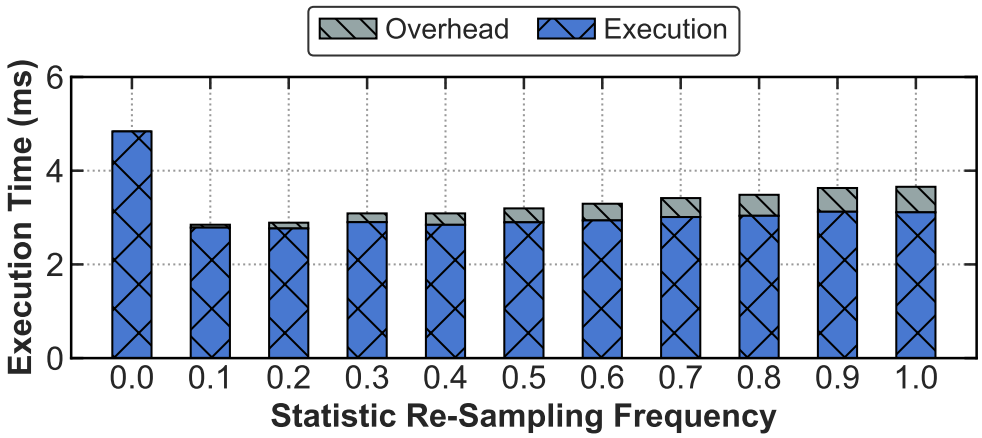


Figure 4.8: Filter Permutation Overhead – Performance of the permutable filter when varying the policy's re-sampling frequency and fixing the overall predicate selectivity to 2%.

The results, shown Figure 4.8, 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 at a 0.1 (i.e., 10%) sampling rate and thus, we use this setting for the remaining experiments.

4.3.3 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 [108]. The static plan represents how existing JIT-based DBMSs execute the query.

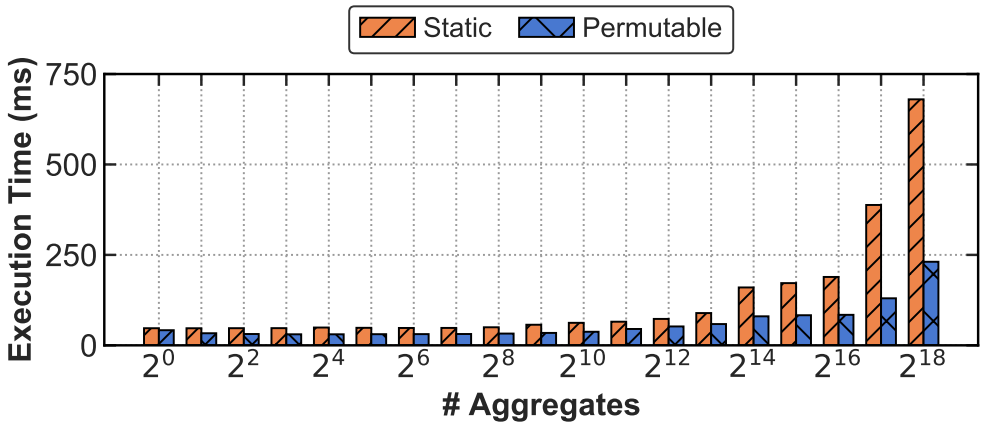


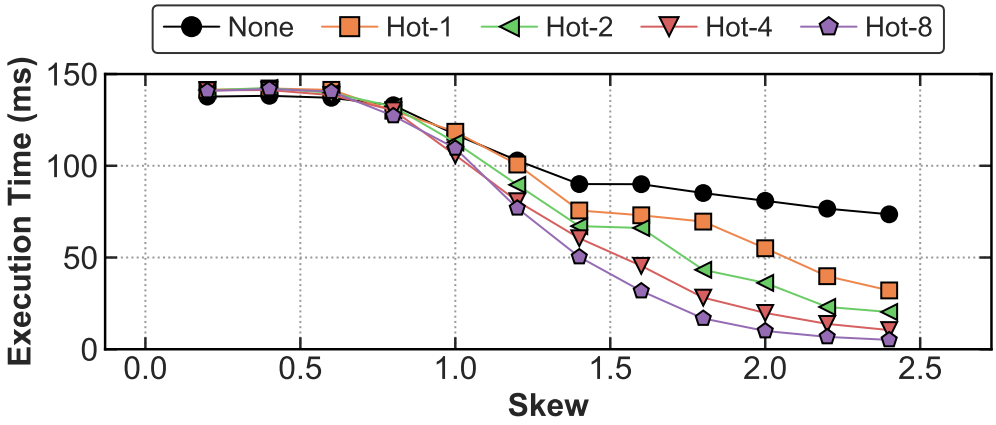
Figure 4.9: Varying Number of Aggregates – Performance of the adaptive aggregation as we vary the total number unique aggregate keys.

The results are shown in Figure 4.9. 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 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. Figure 4.9 shows that this occurs at ∼256k keys where PCQ is 3× faster than the static plan.

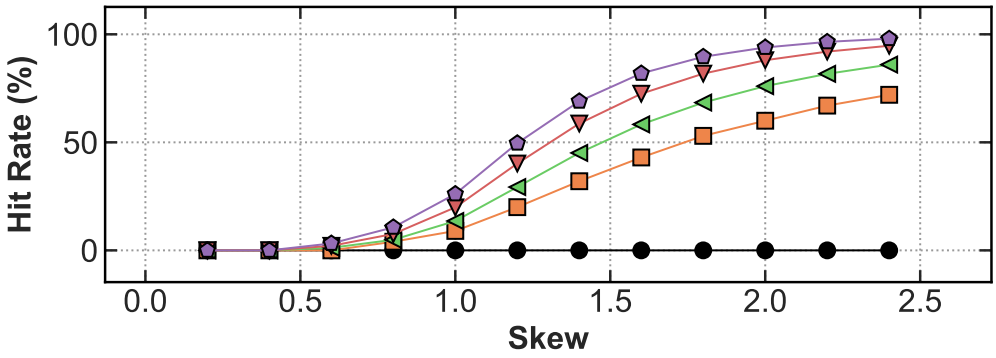
Varying Aggregation Skew: The DBMS must be careful when deciding how many heavy-hitter keys to extract into specialized JIT code for permutable aggregations. Extracting more keys (1) introduces the possibility of branch mispredictions that increase runtime, and (2) generates larger functions that increase 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 Figure 4.10a show that the configurations are within 3% of each other for low skew values (i.e., less than one). 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. At skew level 1.0, None performs 1.15× worse that all other configurations. 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.



(a) Performance



(b) Hit Rate

Figure 4.10: 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.

Figure 4.10b 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.

4.3.4 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.

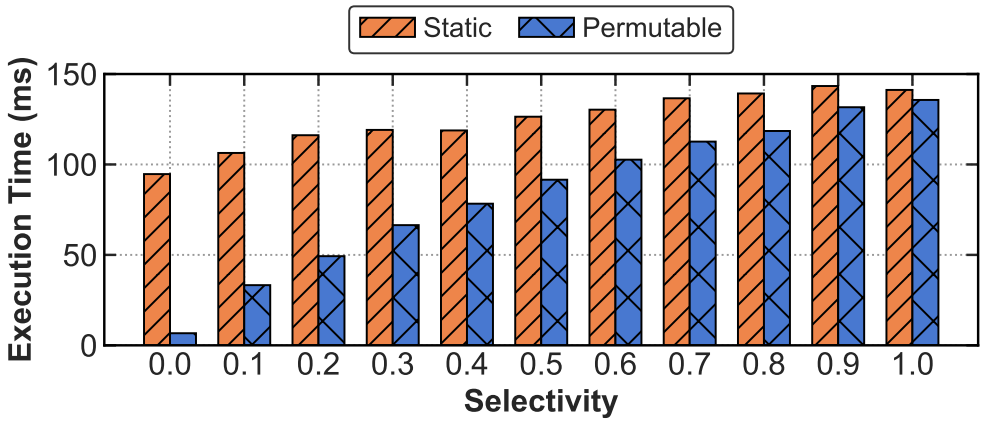


Figure 4.11: Varying Join Selectivity – Execution time to perform three hash-joins while varying overall join selectivity.

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 [108]. 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.

Figure 4.11 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. Some of the benefits of this vectorization, however, are negated by the sampling overhead incurred to support permutability.

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 Figure 4.12 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 auto-vectorization of the key-equality check function. Although the overall selectivity is

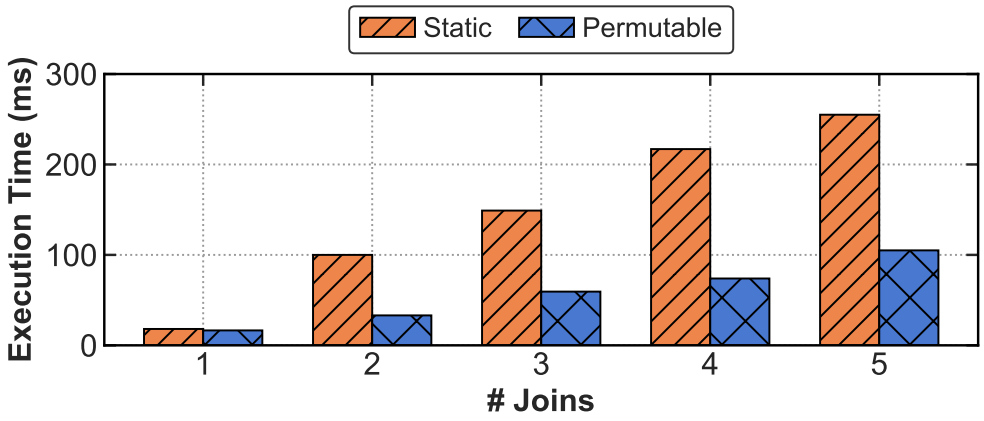


Figure 4.12: Varying Number of Joins – Execution time to perform a multi-step join while keeping the overall join selectivity at 10%.

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. Beyond two joins, PCQ outperforms the static plan even though the overall selectivity is constant. As before, this difference is due to the harmonized combination of vectorized and JIT code used in NoisePage's join processor. PCQ is 3× faster than static when performing two joins, and 2.5× faster when performing greater than three joins.

4.3.5 System Comparison

Lastly, we compare NoisePage with and without PCQ against two state-of-the-art in-memory databases: Actian Vector (v5.1) and Tableau HyPer (v2019.2.18). Vector [1] is a columnar DBMS based on MonetDB/x100 [29] 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 [6] 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 Star Schema Benchmark 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.

Skewed TPC-H

We evaluate the TPC-H benchmark using Microsoft's skewed data generator [12], using a skew of 2.0 (i.e., high-skew). We load the data into each system, warm system and DBMS caches, and take the average over five consecutive executions after dropping the first. The results are shown

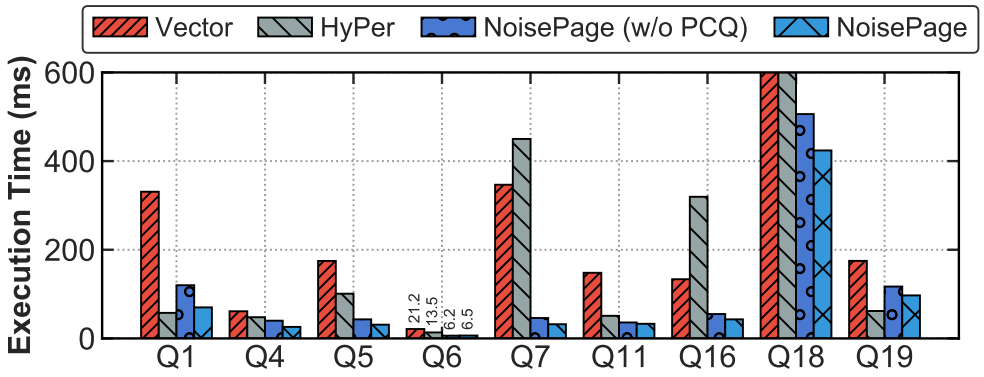


Figure 4.13: System Comparison on Skewed TPC-H – Evaluation of NoisePage, HyPer, and Vector on the *skewed* TPC-H benchmark.

Query	+Filters (§4.2.1)	+Aggregations (§4.2.2)	+Joins (§4.2.3)
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 4.1: TPC-H Speedup – The speedup achieved when incrementally applying each PCQ optimization to TPC-H queries.

Figure 4.13. We also show the effect of each optimization in Table 4.1. 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 4.1 shows that the bulk of the benefit is attributed to the optimized aggregation. Overall, NoisePage with PCQ is 1.8× and 2.3× faster than HyPer and Vector, respectively.

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, to run the highest rank term first (i.e., the `IN`-clause on `p_size`), 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.

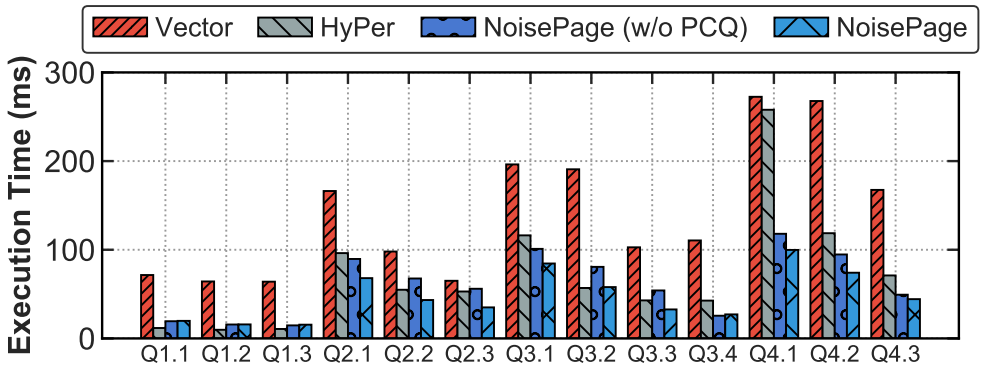


Figure 4.14: System Comparison on the Star-Schema Benchmark – Evaluation of NoisePage, HyPer, and Vector on the Star Schema Benchmark.

Query	+Filters (§4.2.1)	+Aggregations (§4.2.2)	+Joins (§4.2.3)
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 4.2: Star Schema Benchmark Speedup – The speedup achieved when incrementally applying each PCQ optimization to Star Schema Benchmark queries.

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

Star Schema Benchmark

This experiment evaluates all systems on the Star Schema Benchmark [112]. We use the default uniformly random data generator and plot the average execution time of each query over five consecutive executions after discarding the first. The overall results are shown Figure 4.14 along with the benefit breakdown in Table 4.2. 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 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.

4.4 Conclusion

We 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 Star Schema Benchmark benchmark workloads.

Chapter 5

Progressive Code Generation

Chapters 3 and 4 presented techniques to improve robustness during query processing. However, like existing compilation approaches, our techniques share a key characteristic: the DBMS generates all the code necessary for a query before execution. We refer to this strategy as *one-shot* code generation. One-shot generation maximizes the compiler's optimization capabilities since all query code is present at the same time. However, such an approach is limited in its ability to exploit features of the underlying data that a query accesses since the DBMS can only learn this information during execution after compilation has completed. For example, compressing intermediate data structures may be possible to reduce the DBMS's memory requirements. To do this using a one-shot approach requires the query engine to either (1) generate all possible encoding and decoding schemes a priori, (2) fall back to an interpreted code path, or (3) recompiling the query from scratch. None of these options are desirable [102].

A better approach is for the DBMS to adapt generated code based on the data the query reads. But query compilation and runtime adaptivity have conflicting goals. Compilation seeks to specialize code as much as possible, while AQP strives for flexibility.

Previous proposals have sought to bridge this divide between query compilation and AQP, but are limited in their scope and applicability. Most existing approaches require the generation of *all* query code prior to execution [21, 48, 65, 102, 108, 115, 134, 145, 152]; some initially generate instrumented code and recompile based on the observed data distributions [62]; and others target use-cases such as CSV parsing [132] or filter reordering [62]. These previous techniques are insufficient to enable dynamic, low-latency HTAP workloads.

To overcome these limitations, we now present an incremental query compilation method called **Progressive Code Generation** (PCG). PCG draws inspiration from JIT-compiled dynamic language runtimes to optimize generated code for both the SQL query *and* the data it operates on. This resembles how language JITs (e.g., JVM) optimize a program at runtime based on the data that it accesses. Unlike language JITs, a PCG-enabled DBMS need not perform speculative optimization and deoptimization. Rather, the DBMS generates query code only once, but divides the query plan into fragments that are incrementally generated and executed on demand. The DBMS leverages this step-wise execution pattern to inject custom code to analyze intermediate state produced by the query. It uses this information to fine-tune the code it generates for later parts of the query

without the need for "bail-out" paths (as a conventional JIT) since it definitively knows what the data looks like.

5.1 Knowing the Future

Although there are benefits to code generation, existing techniques fail to tailor generated code to the data a query reads at runtime. This flavor of specialization has historically been the hallmark of modern JIT language runtimes. AQP attempts to bring adaptivity to the database community, but previously proposed techniques target interpretation-based DBMSs. An interpretation-based DBMS can adapt with low overhead by splicing new operators into a physical plan, or embedding multiple sub-plans that it can select dynamically based on runtime conditions. However, a compilation-based DBMS must either pay the cost of compiling new code before it can begin execution, or pre-generate alternative code paths for all possible re-optimization scenarios. Neither option is feasible as it leads to intractable compilation times [102]. These overheads prohibit the use of traditional AQP techniques in compilation-based DBMSs.

In contrast to one-shot generation (i.e., the approach used by most JIT-based DBMSs), PCG interleaves code generation and execution. Deferring code generation as late as possible allows the DBMS to learn properties about the data in earlier parts of the query that it uses to tune the code it generates for later parts of the query. Thus, PCG generates code progressively on demand.

To motivate the benefits of progressive code generation, we conduct an experiment evaluating its performance on a traditional data-warehouse query. Consider a database containing two tables, `A` and `B`, composed of six 64-bit NULL-able integer columns (`col1`–`col6`). Table `A` has 500k tuples and table `B` has 7.5m tuples. The workload contains a single query:

```
SELECT COUNT(*),SUM(A.col2),SUM(A.col3),SUM(A.col4),
     SUM(A.col2*(1-A.col3)),
     SUM(A.col2*(1-A.col3)*(1+A.col4))
 FROM A JOIN B ON A.col1 = B.col1
WHERE A.col5 = δ1 GROUP BY A.col6
```

We generate data such that the filtering constant (δ_1) is 95% selective, arranging `A` on the build side of the join. Although the schema declares all columns as potentially containing `NULL` values, the filtering term also happens to filter them all out. Lastly, we ensure that none of the aggregates can overflow a 64-bit integer. We defer a full description of our experimental setup to Section 5.4.

We measure the time the DBMS takes to execute the above query using three code generation variants: (1) Branch, (2) Flag, and (3) Bit. They differ only in how they handle arithmetic overflows when performing the aggregation. Branch relies on "checked" arithmetic instructions that, in addition to performing the operation, expose CPU flags indicating if a numeric overflow occurred:

```
1 // Perform a checked 128-bit addition.
2 bool AddWithOverflow(i128 a, i128 b, i128 *result);
3
4 i128 sum = 0;
5 for (u64 id = 0; id < batch->size(); id++) {
6   const auto a = batch->a[id];
7   if (AddWithOverflow(sum, a, &sum)) throw OverflowError();
8   // More operations ...
9 }
```

Branch generates all arithmetic using checked operations and guards against overflows by conditionally branching to code that raises an error. Most compilation-based engines use this approach. Flag implements a slightly optimized method by continuing execution despite potential overflow, collecting all flags into a single boolean flag using a logical `OR`. The DBMS checks the overflow flag only at the end of a batch and raises an error if required.

```
1 bool overflow = false;
2 i128 sum = 0;
3 for (u64 id = 0; id < batch->size(); id++) {
4   const auto a = batch->a[id];
5   overflow = overflow || AddWithOverflow(sum, a, &sum);
6   // More operations ...
7 }
8 if (overflow) throw OverflowError();
```

This approach amortizes the overhead of overflow checking across all tuples in a batch. Bit uses a similar approach but uses a single bit to track if any operation overflows. Bits are collected using a bit-wise `OR`. The difference here is that a logical `OR` can benefit from short-circuiting, whereas a bit-wise operation cannot. Since all table columns are declared NULL-able, all versions must include `NULL` checking code, but we also evaluate variants that skip these checks for completeness. Lastly, we evaluate our PCG technique that learns the value domains of columns (through a lightweight analysis) to elide `NULL` checking and overflow logic.

The results in Figure 5.1 show that PCG is 2× faster than all other variants. The non-PCG versions *must* generate code to handle arithmetic overflows and `NULL` values because the DBMS cannot know otherwise at query compilation time. In contrast, PCG learns this by decoupling the generation and execution of the left and right side of the join, and injecting a lightweight analysis phase that executes in between. This analysis adds ∼3% overhead to the execution time, but yields a non-negligible speed improvement. Hence, we contend that eschewing one-shot code generation in favor of an incremental approach offers is the better approach for compilation-based DBMS.

The goal of PCG is to enable a DBMS to dynamically tailor a query's generated code to the data it reads without (1) recompiling any part of a query, (2) discarding partial results, or (3) compiling speculative code. PCG decomposes and stages code generation in a manner that allows earlier parts of the query plan to inform and optimize the generation of later parts of the query. By deferring code generation, the DBMS learns the shape and characteristics of the data a query reads to better

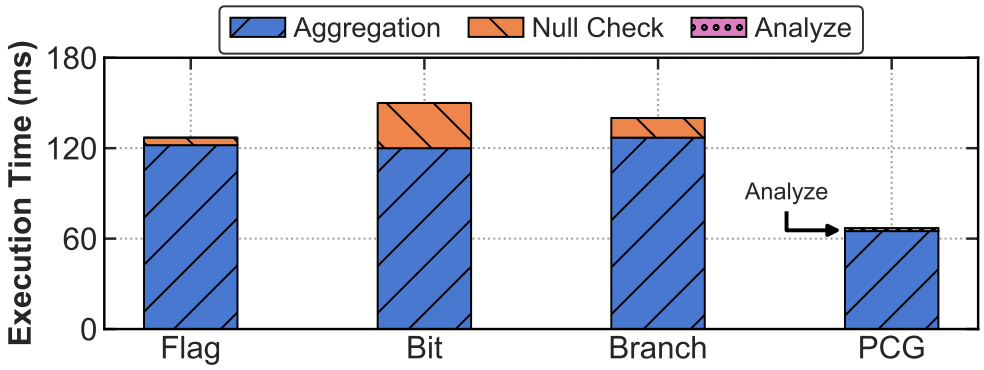


Figure 5.1: Knowing the Future – Deferring code generation enables dynamically specializing instructions to data the query has actually processed.

tailor the code it generates to achieve the desired target objective (e.g., performance or memory consumption).

PCG takes inspiration from JIT-based dynamic language runtimes but differs in two novel ways. First, most JIT runtimes observe the data that a function processes at runtime to specialize only that function for the common case [56]. In contrast, a PCG-enabled DBMS observes and tracks the data a query reads in earlier parts of execution to later tune a *different* part of the query. Second, a JIT runtime must compile both an optimized code path for the case it speculates to be common and a "bail-out" path for exceptional data [155, 156]. The runtime checks all function inputs to guide control flow down the correct path. With PCG, the DBMS *guarantees* that inputs will only ever take the optimized path, thereby eliminating the need to check inputs or generate a "bail-out" path.

5.2 Overview

This section presents an overview of PCG, shown in Figure 5.2. We use the same SQL query from Section 5.1 as a running example. Although we designed the framework for NoisePage's custom DSL-based architecture based on the LLVM, it works with any DBMS execution engine that supports query compilation.

5.2.1 Decomposition

The first step in the compilation pipeline is to decompose a physical query plan into its pipeline components. In the produce-consume code generation model, a pipeline represents a sequence of operators that pass data from child to parent without copying or materializing that data [108]. Pipelines "fuse" their operators' logic and end in a pipeline-breaking operator (i.e., one that fully materializes all intermediate results before continuing processing).

The query in Figure 5.2 contains three pipelines. We assume (1) all joins and aggregations are hash-based and (2) that the left input to a hash join is the build side, and the right input is the probe side. Pipeline **P1** scans table `A`, applies a filter, and builds the hash table, $\bowtie_{A.col_1=B.col_1}$. Pipeline **P2** scans table `B`, probes the join hash table, and performs a grouping using the hash table $\Gamma_{A.col_6}$ while computing all necessary aggregations. Lastly, pipeline **P3** scans the final aggregated tuples and emits them to the desired output.

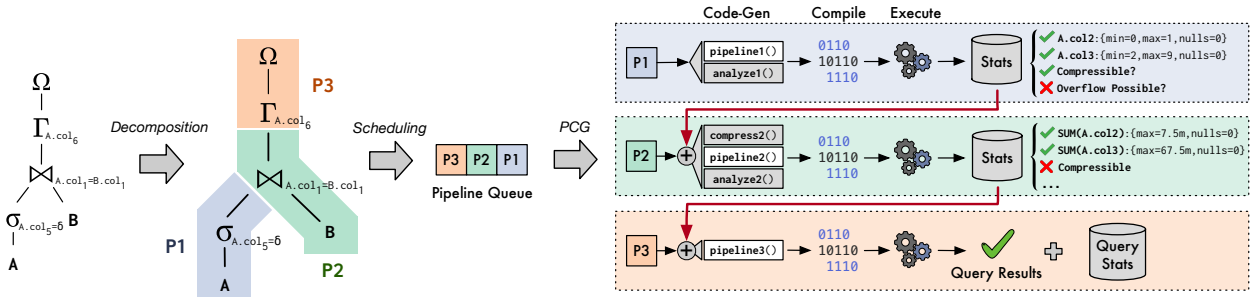


Figure 5.2: System Overview – An overview of the PCG query compilation and execution pipeline. The DBMS first decomposes a physical plan into pipelines to build a pipeline execution graph. Next, the DBMS schedules pipeline fragments for code generation and execution. Each fragment is compiled into a executable bytecode and run adaptively using either an interpreter or native code. In addition to computing query results, fragments collect detailed statistics that are available to subsequent fragments to tune their code.

5.2.2 Scheduling

After decomposing a physical plan into pipelines, a conventional query compiler generates and compiles all code to execute the query. Instead, PCG relies on the observation that pipelines only share state, not code. For example, both **P1** and **P2** require the hash table ($\bowtie_{A.col_1=B.col_1}$), but the logic in **P1** never references **P2**, nor vice versa; this is by design. For a binary operator that materializes its left input (e.g., hash join), the build-side input tuple's schema is different from the probe-side, requiring different code on either side. Most pipeline code fragments are independent, which means that the DBMS can compile and execute them independently[1].

Although PCG decouples the compilation of pipelines, the physical plan still imposes an execution order. Thus, the next step is to determine a valid schedule of pipeline fragments. Before constructing a schedule, the DBMS builds a *pipeline graph* based on their dependencies. A pipeline, $\mathbf{P}_i$, has a directed edge to pipeline, $\mathbf{P}_j$, if $\mathbf{P}_i$ can execute before $\mathbf{P}_j$. For example, a join operator consumes its left input and builds the hash table before the right pipeline can run; thus, the left pipeline induces a directed edge to the right pipeline in the graph. After the DBMS builds the pipeline graph, it uses a topological sort to determine a valid execution order. Unconnected components in the pipeline graph represent pipelines that can run in parallel, but PCG does not currently exploit this optimization. In Figure 5.2, the order of compilation and execution is **P1**, **P2**, and **P3**. The system then inserts these fragments into a code generation queue based on their execution order.

5.2.3 Code Generation and Execution

We now describe how the DBMS interleaves code generation and execution with PCG.

Code Generation: The DBMS processes pipelines in the order they appear in the queue. For each pipeline, the DBMS generates one function encapsulating the fused logic of all operators it contains

[1]Exceptions include set-based relational operator (e.g., `UNION`). NoisePage supports these by nesting their logic within a parent pipeline. In this case, the code generator extracts the nested pipeline's logic into a separate function available across pipelines.

using the data-centric code generation model [108]. PCG extends the model by introducing pre- and post-pipeline functions that execute before a pipeline begins and after it finishes, respectively. We refer to these functions as *bracket functions* to reflect their execution order relative to their associated pipeline. Bracket functions do not implement core query logic. Instead, they can only read the materialized state their pipeline scans or produces after it completes. Some bracket functions may construct a new (reorganized) version of the state that replaces the original before the pipeline runs. At most one pair of bracket functions exist for a pipeline, and is generated only at the request of any contained operators. PCG uses bracket functions to inject custom utility code to achieve a desired query- or operator-specific objective. As we show in Section 5.3, this simple extension is sufficient to enable a wide array of optimizations.

An important post-pipeline function in PCG is the *analysis function*. An analysis function scans and inspects the intermediate state produced by a pipeline after it completes to derive properties about data the query has partially processed. Analysis functions collect information per attribute including domain bounds (i.e., min and max values), approximate distinct value counts, and `NULL` value counts. We limit the types of statistics we collect to these three because the DBMS can collect them with negligible runtime overhead, but PCG allows pipeline operators to generate arbitrary analysis code. Analysis metrics are collected in parallel by partitioning the materialized state across available execution threads and persisting in an in-memory data structure available throughout query processing. Analysis functions are analogous to incrementally running a lightweight `ANALYZE` during query processing.

Analysis functions play a critical role in the PCG framework because they supply the DBMS with fine-grained, specific, and accurate statistics about data the query has actually seen. This differs from the statistics that are available to the DBMS optimizer, which are often coarse-grained at the leaves of the query plan, and inaccurate otherwise [91]. PCG leverages these statistics to rule out certain runtime conditions and specialize code further than a conventional one-shot query compiler.

In Figure 5.2, **P1** wraps all pipeline logic in function `pipeline1()`. The DBMS also generates an analysis function, `analyze1()`, that executes after the pipeline completes. The hash join operator initiated this request to determine whether it can compress the tuples it materialized in its hash table. The results of the analysis, which are available during the generation of **P2**, indicate that the contents of the hash table are indeed amenable to compression. Thus, the join operator generates a pre-pipeline function, `compress2()`, to reorganize tuple data in the hash table to reduce memory overhead. Next, the main logic for **P2** is generated in `pipeline2()`, taking into account the new physical sizes of tuple attributes after compression. Finally, the pipeline breaker in **P2** (i.e., the hash aggregation) also requests an analysis function, `analyze2()`, to examine the tuples in its own hash table.

Compilation: The target of code generation in NoisePage is a custom domain-specific language (DSL), called `TPL`. `TPL` combines Vectorwise-style pre-compiled primitives [29] with HyPer's data-centric code generation [108]. It contains familiar data types and imperative constructs, including variable and structure declarations, loops, conditional control-flow, and functions. `TPL` also provides first-class support for SQL types and the operations they support. Using `TPL` eases the effort required in implementing database-specific optimizations in comparison to a general-purpose lan-

guage (e.g., C/C++). TPL integrates with existing software development tooling like debuggers (e.g., gdb), enabling DBMS engineers to step through code, set breakpoints, and print variable values.

The DBMS next compiles TPL into a compact bytecode representation. This bytecode is a CISC instruction set composed of arithmetic, memory, and branching instructions, combined with more complex database-level instructions. After compilation, the DBMS can immediately begin execution of the bytecode using a bytecode interpreter. The NoisePage compilation pipeline is optimized for query latency. Compiling pipeline fragments into executable bytecode takes 10–50 μs, and is not more than 2–3$\times$ slower than native code. NoisePage also supports HyPer-style adaptive execution by asynchronously compiling the bytecode into native machine code using LLVM [82]. On completion, native code is seamlessly swapped in at runtime to further improve query performance.

Execution: After the DBMS compiles the pipeline into bytecode it begins execution using an interpreter. If a pre-pipeline function is available, it is executed first to prepare the state for the pipeline. This step is almost always run in parallel by partitioning the input state across available threads. Once complete, the core pipeline logic is run to generate the next intermediate state. Finally, any post-pipeline function is executed on the output.

In Figure 5.2, the analysis function in **P1**, `analyze1()`, collects detailed statistics on each of the attributes that are materialized in the hash table during the build phase of the hash join. During code generation for **P2**, the hash aggregation operator uses this information to make two important inferences. First, it can guarantee that overflows for aggregates it computes are impossible given the size of `B` and the inputs' value domains. Second, it knows that all `NULL` values have been filtered out and do not need to be specially handled when updating its aggregates. Thus, during the generation of `pipeline2()`, it safely removes `NULL` checks and uses simple 32-bit addition (rather than "checked" addition) for the aggregation.

5.3 Adaptive Optimizations

This section presents query execution optimizations that are possible using PCG. A PCG-enabled DBMS exploits the independence between a query's pipelines by generating code, compiling, and executing them separately, not simultaneously. Such decoupling exposes well-defined boundaries where the DBMS installs bracket functions that augment query execution to improve robustness.

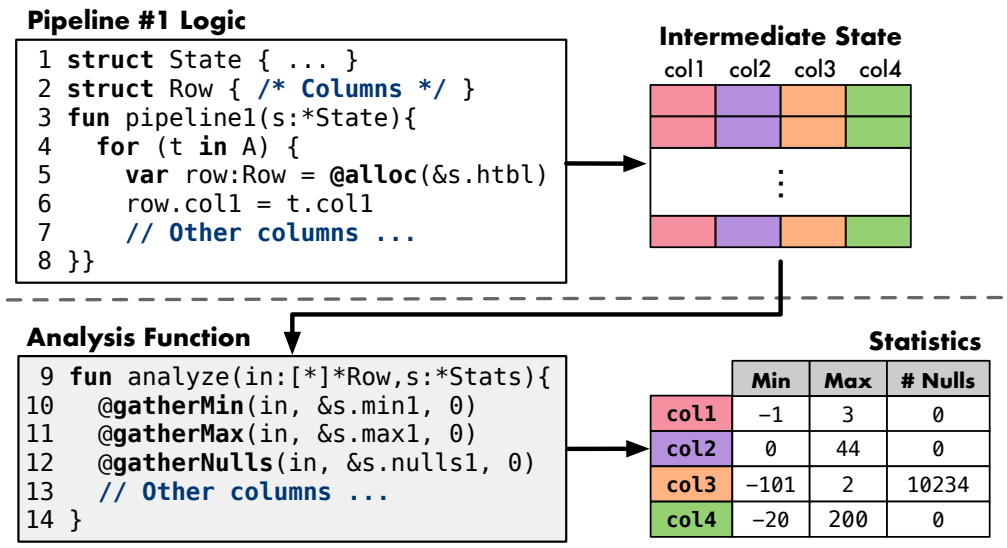
As mentioned in Section 5.2.3, a crucial post-pipeline bracket function is the analysis function. We, thus, begin this section with a description of how analysis functions operate. We then discuss optimizations that leverage the statistics garnered through periodic analysis to improve performance or reduce memory pressure. We describe how the DBMS organizes generated code with these optimizations and how that code executes at runtime. Where appropriate, we distinguish when pipeline code fragments are generated and when they run.

5.3.1 Analyzing State

An analysis function is generated after its associated pipelines' logic function, but only if requested by any operator within the pipeline. We use Figure 5.3 to aid in our discussion and assume that the (hash) join requested an analysis of its materialized hash table.

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

(a) Example Input SQL Query



(b) Generated Code and Execution of an Analysis Function

Figure 5.3: Analyzing State Example – This example shows the interaction of query logic and post-pipeline analysis logic. The DBMS first executes **P1** to produce some intermediate state. Next, the DBMS inspects this state using a pre-generated analysis function that is composed of custom code that calls pre-compiled primitives. This analysis then produces statistics stored that is available to later pipelines during their code generation.

Code Generation: Figure 5.3a shows an example query and Figure 5.3b shows the code the DBMS query compiler generated for the build side of the hash join. The primary pipeline logic resides in the `pipeline()` function on lines 3–8. It scans table `A` using a tuple-at-a-time loop (line 4), allocates memory from the query's assigned hash table (line 5), and copies all the input tuples' attributes unmodified into the allocated space (line 6 onward). The tuples are stored row-wise in main memory to maximize cache locality.

Next, a second traversal of the pipeline's operators generates the body of the analysis function, `analyze()` (lines 9–14). All analysis functions accept a vector of pointers to materialized tuples and a statistics data structure. The former is a read-only batch of input rows to be analyzed, and the latter is where the results of the analysis are persisted on completion. With PCG, pipeline operators can generate arbitrary analysis code. We require metric values to be "combinable" through a commutative and associative reduction operation. This constraint exists because the DBMS may invoke an analysis function multiple times, but on disjoint chunks of materialized state. Thus, the DBMS needs to merge partial metrics to build an overall view of the state.

There is a balance between the degree of statistical detail collected during analysis and the runtime overhead. Analysis functions must be fast to execute so as not to negate any potential benefits it affords. In our example, the DBMS dispatches to pre-compiled vectorized builtins to collect the min value (line 10), the max value (line 11), and `NULL` count (line 12) for each attribute in the batch, storing the results into the output statistics parameter (the second argument). It is also possible

to generate fused tuple-at-a-time logic to collect these metrics. PCG is agnostic to the chosen approach; it only requires the analysis function to abide by API. As we showed earlier in Section 5.1 and will show again in Section 5.4, the DBMS can collect these metrics with minimal overhead.

Execution: The DBMS first executes the pipeline function, which constructs the hash table in Figure 5.3b. At this point, the hash table becomes immutable. The DBMS divides the hash table into fixed-size partitions and invokes the analysis function on each partition in parallel. Partition-local statistics are then collected and combined to form a global view of the hash table's data, shown in the figure's bottom right side. This information is persisted in an in-memory data structure accessible to all operators further in the query plan.

The DBMS takes special care when executing analysis functions. Ideally, the DBMS should skip the analysis step if it would not impact the code structure of later pipelines. We address this by first running the function on a sample of tuples. For this step, the DBMS selects $k \log n$ tuples at random, where n is the total number of tuples, and k is a configurable oversampling factor. The DBMS invokes the analysis function on the sample, and only if this smaller analysis shows promise does it proceed with a full analysis.

5.3.2 Compressing State

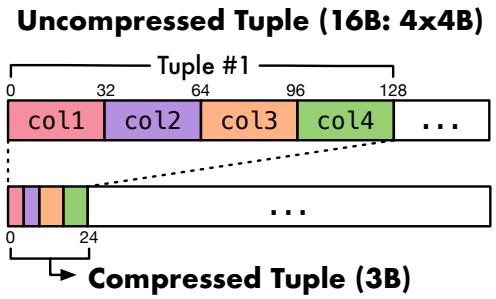
Many relational operators rely on in-memory data structures in their implementation. Hash joins and hash aggregations use hash tables. Order-by operators use large sequential buffers to store tuple data before sorting them. In complex OLAP workloads, these structures are often large and easily exceed even the newest CPUs' cache capacity. Moreover, some query plans require these intermediate structures to exist for their entire lifetime, contributing to peak memory usage. Therefore, it is desirable to minimize each operator's memory footprint to maximize the likelihood that requisite data fits in the CPU's cache and making queries better "citizens" by using resources more frugally.

We now describe an approach to reduce the memory space of ephemeral data structures produced during query processing (e.g., hash tables or buffers) by applying a lightweight compression. It is agnostic to the encoding of the underlying data, and relies solely on runtime analysis. The technique is applicable to any relational operator with only minor differences per operator on when the DBMS applies the compression. For instance, the DBMS compresses data for order-by operations *before* sorting the tuples, which requires the tuple comparison function to operate on compressed tuples. We use the same example query and resulting analysis in Figure 5.3, but now describe how PCG uses the discovered statistics to compress the hash table and alter the probe side of the hash join.

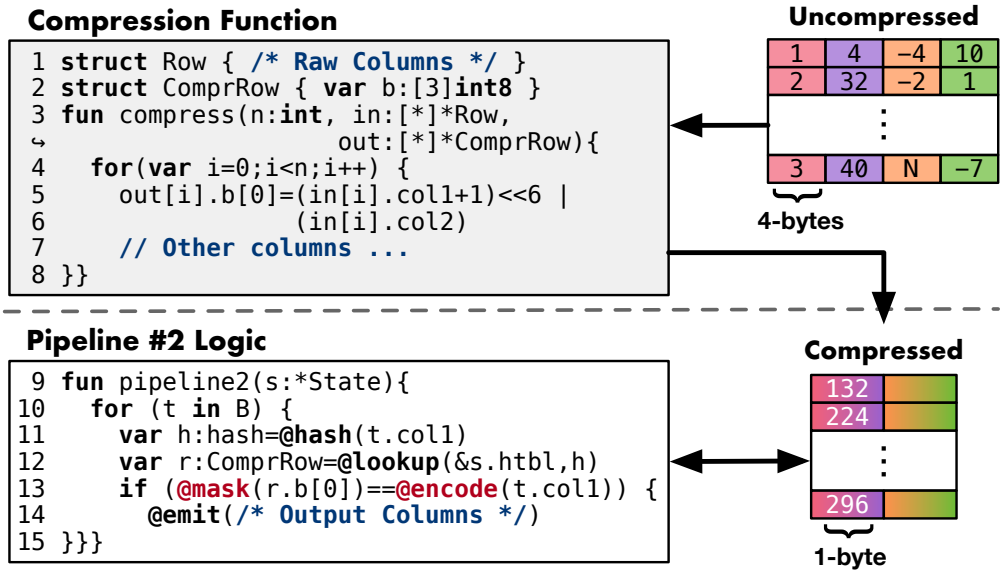
Analysis: PCG supports any compression technique, but in this work, the DBMS applies frame-of-reference (FOR) encoding on tuple attributes [59]. FOR encoding requires knowing only the minimum and maximum (i.e., the range) of all input values. For instance, if the range of values for an attribute is $R = [v_{min}, v_{max}]$, a FOR encoding requires $B = \lceil log_2(v_{max} - v_{min}) \rceil$ bits to represent any value $v \in R$. Operators obtain this information by leveraging analysis functions (see

Column	FOR Value	# Bits
col1	-1	2
col2	0	6
col3	-101	8
col4	-20	8

(a) Results of FOR Analysis



(b) Organization of Compressed Tuple



(c) Generated Code With Execution of Compression and Probe

Figure 5.4: Compressing State – The hash join operator uses the results of the analysis in (a) to generate the compression function and adjust the probing logic to account for the packed layout in (c).

Section 5.3.1). The DBMS then encodes each attribute separately and greedily bit-packs all attributes into as few machine words as possible to minimize space.

The achievable compression depends on the range of values in each attribute. Although techniques exist to optimize FOR compression, they often incur overheads that preclude their use during query processing [19, 44, 161]. Analysis adds overhead because it complicates the compression process and subsequent data access methods. To strike a balance between compression ratio and performance, PCG supports compressing a subset of a tuple's attributes.

The results of the analysis in Figure 5.4a indicate that the DBMS can shrink all four columns' values from their original 16-byte size into just three bytes, 5× reduction; `col1` and `col2` require $\lceil log_2 4 \rceil$=2 and $\lceil log_2 44 \rceil$=6 bits, respectively. Likewise, `col3` and `col4` require one byte each. The resulting bit-packed tuple layout is depicted in Figure 5.4b. Although PCG uses its analysis to collect statistics that guide compression, the DBMS may still decide to not compress the data. Since our

example demonstrates a high compression factor, the hash join registers a pre-pipeline compression function during the second pipeline's generation.

Code Generation: Figure 5.4c shows the code for the second probe pipeline. First, the DBMS generates the compression function on lines 3–8. It relies on two data structures (`Row`, `ComprRow`) that represent the original and compressed tuple layouts, respectively. The function iterates over each tuple (line 4) and writes a bit-packed version to the output tuple (line 5–6). In the example, the DBMS blends the first two columns, placing `col1` in the higher two bits of the first byte, and `col2` in the lower six bits. It also uses the min value in each attribute's domain range as the zero value. Although our example generates tuple-at-a-time code, PCG allows operators to invoke any code including dispatching into pre-compiled SIMD-optimized vectorized primitives, or mixing both techniques. The input batches are 1–4k in size to ensure cache-resident processing.

Next, the DBMS generates the core logic for the second pipeline on lines 9–15. It begins by scanning table `B` using a tuple-at-a-time loop (line 10) and then hashing the join key column, `col1`, on line 11. The DBMS then probes the join hash table on line 12 using the computed hash value, which returns a pointer to a compressed row. This is the first instance where the existence of compression has altered the logic in the pipeline. Next, the DBMS encodes the probe key to validate equality to the build key on line 13 to resolve hash collisions. The DBMS masks off a portion of the retrieved build key because it contains non-key data. We compress the probe key to minimize the number of accesses cache lines. This is useful when dealing with multi-component keys that may span cache lines.

Execution: Recall from Section 5.2.3 that the DBMS executes pre-pipeline functions before the core pipeline logic. Also, recall from Section 5.3.1 that any intermediate state is immutable after the pipeline that produced it finishes running. To perform the compression, the hash join operator allocates a new perfectly sized hash table to store the compressed tuples. It then divides the original and new hash table into disjoint partitions and invokes the compression function on each in parallel. In this way, the DBMS executes custom compression code using multiple execution threads. When compression completes, the DBMS runs the pipeline function to evaluate the probe side of the join. All probe tuples that find one or more join matches are emitted. The existence of compression has required the key equality logic to change slightly; the DBMS must mask off non-key bits, and encodes the probe key into the domain of the (potentially) FOR-compressed build data.

5.3.3 Eliding Overflow Checks

We next present an optimization to skip arithmetic overflow handling normally required when dealing with SQL math. To aid in our discussion, we use the following query:

```
SELECT B.col2, COUNT(*), SUM(A.col2)
  FROM A INNER JOIN B ON A.col1 = B.col1
GROUP BY B.col2
```

We first describe how the DBMS identifies if it can apply the optimization, and then its code generation and runtime behavior.

Analysis: If an input query contains an aggregation, the DBMS installs analysis functions at points in the query plan that materialize any attribute the aggregation requires. Such periodic analysis enables the DBMS to build a more accurate and finer-grained view of the data a query is processing compared to what the DBMS optimizer has at optimization time. To simplify our discussion, we use the statistics in the example in Figure 5.3b. Recall that the schema for tables `A` and `B` declare all columns as 64-bit signed integers. The results from the analysis indicate that the range of values for `A.col2` is $[0, 44]$, which is representable using fewer than 64 bits. The DBMS also infers that the max value of the summation aggregate (i.e., **`SUM`** (`A.col2`)) is $n_B \times 44$ when the cardinality is one and where table `B` has n_B tuples. Thus, it requires an integer data type with at least $\lceil log_2(n_B \times 44) \rceil$ bits, and safely elides generating any overflow handling logic. Similarly, the max value of the count-star aggregate is n_B, requiring at least $\lceil log_2 n_B \rceil$ bits.

The analysis above assumed a foreign-key join, a common occurrence in warehouse workloads. If the query involves a generic join, then the DBMS employs more conservative analysis. In this case, the worst-case max value of the summation aggregate is $n_{HT} \times n_B \times 44$, where n_{HT} is the number of tuples (size) in the hash table. The revised max value may require a larger underlying data type for the aggregate. This information is still available to the DBMS before it generates the aggregation logic, enabling it to make a strictly more accurate decision compared to a one-shot code generation DBMS.

Code Generation: A conventional query compiler will blindly implement the behavior dictated by the operator chosen by the DBMS optimizer. Focusing on the aggregation in the second pipeline, the code generator would assemble code that resembles:

```
1 // The structure holding one group's aggregate information.
2 // Both elements must be 128-bit integers.
3 struct Aggregates { var count, sum_col2 : i128 }
4
5 fun pipeline2(state: *State) {
6   var ht: *AggregationHashTable = &state.ht      // 'ht' stores the aggregates.
7   for (t in B) {                                 // The table scan.
8     var agg: Aggregates = @lookup(ht, t.col2)    // Initial lookup.
9     if (agg == nil) agg = @allocate(ht, t.col2)  // Did we find a group?
10    if (@addOverflow(agg.count, 1, &agg.count) or
11        @addOverflow(agg.sum_col2, t.col2, &agg.sum_col2))
12      @overflowError();
13 } }
```

`Aggregates` is the structure capturing information for one group. It relies on two 128-bit integer types for the total count and summation aggregates. The core pipeline logic, `pipeline2()` listed on lines 5–13, accepts a `state` argument (not shown for brevity) containing all state required for the entire query. The pipeline begins with a sequential scan of table `B` on lines 7–13, then a standard grouping aggregation on lines 8-12. Our interest here are the "checked" additions on lines 10–12 that ensure the DBMS recognizes if an overflow occurs and throws an error.

In contrast, based on the results of the analysis, a PCG-enabled DBMS can instead use a $B = \lceil log_2(7.5 \times 10^6 \times 44) \rceil = 32$-bit integer values for the count and summation aggregates and skip the overflow handling logic altogether:

```
1 // The structure holding one group's aggregate information.
2 struct Aggregates { var count, sum_col2 : i32 }
3 fun pipeline2(state: *State) {
4   var ht: *AggregationHashTable = &state.ht      // 'ht' stores the aggregates.
5   for (t in B) {                                 // The table scan.
6     var agg: Aggregates = @lookup(ht, t.col2)    // Initial lookup.
7     if (agg == nil) agg = @allocate(ht, t.col2)  // Did we find a group?
8     aggr.count++                                 // Simple native addition.
9     aggr.sum_col2 += t.col2
10 } }
```

Thus, the generated aggregation on lines 8–9 use a smaller 32-bit integer without any explicit overflow guard. The optimization is guaranteed to be safe based on the statistics gleaned during the analysis. The resulting code is (1) quicker to compile because it contains fewer instructions, (2) faster to execute, and (3) requires 4× less storage space for all aggregates by using small data types.

5.3.4 Eliding NULL Checks

Existing DBMSs handle `NULL` values in different ways. One approach is to dedicate a special value for it from the domain of the type it represents. For example, the `NULL` value for a `SMALLINT` (i.e., 16-bit signed integer) attribute is the max value in that domain ($2^{15} - 1$). Another approach is to attach each SQL value with a boolean (bit) flag indicating whether it is `NULL`. NoisePage uses the NULL-indication flag method for tuple-at-a-time processing.

When using NULL-indicators, SQL operations must explicitly check if their inputs are `NULL`. For instance, to perform the addition of two NULL-able `INTEGER` (i.e., 32-bit signed integers) values, a and b, a conventional query compiler would generate:

```
1 var a, b, ret: i32 = ...                  // Raw values.
2 var a_null, b_null, ret_null: bool = ... // NULL flags.
3 if (a_null or b_null) {
4   ret_null = true     // Skip addition & set flag.
5 } else {
6   ret_null = false    // Set flag.
7   ret = a + b         // Native addition.
8 }
```

The "simple" addition is guarded by a conditional branch to ensure neither input is `NULL` on line 3, and performed on line 4 otherwise. PCG enables the DBMS to elide these checks if the data the query reads does not have `NULL` values despite the schema declaring them otherwise. It does this in a manner similar to overflow checking in the previous section: it relies on strategically placing analysis functions at materialization points in the query plan. During analysis, in addition to other

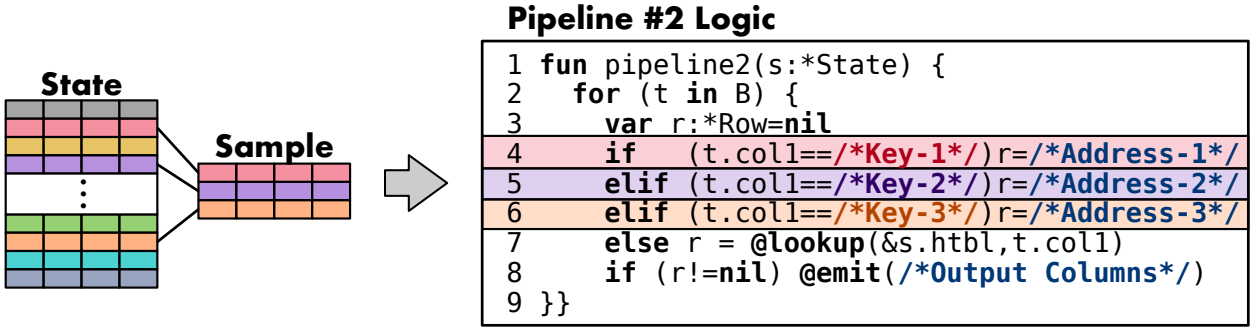


Figure 5.5: Specializing Join Probing Logic – If the DBMS detects join key skew, it samples the input to collect a representative set of "hot" keys, probes the hash table once before generating the probe logic, then generates explicit checks for each key, embedding the address of the result if successful.

metrics collected for materialized attributes, the DBMS also tracks `NULL` counts with the goal of detecting when these checks can be safely removed. If analysis concludes that `NULL` values do not exist in the input, the generation of subsequent pipelines that operate on these attributes can skip `NULL` handling code (i.e., lines 3-4 in the previous example) to save on compilation time and improve query performance.

5.3.5 Value Specialization

Lastly, we discuss an adaptive technique that exploits data skew to optimize hash join performance. Using the example SQL query in Figure 5.3a, we describe how the collected statistics are used to customize the probing logic in the second pipeline.

Analysis: The aim of this optimization is to recognize skew in join keys, extract the N "hottest" keys (where N is configurable), and inline logic for them in the generated code for the probe side of the join. The intuition behind this approach is that skew on the build side translates to skew on the probe side. As this assumption is not always valid, the DBMS selectively enables this optimization based on the joined tables. In this work, we enable it for anti- or semi-joins because the overhead is negligible in these cases.

NoisePage identifies skew using either a histogram- or approximation-based approach. In the former case, the join operator generates analysis code constructing a histogram of the join keys. In the latter case, it uses a HyperLogLog (HLL) estimator to acquire an approximation of the number of distinct keys [53]. Then, if n_{HT} is the total number of tuples in the hash table, and n_D is the (approximate) count of distinct keys, the DBMS applies the optimization if $\frac{n_{HT}}{n_D} > T$, where T is a configurable parameter.

If the DBMS chooses to apply the optimization, the next step is to select the set of N keys, $K = \{k_1, k_2, \ldots, k_N\}$, to extract and inline into the next pipeline's code. If a histogram is used, the first N keys are selected. If an approximate count is used, the DBMS samples N random keys without repetition from the input.

Code Generation: Figure 5.5 shows an example where the DBMS samples three build keys and modifies the generated probing code. The second pipeline's code is contained in the function `pipeline2()` (lines 1–10). It begins with a sequential tuple-at-a-time scan of table `B` (line 2) and contains the modified probing logic. On lines 4–6, the hash join operator explicitly generates a comparison of the current probe key against each key from the set K. If the hash table was previously compressed, the check is modified to encode the probe key and mask the build key appropriately. If any of the specialized key comparisons are successful, the matching tuple's address is written to `row` which stores the current tuple's join partner. The hash join operator computes these memory addresses by performing the lookup of all keys in K *before* generating code. Thus, for the "hot" set of keys, the probe is only performed once at query compilation time rather than query runtime. Finally, a generic hash table lookup is generated on line 7 to handle the fallthrough case, and a tuple is output on line 8 if the probe is valid.

5.4 Evaluation

To evaluate our method, we implemented our PCG framework and execution engine in the NoisePage DBMS [9]. NoisePage is a PostgreSQL-compatible HTAP DBMS that uses HyPer-style MVCC [110] over the Apache Arrow in-memory columnar data [93]. It uses LLVM (v11) to JIT compile our bytecode into machine code.

Our benchmark machine contains two 20-core Intel Xeon Gold 5218R CPUs clocked at 2.1GHz and 192 GB of DRAM. The CPU is 2× hyper-threaded, supports AVX512 SIMD, and contains ten line-fill buffer (LFB) slots supporting at most one outstanding memory prefetch request. For all experiments, we ensure that the DBMS loads the entire database into the same NUMA region using `numactl`. We implemented our microbenchmarks using the Google Benchmark [5] library which runs each experiment a sufficient number of iterations to acquire statistically stable execution times.

We begin with a description of the workloads we use in our evaluation. We then evaluate PCG's ability to improve query performance in comparison to conventional one-shot query compilers.

Microbenchmark: We created a synthetic benchmark to isolate and measure aspects of the DBMS's runtime behavior. The database contains two tables (`A` and `B`) that contain 16×64-bit signed integer columns (`col1`–`col16`). Each table contains 16m tuples and occupies ∼3.8 GB of memory. Each experiment that uses this benchmark states how it varies the distributions and correlations of the database's columns' values to highlight a specific component.

TPC-H: This is a popular decision support system (DSS) workload that simulates an OLAP environment [146]. It contains eight tables in 3NF schema. We use a scale factor of 10 (∼10 GB).

5.4.1 State Compression

We begin with evaluating PCG's ability to compress intermediate state. We explore compression in the context of hash joins and order-by queries. For each case, we isolate and investigate a different aspect of compression (e.g., tuple size, compression ratio etc.) by adjusting the workload query we use and the data we generate.

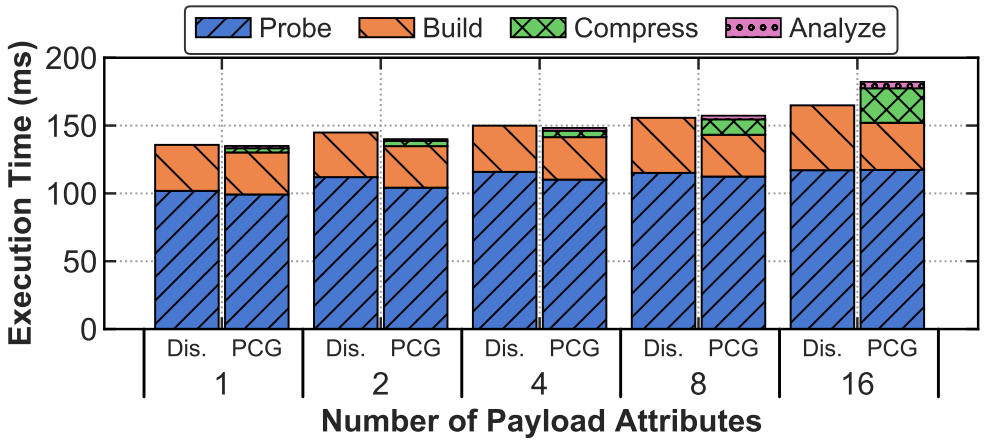


Figure 5.6: Compression with Varying Tuple Size – A breakdown of the time to perform a hash join with and without PCG compression when varying the number of non-key attributes in the build-side tuple.

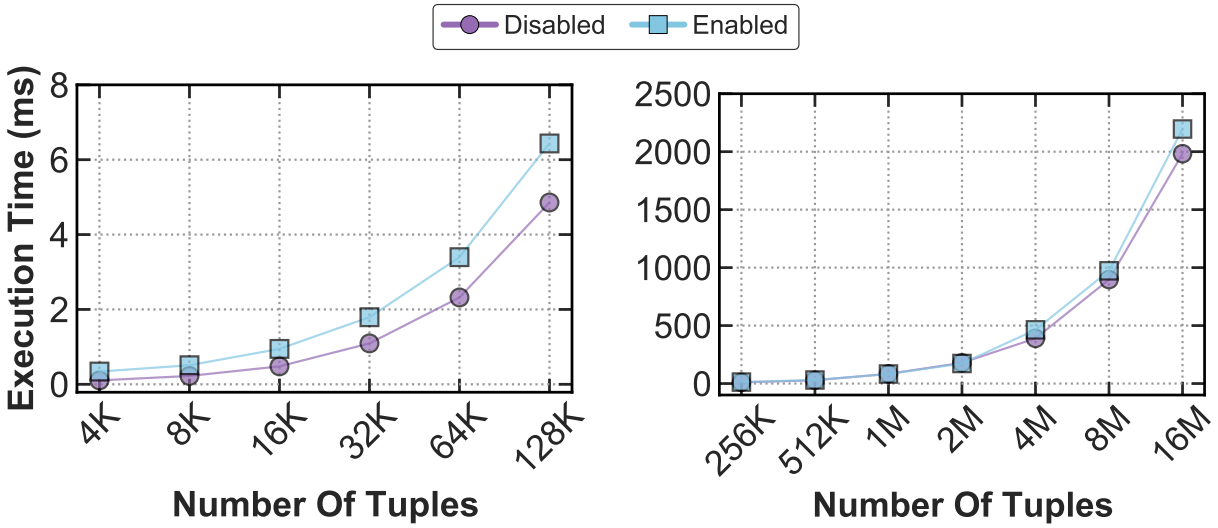
Performance with Varying Payload Size: The first experiment evaluates the performance of NoisePage's hash join as we vary the payload size (i.e., non-key attributes) of the tuples it stores in the join's hash table. The workload consists of a single query:

```
SELECT A.col1, A.col2, ..., A.colk
  FROM A INNER JOIN B ON A.col1 = B.col1
```

The query plan arranges tables `A` and `B` on the build- and probe-side of the join, respectively. Since there is no filter, the DBMS materializes all tuples from `A` in the hash table. We populate the columns `A.col1` and `B.col1` to simulate a foreign-key join with 100% join selectivity; every tuple in table `A` joins with exactly one tuple in `B`, but the keys have a random ordering in both tables. We also generate the data for all materialized attributes from table `A` to achieve a compression factor of 8×. We investigate the effect of compression ratio in a later experiment.

We execute this query with and without PCG's compression, and vary the number of payload attributes from 1–16 to reflect realistic data warehouse queries. Disabling PCG represents how existing JIT compilation-based DBMSs execute this query without compression.

The results in Figure 5.6 show that when the size of the build-side tuple is small, PCG offers a small 1–2% performance improvement over the non-PCG version, but consumes 8× less overall memory because it compresses the hash table. The time that the DBMS spends in both the analysis and compression phases increases with the size of the tuple. This is because wider tuples require collecting more statistics (for analysis) and copying more data (for compression). However, the phases scale at different rates; the overhead of analysis expands by 3× from one to 16 attributes, contributing at most a 3% overhead. Compression time increases by almost 25× and accounts for ∼14% of the overall time when the tuple contains 16 non-key attributes. For wide tuples, PCG compression is feasible to reduce intermediate data sizes with a modest overhead. We leave this decision to the DBMS based on runtime operating conditions.



(a) Low Tuple Count (b) High Tuple Count

Figure 5.7: Compression with Increasing Tuple Counts – Measuring hash join performance with PCG compression when varying the size of the materialized hash table but with a fixed compression factor and tuple size.

Performance with Varying Table Sizes: We next analyze the performance of of NoisePage's hash joins as we vary the size of the intermediate state. We evaluate the following workload query:

```
SELECT A.col1, A.col2, A.col3, B.col1
  FROM A JOIN B ON A.col1 = B.col1
 WHERE A.col1 = δ
```

As before, table A is on the build side of the join and the DBMS populates all columns but only materialize three columns from table A. We adjust the value of the filtering term, δ, to achieve the desired selectivity to control the size of the hash table materialized in the join. Thus, this experiment fixes the compression ratio and tuple size, but varies the compressed intermediate state size.

The results in Figure 5.7a show that enabling compression when the hash table contains fewer than 1m tuples adds up to a 2.5× overhead. In this range, the constructed hash table fits in the CPU's cache. Thus, compression imposes unnecessary work to analyze, compress, and copy tuples into a new hash table that is also cache-resident. NoisePage performs hash joins in a two-phase manner where it can recognize if this situation arises and skips performing compression altogether (see Chapter 4). But we include the results here for the purpose of exposition.

In Figure 5.7b, we see that when the hash table size exceeds the CPU's cache (i.e., more than 1m tuples), the compression overhead falls to 10% over the baseline. This is because the DBMS's execution time is dominated by cache misses while probing the hash table. Compressing the hash table may not guarantee that it fits in CPU cache, but it increases the percentage of tuples that do fit. In this experiment, compression fits 8× more tuples, but still falls out of cache when there are more than 2m tuples. In addition to requiring less memory, compression is beneficial in the presence of

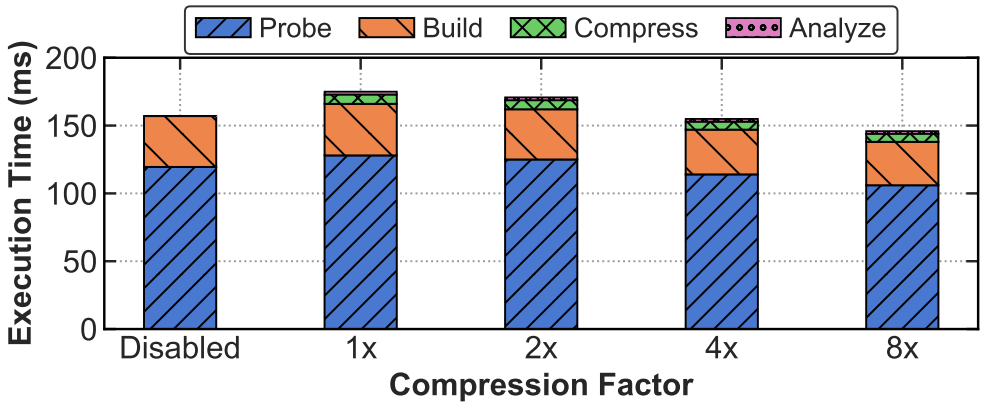


Figure 5.8: Compression with Varying Compression Factors – Measuring hash join performance with PCG compression when varying compression factor but with fixed tuple and hash table sizes.

skew since it increases the likelihood that the most frequently accessed (compressed) tuples fit in cache.

Impact of Compression Factor: We next examine the impact of the compression ratio on hash join performance. We perform a join on tables `A` and `B` as before, but we (1) fix the schema of the build tuple to one key and two payload attributes, and (2) fix the size of the materialized hash table to 2m tuples. We populate all key and payload attributes in table `A` to achieve a specific compression ratio, and measure the join with and without PCG compression. Although NoisePage skips compression if its analysis concludes it to be unnecessary, we include the results here for completeness.

Figure 5.8 shows that PCG compression adds up to a 10% overhead over the baseline to achieve a 2× memory reduction. This is because although the hash table occupies less memory after compression, its total size still exceeds the CPU's cache capacity. Thus, probing the hash table still incurs costly cache misses that dominates overall execution time. We use randomly generated keys (rather than skewed keys) to highlight this effect. However, as the compression ratio increases, the overhead of performing compression is outweighed by the improvement in join's probe phase. At an 8× ratio, the compressed hash table is 48 MB and fits in the CPU cache; this improves the probe phase by ∼20%. In this experiment, the DBMS implemented the probe pipeline using a fused tuple-at-a-time approach. The DBMS could optimize the probe phase by recognizing the cache residency of the hash table in the first pipeline to generate a hybrid vectorized approach with pre-fetching [101].

We also observe that the time that the DBMS spends for compression and analysis is unaffected by this ratio. This is because the total data set size is constant across each configuration. The DBMS collects the same statistics during its analysis for all three attributes in the hash table, and executes the same number of instructions during compression. The analysis phase contributes ∼1% of the runtime and uses vectorized primitives. In contrast, compression is a consistent ∼5% overhead and uses fused loops.

Multi-Component Join Keys: Recall from Section 5.3.2 that when probing a compressed hash table, the DBMS encodes the probe key(s) first. We now investigate the impact of this design choice

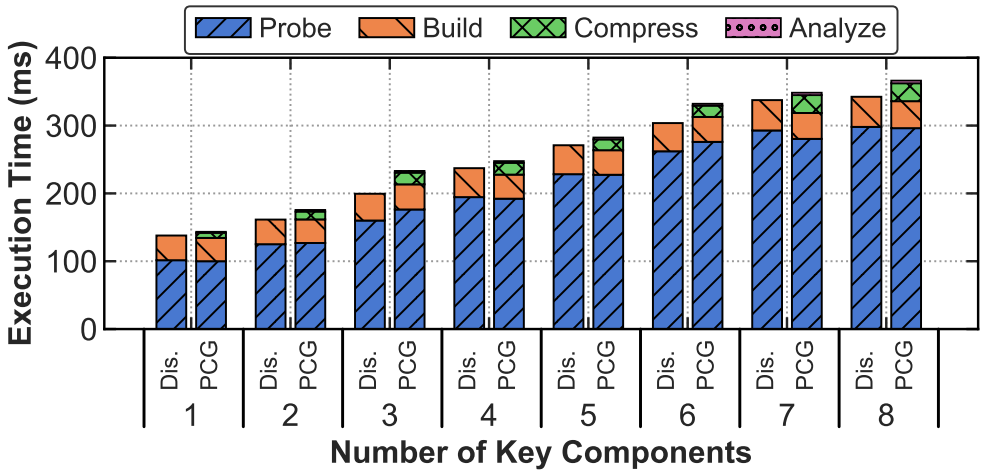


Figure 5.9: Multi-Component Join Keys – Breakdown of time to perform a hash join with a varying number of keys components. The analysis phase is plotted, but not visible.

using the same join query as earlier, but vary the number of join keys between 1–8. We populate the columns `col1`–`col8` in tables `A` and `B` to simulate a 100% selective foreign-key join. We also control data generation for table `A` to achieve an 8× compression ratio, reducing key size from 64-bits to 8-bits. The DBMS materializes all tuples in `A` into the join's hash table.

The results in Figure 5.9 show that PCG adds a 6–8% overhead over the baseline despite reducing total space by 8×. Across all configurations, the build and probe times of both versions are within 1% of each other. Thus, encoding probe keys using the FOR compression scheme adds only a modest overhead. Both versions also issue the same number of random memory accesses when performing the initial lookup into the hash table. In the baseline, the uncompressed tuple has at most eight 8-byte keys, which fills one cache line. In the PCG version, the compressed tuple requires at most only one byte (eight 1-byte keys), but still incurs a full cache-line retrieval due to the machine architecture's granularity of memory access.

We attribute all the observed overhead to the analysis and compression phases. The analysis phase is never more than ∼2% of the total runtime, but it does rise by 2× with increased key complexity. This is because analysis relies on vectorized primitives to reduce instruction counts and maximize cycles-per-instruction. In contrast, the compression phase accounts for 14–16% overhead and increases by 3.8× from one to eight keys because it not only compresses the source data, but also copies that data into a new memory space.

Multi-Component Sort Keys: NoisePage with PCG compression executes order-by queries by first materializing tuple data into a buffer and then analyzes them to determine their suitability for compression. The DBMS then generates a compression-aware tuple comparison function for the sort algorithm. The resulting scan over the sorted tuples processes them in their compressed form. To investigate the impact of compression when sorting data, we evaluate the following workload query that scans table `A`, projects a varying number of columns, and sorts on these columns:

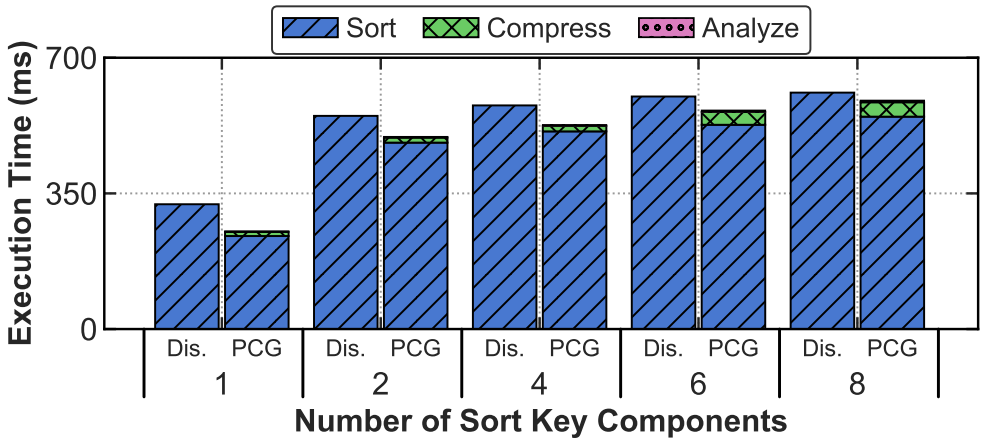


Figure 5.10: Compression During Sort – Breakdown of sorting time when using a multi-component key. The analysis phase is plotted, but not visible.

```
SELECT A.col1, A.col2, ..., A.colk
  FROM A
ORDER BY A.col1, A.col2, ..., A.colk
```

We populate the columns in A with random data that guarantees a 4× compression ratio. We measure the query's execution time and provide a breakdown of its phases.

Figure 5.10 shows that compressing input tuples before sorting yields a ∼30% performance improvement over the baseline for simple keys. However, the gap closes with increasing key complexity. With eight key attributes, PCG offers a modest 4% improvement. In all cases, PCG reduces the order-by operator's memory consumption by 4× and never degrades performance. As in previous experiments, we also observe that analysis adds a constant (negligible) overhead of ∼1%. We attribute this to the use of parallel execution and vectorized primitives. The time taken in the compression phase increases with key complexity growing from 4% to 7% with one to eight attributes, respectively. This is because the DBMS has to read and write more data as the number of attributes in the key grows.

5.4.2 Overflow Checking

We now examine aggregations in NoisePage when employing PCG to learn when to elide arithmetic overflow checking. For this experiment, we use the same query as in Section 5.1, but we vary the total number of computed aggregates. As before, the filtering term is fixed at 95%. The DBMS generates the probe pipeline after analyzing and compressing the join's hash table, and uses a standard fused tuple-at-a-time loop. We include analysis time when the DBMS uses PCG. With PCG, the DBMS uses a smaller 32-bit signed integer type, and elides overflow and null checks in the probe pipeline. The non-PCG versions use 128-bit signed integers for the aggregates.

The results in Figure 5.11 show that PCG-optimized aggregation performs 2.5× over the baselines. Recall that Branch represents how existing one-shot compilation-based DBMSs generate the aggregation logic. Branch is slower because it executes more branching instructions (i.e., the overflow guards) and uses slower 128-bit arithmetic. Flag converts the control dependency from the

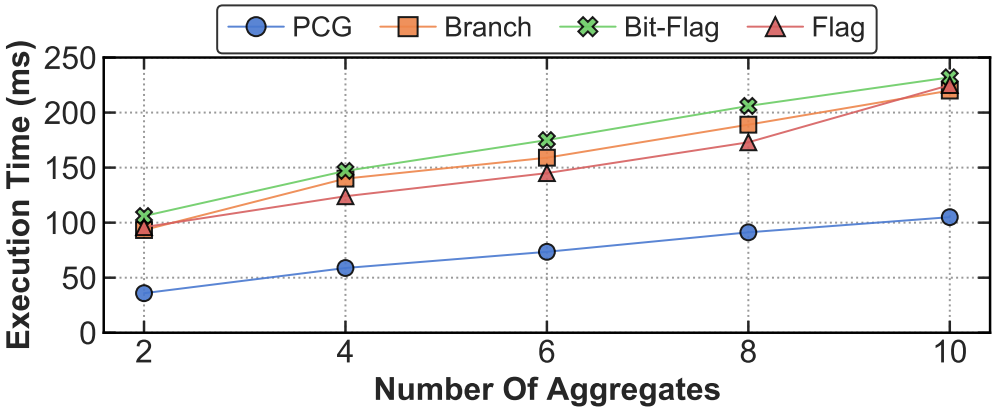


Figure 5.11: Overflow Checking – Performance measurements during aggregation when varying the number of aggregates in the query.

conditional branch into a data dependency, resulting in a ∼13% improvement over Branch. However, it too still requires 128-bit operations. Bit-Flag resembles Flag, but replaces boolean logic data dependencies into bit-wise operations resulting in poorer performance. Bit-Flag is 15–20% slower than the other non-PCG variants because it always executes both sides of the operation regardless if an overflow occurred; thus is cannot benefit from short-circuit evaluation.

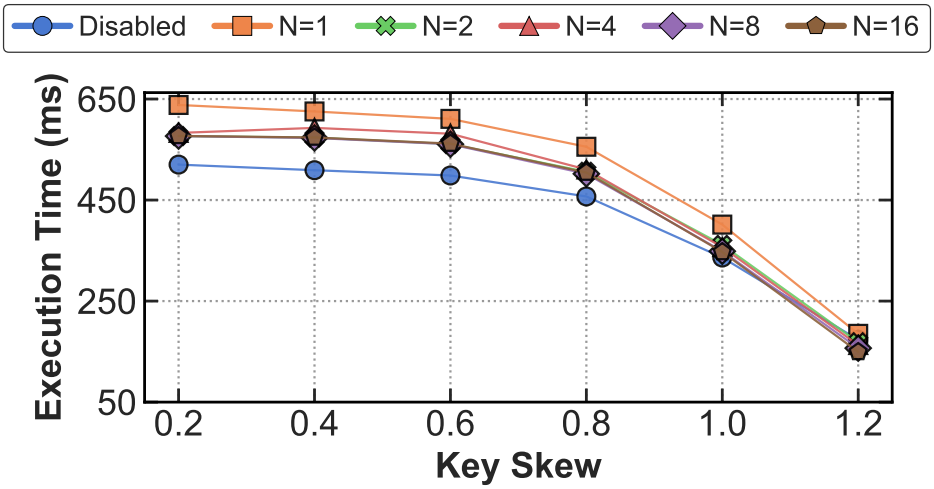
5.4.3 Join Key Specialization

We next evaluate PCG's ability to exploit data skew in hash joins. As described in Section 5.3.5, NoisePage executes hash joins by injecting an analysis phase between the execution of the first (build) pipeline and the generation of the second (probe) pipeline. Analysis enables the DBMS to identify if skew exists and directly embed explicit key checks in the probe pipeline. It does this by performing the lookup at query compilation time once rather than at query runtime.

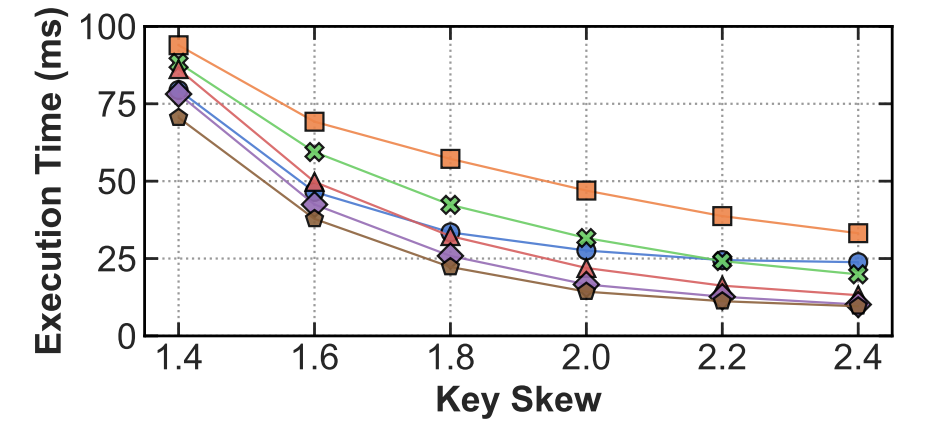
To examine this optimization's impact on join performance, we evaluate the same query as in Figure 5.3a. We populate the joining columns `A.col1` and `B.col1` according to a Zipfian distribution and vary the skew factor. We also vary the number of keys we embed (N) to explore when the optimization is most suitable. A configuration N=n embeds n keys in n branches into the probe pipeline. The baseline is to embed no keys (Disabled); this represents how existing compilation-based DBMSs execute the query. We measure both the execution time and the fraction of hash table probes that hit one of the special cases generated in the PCG versions.

Figure 5.12a shows that all PCG versions perform worse than the baseline when there is low (to no) skew in the join keys. For example, embedding only one join key (N=1) adds 8% overhead. This is because PCG introduces many untaken branches that, although predicted correctly, still execute more instructions than the baseline. Figure 5.12b shows that some of the PCG variants gradually outperform the baseline as skew increases. The performance of the optimized versions plateau as the DBMS approaches the absolute min number of executed instructions to implement the join. N=16 executes half the number of instructions as Disabled, and is more than 2.5× faster.

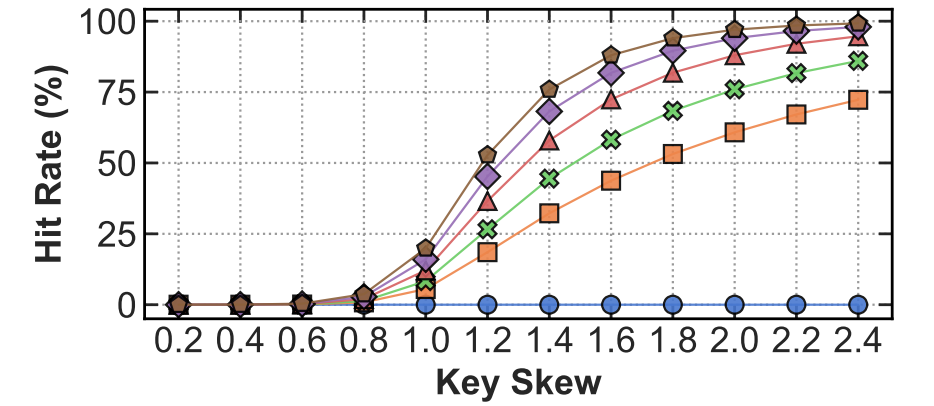
We also observe that not all PCG variants perform better than the baseline at higher skews and those that do occur at different skews. N=1 never betters Disabled, but N=2, N=4, N=8, and N=16 outperform the baseline at skews 2.2, 1.8, 1.6, and 1.4, respectively. These transfer points correspond



(a) Low Skew

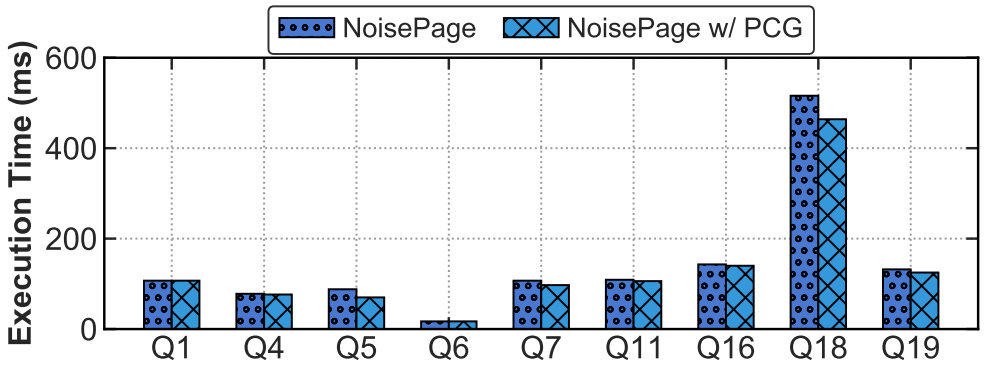


(b) High Skew

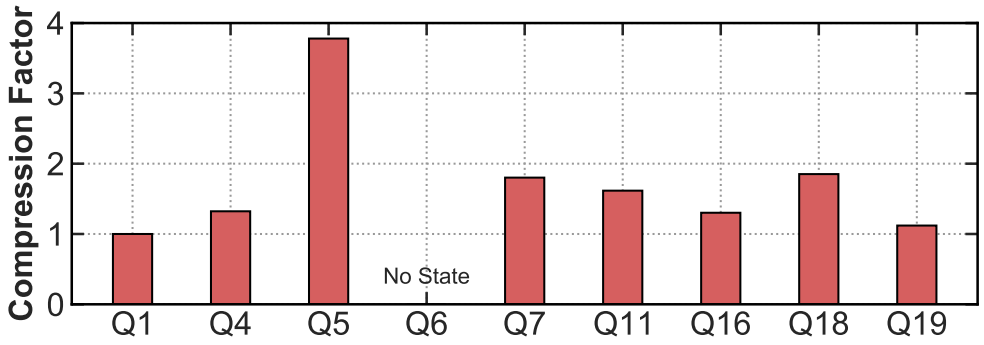


(c) Hit Rate

Figure 5.12: Varying Join Key Skew – Performance of PCG's join key specialization when increasing skew in the join keys. (a) and (b) plot the total execution time when the skew is low and high, respectively; (c) shows the percentage of hash table probes that hit a heavy-hitter branch.



(a) Execution Time



(b) Memory Reduction Ratio

Figure 5.13: TPC-H Performance – Evaluation of a selection of TPC-H queries with and without memory compression enabled using PCG. (a) and (b) show the query performance and memory reduction ratio, respectively.

to points in Figure 5.12c where more than 75% of the hash table probes hit one of the special case branches. When the branches are taken less frequently, the CPU cannot correctly predict their outcome, leading to costly mispredictions. For instance, N=1 is 1.5× slower than Disabled at skew 2.4, but incurs 25× more branch mispredictions. CPU architectures employ different branch prediction techniques, making N machine-dependent. Although the threshold depends on the machine the DBMS is running on, it does not change over its lifetime. The DBMS can derive this bias threshold value on startup using microbenchmarks.

5.4.4 TPC-H

Lastly, we evaluate NoisePage with and without PCG on the TPC-H benchmark. We load a scale factor 10 (∼10 GB) data set and first warm the system caches by running all queries three times. We then evaluate all queries and report the average over five consecutive executions. We also report the reduction in each query's total memory footprint when applying PCG compared to when it is disabled.

The first observation of the results in Figure 5.13 is that the DBMS's performance with PCG is never worse than the baseline. Unlike the previous experiments, we allow the DBMS to bypass compression if it deems it unbeneficial. Since PCG's analysis captures perfect information, the decision

on whether to apply an optimization is always correct. For example, Q1 uses a hash table to store aggregates, but the DBMS does not compress it because it uses four group-by keys and already fits in CPU cache. In contrast, Q5 is ∼25% faster with PCG because it uses smaller data types in its five hash tables. Similarly, Q18 is 10% faster with PCG for the same reason.

Our second observation is that not all queries benefit from compression, but PCG is able to achieve up to a 4× reduction in total memory for some queries. As mentioned, Q1's hash table and sort buffers fit in cache and, thus, do not benefit from compression. As a counter point, all five hash joins' hash tables in Q1 are compressed; but, the aggregation and sorting buffers are untouched because they are cache-resident.

PCG's compression is not meant to always improve performance. Still, we emphasize that it can also reduce memory consumption through its lightweight analysis and custom code generation. In general, we observe that the hash table data structure often consumes the largest amount of memory in a query. Thus, we expect greater benefit from PCG compression with increasing data sizes and query complexity.

5.5 Related Work

Compression: Westmann et al. explore integrating lightweight compression holistically throughout all database components, and evaluate its impact on query performance in a row-store DBMS [153]. The authors show that using lightweight compression methods like FOR [59] or null suppression [130] for integers, and dictionary compression [130] for strings, yields up to a 55% reduction in query response times. Abadi et al. present an evaluation on compression in column-stores DBMSs [13]. The authors integrate five compressed schemes into a DBMS by abstracting access to base and intermediate column data behind an API. This API hides how a column is formatted and instead exposes properties about the underlying data (e.g., sortedness) that compression-unaware operators exploit. However, both these works focus compression only on storing and processing base table data.

Chen et al. are one of the first to compress intermediate results for disk-centric row-stores [39]. They introduce transient decompression wherein operators incapable of processing compressed data temporarily decompress their inputs and retain the original input as well. The DBMS then processes the data, and then recompresses and combines the output with the original compressed input data. Thus, subsequent operators still benefit from compression.

MorphStore [43] presents the design of a column-store query engine that implements continuous compression for both base table data and all intermediates. Built atop MonetDB [30], MorphStore operators consume compressed inputs and produce compressed output in potentially different formats. Operators rely on a library of SIMD-optimized vectorized wrappers to *morph* data in one scheme into different scheme without decompressing. Instead, wrappers deliver small vector register-resident chunks of data to operators. The engine also dynamically selects a suitable compression format for an operator's output based on its characteristics. PCG only compresses data at pipeline boundaries rather than after each operator.

Gubner et al. describe an approach to compress intermediate hash tables using domain-guided prefix suppression [64]. Before execution, the DBMS determines a suitable encoding of the attributes materialized in a hash table by inferring their domain bounds and truncating unused prefix bits. During execution, the DBMS dispatches to pre-compiled vectorized primitives to encode and

decode tuple data when writing to and reading from the hash table. PCG supports any compression technique, determines suitability for compression dynamically at runtime, and relies on generating custom TPL compression code.

5.6 Conclusion

We presented PCG, a framework that interleaves code generation and query execution. With PCG, the DBMS decomposes queries into smaller pipelines compiled and executed independently. In this manner, PCG enables the DBMS to hyper-optimize code to both the input query and the data the query reads but do so safely without the need for a fallback path. It does this by strategically injecting custom utility code between pipelines to observe data and uses this data to specialize how later parts of the query are generated. We present four optimizations built on top of PCG that target performance and memory reduction through compression. Our evaluation shows that PCG incurs a modest (tunable) overhead but can achieve up to a $4\times$ decrease in memory and $2\times$ boost in performance in both synthetic and TPC-H benchmarks.

Chapter 6

Related Work

The techniques developed in this dissertation span several areas of research: (1) query compilation, (2) adaptive query processing, and (3) dynamic compiler optimizations outside the database community. This chapter highlights the projects, systems, and techniques from these areas that are most relevant.

6.1 Query Compilation

A primitive form of code generation was developed for IBM System R in 1970s [34]. System R directly compiled SQL statements into assembly code by selecting pre-defined code templates for each operator. Query compilation is performed for both both OLTP-style queries and ad-hoc queries. The authors later remarked that though compiling repetitive queries had obvious benefits by avoiding the cost of parsing and optimization, the benefits of compiling ad-hoc queries were less clear. Interestingly, they conclude that the generated code was often more efficient than its interpreted counterpart and thus additional latency due to code generation was worth it. Ultimately, IBM abandoned this approach in the early 1980s because of the high cost of external function calls, poor portability across operating systems, and software engineering complications.

Query compilation was not considered in any major DBMS in the 1980s and 1990s (with a few minor exceptions). Instead, it was supplanted by the Volcano query processing model [60]. More recently, query compilation has been used in modern in-memory DBMSs. One of the first systems to revive the technique was Microsoft's Hekaton [55], an in-memory OLTP storage manager for the SQL Server DBMS. Hekaton compiles queries by transforming a conventional query plan into a single C-code function that implements a Volcano-style iterator. This function contains the logic of all the operators, and the data-flow path is woven through the operators using `goto` statements. Since the advent of Hekaton, there have been several systems developed that also use query compilation.

Cloudera's Impala [83] is a distributed OLAP DBMS with a mixed C++/LLVM vectorized execution engine. Impala uses LLVM to compile query-specific versions of frequently executed functions, including functions to parse tuples, compute tuple hashes, and evaluate predicates. These specialized functions take advantage of type information to avoid conditional branches and omit function calls. During query execution, these specialized functions are JIT compiled and replace interpreted versions. Impala also compiles and inlines UDFs into the query's execution plan.

HyPer [77] pioneered the data-centric (push-based) query execution model [108]. HyPer translates a given query plan into LLVM IR, but relies on precompiled C++ code for the more complex query-agnostic database logic. The push-based engine fuses together all operators in a pipeline, obviating the need to materialize data between operators, instead allowing them to access tuple attributes directly in CPU registers. This produces compact loops that improve code locality and overall execution time.

Umbra [109] (the successor to HyPer) is an HTAP DBMS that also uses data-centric code generation. Umbra translates query plans into a custom IR optimized for low-latency query execution rather than using LLVM [79]. The authors note that LLVM is a general-purpose IR and emphasizes common interaction patterns such as instruction reordering, replacement, and deletion. A DBMS query processor does not use these IR features but has to pay the overhead nonetheless. Instead, Umbra's IR is faster to generate and execute, despite resembling LLVM. Umbra either interprets their IR or compiles it directly to x86 assembly.

JAMDB [126] from IBM presented a relational, in-memory, Java-based database prototype that compiled query plans into Java classes that are JIT by the JVM. They implemented compilation of all relational operators and expressions required to support the TPC-H benchmark and compared against an interpreted engine. Interestingly, the JAMDB query compiler structures generated code in a manner that strikingly resembles the data-centric model from [108], but do not provide a transparent model or API for how they achieve this.

SingleStore [8] (previously MemSQL) underwent two incarnations of their code generation query engine. Their first version used a template-driven approach to generate C++ code that was compiled to machine code by forking a GCC process. Due to long compilation latencies, they rewrote the engine to translate query plans into a DSL called the MemSQL Programming Language (MPL), which is compiled into a custom bytecode. The DBMS either interprets this bytecode or compiles it into native code using LLVM. Both versions of the engine perform one-shot code generation without any adaptive techniques. Query parameters are stripped out from queries to avoid recompilation when the query runs with new input values.

LegoBase [80] uses generative-programming (i.e., staging) to partially evaluate a Volcano-style interpretation engine and produce highly customized C query code. Optimizations to convert to push-based dataflow, row or columnar formats, or vectorized or tuple-at-a-time processing are applied during this transformation. DBLAB/LB [134] is the spiritual successor to LegoBase and also uses the staging based technique to generate query code. Unlike LegoBase, DBLAB/LB relies on a stack of DSLs to incrementally lower an interpreted query engine into C code. Each DSL functions at a different level of abstraction and is optimized independent of each other.

DBToaster [17, 18] is a stream processing engine designed for efficient view maintenance. In existing DBMSs, incremental updates to materialized views are treated like an update to a regular table. This simplifies the implementation by reusing existing machinery, but it does not consider the nature of the view and its relationship to the base tables. DBToaster instead analyzes these relationships to construct an optimized delta query that often obviates the need for subsequent scans of base tables. It then translates this delta query into C++ code and compiles it into machine code using a standard compiler. In this way, the DBToaster system is able to offer orders of magnitude performance improvements for long-standing queries.

Tupleware [41] is a distributed DBMS that automatically compiles workflows composed of UDFs into LLVM IR. Workflows are introspected to find vectorizable and non-vectorizable portions that drives the code generation process. Tupleware also presents a new hybrid predicate evaluation technique that separates predicate checking with output copying using a heuristic model.

In [137] and [136], the authors compared the performance of a vectorized and compiled query engine across three different simple query types: projections, selections and hash joins. They conclude that neither technique is always optimal, but that a combination of the two techniques is required to achieve the best performance. Specifically, projections are best performed by a compiled query, while selections are best performed by a vectorized query. Hash joins are significantly more complicated and require a custom combination of the two techniques.

The HIQUE [84] system uses query compilation without the Volcano iterator model. Instead, HIQUE translates the algebraic query plan into C++ using code templates for each operator. These templates form the structure of the operator, but low-level record access methods and predicate evaluation logic is customized per query. Unlike our ROF model, each operator in HIQUE always materializes its results, which prevents operator pipelining.

OmniSci [144] (previously MapD) is a GPU-accelerated DBMS designed to handle read-only queries. It implements a mixed C++/LLVM execution engine. All query-specific routines and predicate expressions are compiled into LLVM IR, then JIT compiled into native GPU code through Nvidia's intermediary NVVM IR. Like MemSQL, extracts constants to avoid recompilation when a query is re-executed with different parameters. It does this by using the generated IR as a key into a hash-table that maps IR to JITed query code.

Voodoo [120] presents an intermediate vector-based algebra that abstracts away the specifics of a system's underlying hardware such as cache configurations, NUMA layouts, or availability of SIMD instructions. A DBMS translates a physical plan into Voodoo and relies on a back-end that compiles it into portable OpenCL [10] code, which can be deployed in heterogeneous hardware environments that include both CPUs and GPUs.

The Tungsten [21] engine in Apache Spark introduced data-centric code generation for expressions. The DBMS converts a query's `WHERE` clause into an AST that it then compiles into JVM bytecode. The JVM internally decides whether this code is interpreted or compiled to native code. Using JVM bytecode simplifies interoperating with generated code since Spark is written in Scala (i.e., a JVM language). Like existing systems, Tungsten performs one-shot code generation (referred to as whole-stage generation in their work) and does not attempt to adapt this code at runtime. Code generation is also disabled for large queries due to platform and JVM limitations.

LB2 [145] presents an exciting approach to implementing a code generation query engine. With LB2, the query interpreter is written in the Scala programming language. The interpreter uses the data-centric model by relying on callback functions. Using callback functions enables the DBMS to benefit from data-centric efficiency and the Volcano-style iterator's well-understood structure. The authors use staged compilation [80, 134] and Futamura projections to derive a query compiler from an interpreter. The resulting compiler generates C code which is further compiled into native code using GCC. However, LB2 suffers from very long compilation times, making it infeasible for use in real-time analytic workloads.

6.2 Adaptive Query Processing

Deshpande et al. provides a thorough survey of the AQP methods up to the late 2000s [46]. 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 based on the data that it observed in the first run 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 [40, 61] and proactive reoptimization [24]. The parametric optimization method for the Volcano optimizer generates multiple plans for a pipeline and embeds them in the plan. The optimizer then inserts 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 [24]. Rio also supports collecting statistics during query execution. Plan Bouquets [51] 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 [118]. Their work advocates for query re-optimization as a cost-effective solution. 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 [92]. 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 [148]. 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 [82]. 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 Section 4.2.

Another in-memory DBMS that supports adaptivity is Vector [29]. 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 [124]. 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 [158]. Their framework estimates the selectivities of multi-table queries to adapt execution orderings. Our PCQ framework does not rely on low-level counters and supports additional optimizations beyond filtering, including joins and aggregations.

A more recent adaptive approach for JIT compiled systems was proposed for Apache Spark [132]. This method provides dynamic speculative optimizations for compiling data file parsing logic. Grizzly [62] 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 [68]. 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 [138]. 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 [50]. Their work groups the canonical choke-point 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.

6.3 Adaptive Compiler Techniques

Optimizing generated code based on runtime conditions is well-studied the compiler literature, specifically in the context of high-performance dynamic language virtual machines [35, 36, 47, 69, 71, 74, 98]. The use of speculative optimization and deoptimization as a technique to enhance performance was first introduced in the Self programming language [70]. At a high level, the compiler first generates a generic version of a code fragment to execute. Depending on the scope of optimization, a code fragment may be restricted to a single loop (e.g., tracing JIT) or an entire method (e.g., method-based JIT). The compiler profiles the generated code and if it speculates that the collected information is stable (i.e., the information is not going to change in future executions), it generates a new version that optimizes for the observed common case. The optimized version also contains a fallback code path to handle inputs that invalidate speculative assumptions. When this occurs, the code is deoptimized by transferring control back to the runtime and reverting to the generic version. PCG exploits the semantics of query processing to safely remove the need for a fallback and its associated overhead.

The inspector/executor paradigm is used by compilers to generate parallel code for sparse matrix multiplication applications. In this model, the compiler generates *inspection* code to examine data at runtime first, and then *executor* code that exercises specific optimizations incorporating the

runtime information [25, 103, 128, 150]. These inspector/executor optimizatations have also been employed outside the domain of matrix multiplication, including optimizing parallelization and communication [127, 131], and data reorganization [49, 66, 99, 104, 154]

Chapter 7

Future Work

In this chapter, we propose several future directions for the work presented in this dissertation. We first discuss immediate extensions that build upon our work to further improve query robustness. We conclude with a broader outlook on how modern DBMS can achieve robustness in an ecosystem that includes heterogeneous hardware platforms.

7.1 Inter-Query Optimization

The techniques presented in this dissertation attempt to ensure the DBMS executes a *single query* in the most efficient manner possible. The DBMS collects comprehensive statistics on the data a query is processing dynamically at runtime to achieve this. The DBMS then exploits this data to fine-tune the query's code and temporary data structures. However, despite spending effort acquiring this information, the DBMS discards it entirely after the query finishes execution. The DBMS cannot use the statistics garnered from an earlier execution of a query to optimize future queries. Although our techniques impose negligible overhead, repeating work is still wasteful. It would be desirable if the DBMS's execution engine could reincorporate runtime information back into the DBMS optimizer to *correct* optimizer errors.

Runtime statistics differ from those maintained by the DBMS optimizer in several fundamental ways. Foremost is that runtime statistics are 100% accurate, whereas optimizer statistics are only rough summarizations. Also, runtime statistics can be fine-grained while that available to the optimizer is coarse-grained. Hence, the first step is to investigate how to reconcile this impedance mismatch. The presence of constants and parameters in a query complicates this further as they influence the data distributions observed during execution. One promising approach to address this issue is to augment the DBMS optimizer to store *virtual* catalog entries that treat ephemeral results (e.g., the result of a hash join between two potentially virtual tables). In addition to cardinality information, the DBMS can track domain information, `NULL` information based on the techniques in this work. Interestingly, the DBMS can track arbitrary statistics but must balance the runtime overhead with the expected utility of information it gleans.

7.2 Advanced Adaptive Policies

Building on top of its efficient mechanism for permuting compiled queries without the need for recompilation, PCQ enables a wide set of potential policies and strategies for controlling the dynamic re-optimization process. Although a thorough exploration of the design space for these policies is beyond the scope of this dissertation, we highlight some key considerations.

First, as with any dynamic optimization scheme, one must be careful to balance the runtime overhead of measuring runtime behaviors against the likely performance gains from improved optimization. For example, with a sampling-based approach, one important parameter is the sampling frequency. In addition, while one approach would be to choose a sampling frequency and then perform that sampling uniformly throughout query execution, another approach would be to dynamically adjust the sampling frequency based upon how stable or unstable the dynamic conditions appear to be. To the extent that hardware support can help reduce sampling overhead (e.g., by making use of hardware counters to monitor cache misses, branch mispredicts [62, 158]), this can help provide more flexibility.

Second, when re-optimization does appear to be warranted, another key issue is how to effectively explore the space of potential optimizations. If the set of potential permutations is small (e.g., if there are two filters to order), it may be possible to exhaustively explore all possibilities whenever re-optimization appears to be warranted. But if the set of potential optimizations is large, then heuristics for efficiently exploring that space may be helpful.

7.3 Lightweight Recompilation

Recall from Chapter 5 that PCG achieves query robustness by staggering when query fragments are compiled and executed. Although this approach improves robustness, it restricts the DBMS to consider alternate plans and optimizations only at pipeline boundaries. Fully (or mostly fully) pipelined plans or plans that materialize too late in query execution may never reap the benefits of adaptivity. PCQ goes a long way towards enabling intra-pipeline adaptivity, but it cannot perform complex structural changes to a query plan. What is needed is the ability to recompile a pipeline *mid-execution*. A motivating example is when a stateful operator (e.g., a hash join) needs to spill to the disk. If the optimizer planned incorrectly, the generated code would not be able to handle this transition. The simple solution is to abort the transaction, re-generate the query plan using spill-aware operators, and re-execute the query. However, this results in wasted work.

The majority of our work relies on TPL, a lightweight DSL that is both fast to generate and efficient to execute without requiring complete JIT compilation. An exciting direction is to explore how these characteristics can be utilized to *recompile* pipeline logic during execution with minimal overhead. The idea is to allow a pipeline to observe data at runtime using the PCG and PCQ infrastructure discussed in this dissertation, but signal back to the DBMS to consider reoptimizing the pipeline or a subtree of the query plan. We believe it possible to achieve this since generating TPL is fast (tens of microseconds), and PCG localizes the compilation scope. In this model, operators notify the DBMS runtime (with sufficient lead time) that recompilation is needed (e.g., due to an impending spill). The runtime generates new TPL code modified to incorporate this new information and continues execution in a cautionary mode. Over time, the new code is eventually JIT-compiled to improve performance further.

7.4 Specialization Outside Query Processing

Existing work on compilation within the DBMS has focused predominantly on optimizing query processing performance. However, we see several other areas where we believe compilation lends improved performance. Generating specialized code is best suited when it replaces interpreted logic, thereby reducing CPU instruction count. One interesting use case is DBMS logging. When logging during transaction processing, the DBMS serializes attributes from one or more tuples into memory. Then when deserializing log records during recovery, it *interprets* a table's schema to (1) determine how data in a log record is serialized and (2) how to appropriately write these bytes into the appropriate space in the table's heap memory. This interpretation can potentially be specialized away using generated code to improve performance.

Another exciting idea is to specialize access and traversal of B-tree indexes. Probing a conventional B-tree node involves a binary search in a sorted array of index keys. Binary search does not exploit any properties of the keys stored in the array; it is, by design, a general-purpose algorithm. On the other hand, if the DBMS can analyze the keys in the tree and the access pattern, it is feasible to (1) *bias* the search in a given direction and (2) extract "common" keys out of the search to provide a fast-path for common key accesses.

7.5 Heterogeneous Hardware

Although Moore's Law remains true even at the time of writing, there is growing uncertainty whether it is possible to reduce the size of CPU transistors beyond 2nm [7]. Even if semiconductor manufacturers can improve either their fabrication process (e.g., extreme ultraviolet lithography) or CPU transistor technology (e.g., nanosheet gate-all-around), the total cost of design may make them financially unviable [4]. Instead, hardware vendors are offering alternative specialized computation platforms. For example, Intel SIMD registers continue to increase in width, now reaching 512 bits. Graphics processing units (GPU) have tens of gigabytes of memory, thousands of simple cores, and far higher memory bandwidth than general-purpose CPUs. Finally, field-programmable gate arrays (FPGA) offer an exciting platform blurring the line between hardware and software.

Hardware platforms of the future will be heterogeneous. On query execution, the DBMS must decide which portion of the query to execute on which device, taking into account the cost associated with data movement. Moreover, although the DBMS knows the strengths and weaknesses of each device beforehand, it only obtains clarity on the data a query is processing during execution. Thus, it must dynamically map query computation to the appropriate device, accounting for existing workloads and resource availability. These challenges further stress the need for runtime robustness.

Chapter 8

Concluding Remarks

In this dissertation, we presented several techniques to improve the performance and robustness of compilation-based DBMS query processing engines. Existing DBMSs employ a "compile-then-execute" strategy wherein all code for a query plan is generated first and only then compiled and executed without regard for the plan's quality. Although this works in OLTP workloads characterized by repetitive and short-lived queries, this is not the case in HTAP settings. HTAP workloads are far less predictable (i.e., ad-hoc), more complex because they involve many more tables, and are long-lasting as they typically process entire tables. These properties complicate query planning and optimization. The update rate in HTAP environments exacerbates the problem since the DBMS's statistics are often outdated, misguiding the query optimization process. Although the AQP literature offers some hope, none of the existing techniques work in compilation-based DBMSs. They either impose non-trivial compilation overhead or require aborting a query during execution, wasting resources and time.

This dissertation bridges the gap between compilation-based query processing and AQP in three parts presented in increasing scope of flexibility. In Chapter 3, we presented ROF, a technique that seamlessly blends query compilation and vectorized processing throughout the query engine. We achieve this by introducing staging points into a query plan where intermediate results are temporarily materialized in cache-resident buffers. Operators individually decide where to inject staging points with the assistance of the DBMS optimizer. Staging buffers enable operators to exploit inter-tuple parallelism using a combination of SIMD vectorization and software prefetching.

With ROF, query operators optimize themselves independently of each other. Although this improves performance, the DBMS can reach a local optimum. To address this, in Chapter 4 we presented PCQ, a technique that expands the granularity of adaptability across *multiple* operators within a pipeline. With PCQ, the DBMS structures generated code to utilize dynamic runtime structures behind a layer of indirection. Indirection plays a critical role in enabling the DBMS to switch between query plans during execution safely. PCQ employs the relaxed fusion technique from Chapter 3 to minimize any potential overheads introduced by adding a level of indirection. Using PCQ as a foundation, we implemented three optimizations that span single and multiple operators, including table scans, aggregations, and multi-way joins.

Both ROF and PCQ enable query plans to optimize themselves dynamically but require the DBMS to generate all code for a given query before execution. This approach, referred to as one-shot

code generation, is widely adopted by most existing compilation-based DBMSs. A crucial drawback of one-shot code generation is that it prevents the DBMS from learning properties about the data the query reads in an earlier phase of processing to tailor the code it generates for a later phase. To address this, we observe that pipelines only share state, not code; pipelines form a DAG and can be compiled and executed independently. Thus, in Chapter 5 we presented PCG, which interleaves code generation and execution of the different pipelines constituting a query. PCG allows the DBMS to adapt pipelines based on data observed in earlier parts of a query. We use this framework to implement several operations targeting both performance and memory compression.

Taken together, the work described in this dissertation enables any DBMS that uses query compilation to achieve dynamic runtime robustness without succumbing to any of its overheads.

Bibliography

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

[2] Apache Cassandra. http://cassandra.apache.org/.

[3] Apache Spark. http://spark.apache.org/.

[4] As chip design costs skyrocket, 3nm process node is in jeopardy. https://www.extremetech.com/computing/272096-3nm-process-node.

[5] Google Benchmark. https://github.com/google/benchmark.

[6] HyPer. https://hyper-db.de.

[7] Making chips at 3nm and beyond. https://semiengineering.com/making-chips-at-3nm-and-beyond/.

[8] MemSQL. http://www.memsql.com.

[9] NoisePage. https://noise.page.

[10] OpenCL. https://www.khronos.org/opencl/.

[11] Peloton Database Management System. http://pelotondb.io.

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

[13] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006.

[14] D. J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, MIT, 2008.

[15] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980, 2008.

[16] J. Adzic, V. Fiore, and L. Sisto. Extraction, transformation, and loading processes. In *Data warehouses and OLAP: Concepts, architectures and solutions*, pages 88–110. IGI Global, 2007.

[17] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[18] Y. Ahmad and C. Koch. Dbtoaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[19] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software—Practice & Experience*, 40(2):131–147, 2010.

[20] A. Appleby. MurMur3 Hash. https://github.com/aappleby/smhasher.

[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[22] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[23] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, January 2005.

[24] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, January 2005.

[25] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, page 119–128, New York, NY, USA, 2006. Association for Computing Machinery.

[26] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[27] A. Böhm, J. Dittrich, N. Mukherjee, I. Pandis, and R. Sen. Operational analytics data management systems. *Proc. VLDB Endow.*, 9(13):1601–1604, 2016.

[28] P. Boncz, T. Neumann, and O. Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. 2014.

[29] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[30] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, December 2008.

[31] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its geographic extensions: A novel approach to high performance gis processing. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, page 147–166, Berlin, Heidelberg, 1996. Springer-Verlag.

[32] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[33] D. Broneske, A. Meister, and G. Saake. Hardware-sensitive scan operator variants for compiled selection pipelines. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 403–412, 2017.

[34] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24:632–646, October 1981.

[35] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 1–15, New York, NY, USA, 1991. Association for Computing Machinery.

[36] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 49–70, New York, NY, USA, 1989. Association for Computing Machinery.

[37] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, et al. Procella: Unifying serving and analytical data at youtube. *Proceedings of the VLDB Endowment*, 12(12):2022–2034, 2019.

[38] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127, 2004.

[39] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 271–282, 2001.

[40] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 150–160, 1994.

[41] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "big" data, big analytics, small clusters. In *CIDR*, 2015.

[42] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[43] P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. *Proc. VLDB Endow.*, 13(11):2396–2410, 2020.

[44] R. Delbru, S. Campinas, and G. Tummarello. Searching web data: An entity retrieval and high-performance indexing model. *Journal of Web Semantics*, 10:33–58, 2012.

[45] P. J. Denning and T. G. Lewis. Exponential laws of computing growth. *Commun. ACM*, 60(1):54–65, Dec. 2016.

[46] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[47] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. POPL '84, page 297–302, New York, NY, USA, 1984. Association for Computing Machinery.

[48] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *ACM International Conference on Management of Data 2013*, June 2013.

[49] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. PLDI '99, page 229–241, New York, NY, USA, 1999. Association for Computing Machinery.

[50] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker. Quantifying tpc-h choke points and their optimizations. *PVLDB*, 13(8):1206–1220, 2020.

[51] A. Dutt and J. R. Haritsa. Plan bouquets: Query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1039–1050, 2014.

[52] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[53] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, DMTCS Proceedings, pages 137–156, June 2007.

[54] J. Frazelle. Chipping away at moore's law: Modern cpus are just chiplets connected together. *Queue*, 18(1):5–15, Feb. 2020.

[55] C. Freedman, E. Ismert, and P. Larson. Compilation in the microsoft sql server hekaton engine. *IEEE Data Eng. Bull.*, 37:22–30, 2014.

[56] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[57] J. Giceva and M. Sadoghi. *Hybrid OLTP and OLAP*, pages 1–8. Springer International Publishing, 2018.

[58] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, Aug. 2015.

[59] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998.

[60] G. Graefe. Volcano- an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6:120–135, 1994.

[61] G. Graefe and K. Ward. Dynamic query evaluation plans. *SIGMOD Rec.*, 18(2):358–366, June 1989.

[62] P. M. Grulich, S. Breß, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl. Grizzly: Efficient stream processing through adaptive query compilation. SIGMOD, June

2020.

[63] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, Nov. 2010.

[64] T. Gubner, V. Leis, and P. Boncz. Efficient query processing with optimistically compressed hash tables strings in the ussr. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 301–312, 2020.

[65] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1917–1923, New York, NY, USA, 2015. Association for Computing Machinery.

[66] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):606–618, July 2006.

[67] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 981–992, New York, NY, USA, 2008. Association for Computing Machinery.

[68] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

[69] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, page 21–38, Berlin, Heidelberg, 1991. Springer-Verlag.

[70] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. PLDI '92, page 32–43, New York, NY, USA, 1992. Association for Computing Machinery.

[71] IBM. Java 2 platform, standard edition. https://www.ibm.com/developerworks/java/jdk/, 2013.

[72] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 268–277, 1991.

[73] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, page 24–35, New York, NY, USA, 2009. Association for Computing Machinery.

[74] H. JVM. Java version history. http://en.wikipedia.org/wiki/Javaversionhistory, 2013.

[75] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*,

1(2):1496–1499, Aug. 2008.

[76] F. L. O. S. K. O. L. H. Karl Rupp, M. Horowitz and C. Batten. Microprocessor trend data. 2020.

[77] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[78] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, Sept. 2018.

[79] T. Kersten, V. Leis, and T. Neumann. Tidy tuples and flying start: Fast compilation and fast execution of relational queries in umbra. *VLDB J*, 30, 2021.

[80] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[81] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[82] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *ICDE*, pages 197–208, 2018.

[83] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research*, 2015.

[84] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010.

[85] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1253–1258, 2015.

[86] T. Lahiri, M. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[87] D. Laney. 3-D data management: Controlling data volume, velocity and variety. Feb. 2001.

[88] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. SIGMOD '16, page 311–326, 2016.

[89] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, and W.-S. Han. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *Proc. VLDB Endow.*, 10(12):1598–1609, Aug. 2017.

[90] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[91] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[92] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively reordering joins during query execution. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 26–35, April 2007.

[93] T. Li, M. Butrovich, A. Ngom, W. McKinney, and A. Pavlo. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. 2019. *Under Submission*.

[94] R. MacNicol and B. French. Sybase iq multiplex - designed for analytics. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 1227–1230. VLDB Endowment, 2004.

[95] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid oltp+olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 37–50, New York, NY, USA, 2017. Association for Computing Machinery.

[96] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 659–670, 2004.

[97] J. C. McCallum. Memory prices 1957+. https://www.jcmit.net/memoryprice.htm, 2021.

[98] E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29(7).

[99] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, page 425–433, New York, NY, USA, 1999. Association for Computing Machinery.

[100] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[101] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11:1–13, September 2017.

[102] P. Menon, A. Ngom, L. Ma, T. C. Mowry, and A. Pavlo. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, October 2020.

[103] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. ICS '88, page 140–152, New York, NY, USA, 1988. Association for Computing Machinery.

[104] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. PACT '99, page 192, USA, 1999. IEEE Computer Society.

[105] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[106] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[107] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, 1992.

[108] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[109] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[110] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. SIGMOD, 2015.

[111] F. Özcan, Y. Tian, and P. Tözün. Hybrid transactional/analytical processing: A survey. New York, NY, USA, 2017. Association for Computing Machinery.

[112] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[113] R. B. G. L. I. Pandis, V. R. R. Sidle, G. A. N. C. S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[114] S. Pantela and S. Idreos. One loop does not fit all. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2073–2074, 2015.

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

[116] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proc. VLDB Endow.*, 11(6):663–676, Feb. 2018.

[117] A. Pavlo. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. PhD thesis, Brown University, 2013.

[118] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker. How i learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.

[119] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/doc/2657815/, 2014.

[120] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, 2016.

[121] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. SIGMOD, pages 1493–1508, 2015.

[122] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced simd processors. DaMoN '14, pages 6:1–6:6, 2014.

[123] M. Raasveldt and H. Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984. Association for Computing Machinery, 2019.

[124] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *SIGMOD*, pages 1231–1242, 2013.

[125] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. In *VLDB*, volume 6, pages 1080–1091, 2013.

[126] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 23–23. IEEE, 2006.

[127] L. Rauchwerger and D. A. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[128] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[129] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

[130] M. A. Roth and S. J. Van Horn. Database compression. *ACM Sigmod Record*, 22(3):31–39, 1993.

[131] J. Saltz, G. Agrawal, C. Chang, R. Das, G. Edjlali, P. Havlak, Y. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, and M. Uysal. Programming irregular applications: Runtime support, compilation and tools. *Adv. Comput.*, 45:105–153, 1997.

[132] F. Schiavio, D. Bonetta, and W. Binder. Dynamic speculative optimizations for sql compilation in apache spark. *Proc. VLDB Endow.*, 13(5):754–767, Jan. 2020.

[133] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on*

Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 1961–1976, 2016.

[134] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922, 2016.

[135] N. Shamgunov. The memsql in-memory database system. In *IMDM@VLDB*, 2014.

[136] J. Sompolski. Just-in-time Compilation in Vectorized Query Execution. Master's thesis, University of Warsaw, Aug 2011.

[137] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, 2011.

[138] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - db2's learning optimizer. In *VLDB*, pages 19–28, 2001.

[139] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. VLDB '05, page 553–564. VLDB Endowment, 2005.

[140] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? – part 2: benchmarking results. In *In CIDR*, 2007.

[141] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, page 2–11, USA, 2005. IEEE Computer Society.

[142] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[143] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). VLDB '07, page 1150–1160. VLDB Endowment, 2007.

[144] A. Suhan and T. Mostak. MapD: Massive Throughput Database Queries with LLVM on GPUs. http://devblogs.nvidia.com/parallelforall/mapd, June 2015.

[145] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 307–322, 2018.

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

[147] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[148] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1153–1170, New York, NY, USA, 2019. Association for Computing Machinery.

[149] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[150] A. Venkat, M. Hall, and M. Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 521–532, New York, NY, USA, 2015.

[151] S. D. Viglas. Just-in-time compilation for sql query processing. *PVLDB*, 6(11):1190–1191, 2013.

[152] S. Wanderman-Milne and N. Li. Runtime code generation in cloudera impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014.

[153] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM Sigmod Record*, 29(3):55–67, 2000.

[154] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 57–68, New York, NY, USA, 2013. Association for Computing Machinery.

[155] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. PLDI 2017, page 662–676. Association for Computing Machinery, 2017.

[156] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[157] A. X3.135-1992. American national standard for information systems — database language — sql. November 1992.

[158] S. Zeuch, H. Pirk, and J. Freytag. Non-invasive progressive optimization for in-memory databases. *PVLDB*, 9(14):1659–1670, 2016.

[159] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[160] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust. *PVLDB*, 10(8):889–900, 2017.

[161] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.

[162] M. Zukowski, N. Nes, and P. Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. DaMoN '08, pages 47–54, 2008.