Abstract of “Code Generation for In-Memory Data Analytics” by Andrew Crotty, Ph.D., Brown University, May 2019.

Recent advancements in hardware have caused a shift toward purely in-memory data processing, forcing a complete redesign of the high-overhead abstractions (e.g., Volcano-style iterators) at the core of traditional, disk-based systems. One popular replacement for these outdated query processing models is code generation, which refers to the process of generating query-specific, machine-executable code to evaluate a query. In general, code generation is highly efficient and enables a variety of low-level optimizations, yet it comes with its own set of drawbacks.

This dissertation provides an in-depth exploration of code generation for in-memory data analytics. First, we present two novel code generation strategies that jointly consider properties of the operators, data, and underlying hardware to significantly improve performance compared to existing code generation approaches. Then, in order to mitigate the main disadvantages associated with code generation, we propose a new, alternative query processing model that achieves comparable performance while avoiding these downsides.

Code Generation for In-Memory Data Analytics

by
Andrew Crotty

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2019

This dissertation by Andrew Crotty is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ________________ ________________________________
Tim Kraska, Director

Recommended to the Graduate Council

Date ________________ ________________________________
Uğur Çetintemel, Reader
Brown University

Date ________________ ________________________________
Kenneth A. Ross, Reader
Columbia University

Approved by the Graduate Council

Date ________________ ________________________________
Andrew G. Campbell
Dean of the Graduate School

Acknowledgments

Foremost, I would like to thank my advisor, Tim Kraska, for teaching me literally everything I know about how to do good research. He always pushed me to produce my best work, and I am deeply grateful for his mentorship. More often than not, his advice turned out to be right, and I regret not taking it more frequently.

Next, I would like to thank Alex Galakatos, without whom this dissertation would not have been possible. I have a tendency to set my sights a bit too high, but I can always count on him to keep me from straying too far afield. We have become great friends throughout this process, and I look forward to continuing to work together in the future.

I am incredibly grateful for all of the guidance I received from faculty members, including my committee members Ugur Cetintemel and Ken Ross, as well as Carsten Binnig, Stan Zdonik, Rodrigo Fonseca, and my undergraduate advisor, Ed Sciore, who initially sparked my interest in data management research and encouraged me to apply to a Ph.D. program. I would also like to thank the many students with whom I worked, including Kayhan, Emanuel, Philipp, John, Erfan, Yeounoh, Eric, Sam, and Ani. I am privileged to count all of you among my friends.

Finally, I would like to thank my family—I attribute a large part of my success to their love and encouragement. In particular, I would like to thank my mother for always supporting me in my pursuits, my father for teaching me to always do the right thing, and Abigail for always being my biggest advocate. I love you all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The growing prevalence of big data across all industries and sciences is causing a profound shift in the nature and scope of analytics. In order to extract value from this ever-increasing pool of data, a wide variety of techniques—ranging from simpler online analytical processing (OLAP) queries to more complex machine learning (ML) algorithms—have become an integral part of the data-driven decision-making process.

Most traditional, disk-based systems for performing these data analytics tasks utilize an iterator-based query processing model (e.g., Volcano [64]) due to simplicity of design and ease of implementation, since the inherent overheads of the iterator abstraction are masked by the high costs of disk I/O. However, recent hardware advancements and the steady decline in memory prices have precipitated a transition toward purely in-memory data processing, with all but the largest datasets fitting entirely in main memory. For these in-memory systems, the iterators themselves can become a bottleneck, and *code generation* has emerged as a popular replacement.

Code generation is the process of generating query-specific, machine-executable code in order to evaluate a query. Although originally proposed as early as System R [25], code generation has recently enjoyed a renewed interest, with a huge number of both academic and commercial systems [42, 49, 56, 75, 78, 81, 82, 90, 101] demonstrating significant performance improvements for in-memory query processing. The query evaluation code produced by these types of systems is typically much more efficient than traditional iterator-based processing, since the generated code has been specialized and optimized specifically for a particular query, with all additional sources of overhead removed.

Yet, while code generation is an attractive option because it enables substantially more flexibility and specialization in query processing, it is complicated to implement, maintain, and debug. For example, IBM actually abandoned code generation in System R because of how difficult it was to maintain [113]. Moreover, the cost of performing on-the-fly code generation can become so expensive that recent work [79] proposed a hybrid interpreted execution model where queries are incrementally generated, compiled, and optimized in the background to avoid large up-front costs, but large query volumes can still result in cumulatively prohibitive overhead. For these reasons, system designers must carefully weigh the performance benefits that can be achieved via code generation against (1) implementation complexity, (2) codebase maintainability, and (3) runtime overhead.

This dissertation explores the intricacies of code generation for in-memory data analytics. Specifically, we argue that, while code generation presents myriad opportunities for unique optimizations, many of the most impactful of these optimizations can actually be replicated without needing to perform any code generation, thereby drastically improving performance with none of the associated downsides. To that end, we begin by presenting a wide range of novel code generation techniques that achieve significant improvements compared to the state of the art, and then we show how these techniques can be implemented without relying on code generation at all.

In the rest of this chapter, we explain the key concepts behind code generation using a concrete example, followed by a discussion of the main pros and cons of code generation approaches. We conclude the chapter by outlining the key contributions of our work.

The remainder of this document is organized as follows. Chapter 2 presents a novel code generation strategy that jointly considers properties of the operators, data, and underlying hardware to generate optimized query execution code on a case-by-case basis. Then, in Chapter 3, we introduce an extension of these techniques that eschews traditional query operators in favor of laying out code optimized at the granularity of the entire query. In Chapter 4, we revisit both of these code generation strategies and instead propose a new vectorized query processing model that implements many of the same optimizations without resorting to any code generation. Chapter 5 provides a detailed survey of the related work. Finally, we summarize our contributions and outline interesting avenues for future work in Chapter 6.

1.1 Code Generation

Iterator-based execution, which was popularized by Volcano [64], is the most common query processing model for traditional, disk-based DBMSs. In this paradigm, each operator (e.g., select, project, join) is implemented as an iterator with a `next` function that returns the next tuple to the caller, and an execution engine can answer an arbitrary query by nesting iterators to match the query plan. This abstraction is simple, flexible, and easy for system developers to reason about.

However, with the advent of in-memory query processing, the high-overhead iterator model was abandoned in favor of more efficient alternatives. One such alternative currently in vogue is code generation, which involves generating query-specific, machine-executable code to evaluate a query.

In this section, we begin by providing a straightforward example to illustrate the core concepts of code generation. Then, we outline some of the key advantages and major drawbacks of this approach.

1.1.1 An Illustrative Example

TPC-H [14] is perhaps the most well-known and widely utilized OLAP benchmark. The queries mimic an ad hoc, decision support workload in a realistic business scenario. As such, we refer to several TPC-H queries throughout this document in order to concretely illustrate different algorithms and approaches.

Figure 1.1 shows an example of code generation using TPC-H Q6. The SQL for the query appears in Figure 1.1a, while an example of generated code appears in Figure 1.1b. Q6 is a “what-if” query that computes the increase in revenue for a particular historical year that would have occurred by eliminating specified

```
select
   sum(l_extendedprice * l_discount) as revenue
from
   lineitem
where
   l_shipdate >= '1994-01-01'
   and l_shipdate < '1995-01-01'
   and l_discount between 0.05 and 0.07
   and l_quantity < 24
```

(a) SQL for Q6

```
size_t i;
revenue = 0.0;
for (i = 0; i < l_len; i++)
   if (l_shipdate[i] >= '1994-01-01'
         && l_shipdate[i] < '1995-01-01'
         && l_discount[i] >= 0.05
         && l_discount[i] <= 0.07
         && l_quantity[i] < 24)
      revenue += l_extendedprice[i] * l_discount[i];
```

(b) Generated Query Evaluation Code (data-centric [95])

Figure 1.1: Example Query – TPC-H Q6

discounts. This query is structurally simple, involving only a single table (`lineitem`) and applying four predicates before performing a straightforward scalar aggregation.

The generated code shown in Figure 1.1b is based on the data-centric [95] approach, which we use as a simple and intuitive example of code generation. The data-centric implementation involves a `for` loop over all tuples stored in the `lineitem` table in order to compute the `revenue` (i.e., `sum(l_extendedprice * l_discount)`) of the subset of tuples that pass the selection criteria. Since this approach uses code generation, all of the predicates are merged together into a single `if` conditional statement rather than being handled independently by precompiled selection iterators. Similarly, the `sum` and `multiply` operations are merged into a single statement.

Merging operations together in this way maximizes the benefits of short-circuit evaluation and data locality while removing the overheads of the iterator model. However, the data-centric approach unfortunately prevents many useful low-level optimizations (e.g., SIMD vectorization) due to the control dependency introduced by the branching code.

It is fairly obvious why a code generation version is much more efficient than traditional, iterator-based approaches. Specifically, the selection component of the query requires five comparison instructions and four branching instructions, and merging them together into a single statement maximizes the benefits of short-circuit evaluation. Moreover, there is no overhead from external function calls, indirections, or other sources of inefficiency.

However, code generation comes with its own set of challenges. The question naturally arises, then: when is it a good idea to use code generation? In the following two sections, we consider some of the key arguments both for and against code generation.

1.1.2 Code Generation Benefits

The performance benefits of code generation are very well-documented [42, 77, 82, 90, 95, 122]. This section highlights some of the main sources of these performance benefits.

Type Specialization

Type specialization is an optimization that involves tailoring algorithms or data structures for specific data types rather than in a generic fashion. For example, type specialization might help when performing a group-by aggregation. When the group-by key is a 4-byte integer, a hash table would be a natural candidate. On the other hand, if the group-by key is only 1-byte in size, it would be more efficient to allocate an array of size 256 and instead of using a hash-based approach, simply use the 1-byte keys as a direct offset into the array.

Since systems that use code generation know the full query specification, they can produce code that is highly specialized for the data types used only in that particular query, rather than trying to piece together generic operator building blocks in an interpreted execution model. Since all types are known at compile-time, this approach can completely eliminate runtime casts or switch statements to determine type-specific processing, as well as enable a variety of low-level optimization benefits at the compiler level, such as better register and cache allocation.

However, code generation might not necessarily be required in order to perform type specialization for DBMS operators. In fact, type specialization is quite straightforward to achieve in an interpreted system. For example, it is possible to use macros for the C preprocessor to generate type-specific implementations or template specialization of generic code in C++. We explore this possibility in later chapters.

Minimal Overhead

Another major benefit of code generation is the ability to completely remove much of the baggage associated with high-level abstractions like Volcano-style iterators. By generating query-specific code, there is no longer a need for polymorphism or dynamic dispatch of function calls. Moreover, the compiler can perform more aggressive optimizations than would be possible in an interpreted environment, such as inline expansion for functions and loop optimizations.

Again, as is the case with type specialization, these optimizations can be leveraged by implementing static DBMS operators with code written in a way conducive to automatic compiler optimization, or even by implementing hand-tuned and highly optimized versions of those operators. However, the types of cross-operator optimizations enabled by code generation would still remain out of reach.

Holistic Optimization

One of the most important benefits of code generation that has not yet been replicated in an interpreted query execution engine is holistic optimization, which is the ability of a query optimizer to reason about and optimize across traditional operator boundaries. For example, whereas an iterator-based execution engine would implement each selection operation from Q6 as a separate operator, a system that uses code generation can seamlessly merge those selection predicates into a single conditional statement (Figure 4.1a). In the extreme case, a code generation engine can even merge all of the Q6 operations into a single imperative statement, and we describe techniques that apply similar optimizations in later chapters.

Overall, the main advantage of holistic optimization is the restructuring of execution code tailored to individual queries on a case-by-case basis to unlock opportunities for low-level optimizations (e.g., SIMD vectorization, maintaining CPU register locality). Therefore, even though vectorized query processing can often get close to the performance of code generation systems through careful implementation, the final mile separating the two approaches can only be overcome through holistic optimization.

1.1.3 Code Generation Drawbacks

While the performance benefits of code generation are undisputed, they are accompanied by several clear drawbacks that have been largely overlooked in the literature. This section describes some of these drawbacks.

Clean Slate

Since the creation of the earliest DBMSs in the 1960s, the area of data management has received widespread attention from researchers in both academia and industry, as well as broad interest in the commercial arena. These collective efforts have produced a rich and extensive body of work that forms the foundations of the field, as well as a variety of battle-tested systems deployed in the wild.

Unfortunately, while attempts to integrate code generation techniques into legacy architectures like PostgreSQL have succeeded in dramatically improving query performance from the interpreted baseline [34, 89], they still fall consistently short of the performance achieved by state-of-the-art code generation systems. For example, the Peloton project [104] originally forked PostgreSQL as a starting point but eventually needed to restart completely from scratch due to prohibitive legacy design decisions [103].

The bottom line of all these trials is that existing architectures and design paradigms, such as the iterator model, break down completely in the context of code generation. Therefore, designers of code generation systems necessarily need to throw out existing systems and start from scratch. While the enthusiasm for yet another "complete rewrite" might benefit the academic paper-writing industry, this mentality completely disregards the enormous costs associated with building a new DBMS from the ground up.

Compilation Overhead

In code generation systems, queries can be compiled either a priori (e.g., when declaring a stored procedure) or just-in-time (JIT) compiled (e.g., at runtime when issuing an ad hoc query). Given the exploratory nature of OLAP workloads, most queries will end up needing to be JIT compiled as they are issued to the system. Unfortunately, this process can incur a significant amount of associated overhead, making it difficult to do on-the-fly code generation and optimization in a timely fashion, particularly for short-lived queries, or large query volume environments.

This problem is so pronounced that recent work actually proposed an adaptive execution strategy to switch between an interpreted execution model and a code generation execution engine [79]. Not only does this hybrid execution approach end up consuming additional system resources that could otherwise be used for query execution, but it leads to the additional problem of having to maintain two completely separate execution engines that still need to remain fully interoperable, which is an enormous engineering challenge.

Difficulty of Implementation

The biggest drawback of code generation systems is that their design is difficult to reason about and they are, consequently, hard for software engineers to implement, maintain, and extend. For example, code generation as part of the System R project was abandoned by IBM because it was too complicated and brittle, not portable, and prohibitively expensive [90, 113]. Moreover, code generated queries are even harder to debug, since they are generated by machines in an often idiosyncratic manner that is not straightforward for humans to understand.

Modern DBMSs are already highly sophisticated and intricate pieces of software, often written by hundreds (or even thousands) of software engineers over a period of decades. For example, SQLite [12] is an extremely lightweight DBMS, and its codebase already contains approximately $129,000$ lines of code [5]. Other DBMS mainstays include PostgreSQL [11] and MySQL [10], which are, by now, implemented in well over a million lines of code each. Given the added complexity of code generation, how many more lines of code and man-hours would be required to build an equivalent code generation DBMS?

1.2 Key Contributions

The key contributions of this work are divided into three chapters, each of which tackles a different aspect of code generation for in-memory data analytics. In the following, we provide a high-level summary of each of these contributions, as well as their broader impact.

1.2.1 Operator-centric Code Generation

Chapter 2 presents TUPLEWARE, an architecture for compiling UDF-centric workflows, and proposes a novel operator-centric code generation strategy that jointly considers properties of the operators, data, and underlying hardware to generate optimized query execution code on a case-by-case basis. Unlike data-centric code generation, which attempts to maximize data pipelining, our operator-centric approach looks for opportunities to split fused operators into sub-pipelines and materialize vectors of intermediate results between them. We demonstrate that TUPLEWARE provides significant performance improvements over state-of-the-art alternatives.

1.2.2 Query-centric Code Generation

Chapter 3 introduces an extension of operator-centric code generation, called BESPOKE, that discards the notion of traditional operators in favor of a more holistic, query-centric approach to code generation. Since code generation systems are no longer constrained by traditional operators, we present several techniques to optimize across operator borders, freely reordering operations at the granularity of the entire query. Contradictory to the traditional wisdom, BESPOKE heavily leverages “predicate pullups” to produce more streamlined code that outweighs the penalties of performing wasted work. Our experiments show that our query-centric code generation strategy is able to outperform both data- and operator-centric approaches over a diverse range of queries.

1.2.3 Rethinking Code Generation

Based on the discussion of the pros and cons of code generation, Chapter 4 proposes ERSATZ, a new vectorized query processing model that achieves similar performance to our novel code generation techniques while avoiding the associated downsides. In particular, ERSATZ addresses the problem of holistic optimization, which was previously impossible for traditional query processing approaches, by introducing a new set of fused query operators. These fused operators allow ERSATZ to apply the exact same optimizations described throughout Chapters 2 and 3, and our experimental evaluation demonstrates that we can achieve nearly identical performance to code generation approaches without resorting to any code generation whatsoever.

Chapter 2

Operator-centric Code Generation

This chapter describes TUPLEWARE, a novel architecture for the automatic compilation of UDF workflows for in-memory data analytics. Yet, while we focus here on the broader category of UDF workflows, many of the techniques described as part of this work apply equally well to the more specialized subcategory of SQL operators, as we show in later chapters.

In particular, we propose an operator-centric code generation strategy that, unlike static approaches, jointly considers properties of the operators, data, and underlying hardware to generate optimized query execution code on a case-by-case basis. The key idea of TUPLEWARE is to integrate high-level query optimization techniques with low-level compiler techniques in order to unlock a new breed of code generation optimizations that were previously impossible.

The remainder of this chapter is organized as follows. Section 2.1 provides a high-level overview of TUPLEWARE's architecture. In Section 2.2, we describe the important aspects of the user frontend. Section 2.3 explains our process for compiling UDF-centric workflows. In Section 2.4, we outline the key optimizations that are part of our operator-centric code generation strategy. Section 2.5 briefly mentions some of the primary distributed execution challenges. Finally, we present our evaluation in Section 2.6.

2.1 Tupleware

TUPLEWARE leverages the LLVM [87] compiler framework in a novel way to: (1) provide a language-agnostic frontend that lets users choose from a wide variety of programming languages with minimal overhead; and (2) allow our compilation process to introspect UDFs and gather statistics used for applying low-level optimizations. While prior work has independently investigated LLVM for SQL query compilation [95] and UDF introspection for high-level workflow optimizations [35, 68], TUPLEWARE is the first to combine these two ideas in order to optimize UDF workflows at the code generation level. This section provides a high-level overview of TUPLEWARE and introduces a running example that we reference throughout this chapter.

2.1.1 Architecture

As shown in Figure 2.1, our proposed architecture consists of three parts. The *Frontend* (Section 2.2) allows users to define workflows of UDFs directly inside any LLVM-supported host language using operators like map and reduce. These workflows are translated to optimized, self-contained distributed programs during the *Compilation* process (Section 2.3). Compiled workflows are then executed automatically on a cluster during the *Execution* phase (Section 2.5).

This chapter primarily focuses on our novel process for compiling UDF workflows. Although other systems [35, 68] introspect UDFs to infer high-level semantics (e.g., whether a UDF performs a selection) for applying rewrite rules, our approach goes a step further by determining low-level characteristics (e.g., the approximate number of CPU cycles) to generate better code.

2.1.2 Example Workflow

Figure 2.2 depicts the full lifecycle of a workflow. First, the user composes the workflow on the client-side using operators from our API to transform an object called a T-Set, which is similar to a Resilient Distributed Dataset (RDD) in Spark. The example workflow shown in the figure corresponds to k-means, an iterative algorithm that groups instances into one of *k* clusters, which we use as a running example throughout this chapter. This workflow constructs a T-Set from the `data.csv` file and transforms it using the specified UDFs (e.g., `distance`, `minimum`). UDFs can be authored in the host language either as a named function (shown in the figure) or inline as an anonymous function.

To compute the result, the client sends a directed graph representing the workflow and the LLVM *intermediate representation* (IR) for each UDF to the server. These pieces are then converted into a distributed program during the Compilation process, which consists of the (1) *UDF Analyzer*, (2) *Optimizer*, and (3) *Linker*. The UDF Analyzer introspects each UDF by examining the LLVM IR to gather statistics for predicting execution behavior. The Optimizer translates the workflow graph into a distributed program by generating execution code with embedded references to the associated UDFs. As shown in Figure 2.2, this execution code includes all control flow (e.g., the inner `for` loop over a data block), communication (e.g., the `getBlock()` data request mechanism), and synchronization (e.g., the `sync()` function) components necessary to form a self-contained distributed program. During code generation, the Optimizer uses the UDF statistics to apply low-level optimizations that specifically target the underlying hardware. The Linker then merges the LLVM IR for the UDFs with the generated execution code, and the distributed program is then deployed for execution on the cluster.

2.2 Frontend

In many regards, TUPLEWARE's API is similar to other recent frameworks (e.g., Spark [3], Flink [1], DryadLINQ [131]), where users compose workflows on a data-parallel abstraction directly inside a host language. However, despite the importance of shared state for complex analytics tasks, few frameworks treat shared state as a fundamental component of their programming models. We therefore make shared state

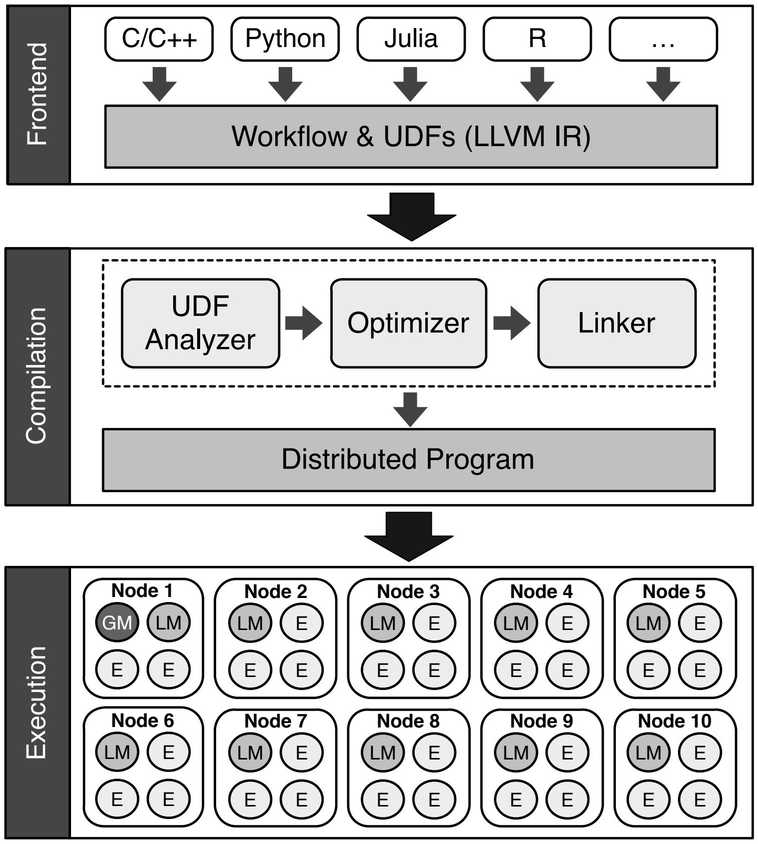


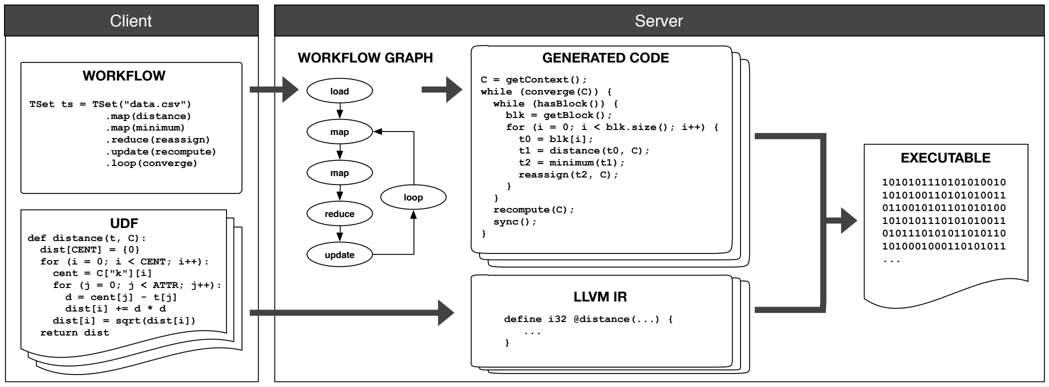
Figure 2.1: An overview of our proposed architecture. The Frontend allows users to compose UDF workflows in any LLVM-supported language (top boxes). These workflows are then translated to self-contained distributed programs during the three-stage Compilation process. Finally, the Execution phase deploys the distributed programs on a cluster, shown as 10 nodes (labeled boxes) each with four cores (circles inside the boxes) that have specialized execution roles (GM, LM, E).

explicit by extending the traditional data-parallel abstraction to include global variables that are logically shared across all nodes in the cluster. We call the resulting abstraction a *T-Set*.

Definition 1 (T-Set) *A T-Set is a pair* (R, C)*, where* R *is a relation, which is a set of n-tuples, and* C *is a Context, which is a dictionary (i.e., set of key-value pairs) of shared state variables.*

Users can chain together operators like map and reduce to define workflows that transform a T-Set. We formally define an operator as a second-order function that takes zero or more T-Sets as input and produces a new T-Set as output by invoking an associated first-order UDF. For example, a map operator returns a new T-Set by applying the supplied UDF to each element of the input T-Set's relation R. Table 2.1 shows some of the most common operators in our API.

Although we incorporate the best features from other frameworks, our Frontend distinguishes itself through (1) low-level optimizability, (2) explicit shared state, and (3) an LLVM foundation.



```
TSet ts = TSet("data.csv")
            .map(distance)
            .map(minimum)
            .reduce(reassign)
            .update(recompute)
            .loop(converge)
```

```
def distance(t, C):
  dist[CENT] = {0}
  for (i = 0; i < CENT; i++):
    cent = C["k"][i]
    for (j = 0; j < ATTR; j++):
      d = cent[j] - t[j]
      dist[i] += d * d
    dist[i] = sqrt(dist[i])
  return dist
```

```
C = getContext();
while (converge(C)) {
  while (hasBlock()) {
    blk = getBlock();
    for (i = 0; i < blk.size(); i++) {
      t0 = blk[i];
      t1 = distance(t0, C);
      t2 = minimum(t1);
      reassign(t2, C);
    }
  }
  recompute(C);
  sync();
}
```

```
define i32 @distance(...) {
   ...
}
```

```
1010101110101010010
1010100110101010011
0110010101101010100
1010101110101010011
0101110101011010110
1010001000110101011
...
```

Figure 2.2: The full lifecycle of a workflow.

Category	Operator	UDF Signature	Optimizations
Apply	map$(T)(\lambda)$ flatmap$(T)(\lambda)$ filter$(T)(\lambda)$	$(t, C?) \rightarrow t'$ $(t, C?) \rightarrow \{t'\}$ $(t, C?) \rightarrow b$	Section 2.4.1
Aggregate	reduce$(T)(\lambda, k?)$	$(t_1, t_2) \rightarrow t'$ $(t, C) \rightarrow ()$	Section 2.4.2
Relational	selection$(T)(\lambda)$ join$(T_1, T_2)(\lambda)$	$t \rightarrow b$ $(t_1, t_2) \rightarrow b$	Section 2.4.3
Control	loop$(T)(\lambda)$ update$(T)(\lambda)$	$C \rightarrow b$ $C \rightarrow ()$	-

Table 2.1: A subset of the operators in our API, showing their categories and UDF signatures. Operators take zero or more T-Sets T as input and apply a UDF λ to produce a new T-Set as output. The UDF signatures specify the arguments and return types of each operator, with optional arguments denoted by the $?$ symbol. For example, the expected signature of a map UDF is: $(t, C?) \rightarrow t'$ where t is an input tuple, $C?$ is an optional Context, and t' is an output tuple.

2.2.1 Low-Level Optimizability

Our API includes many of the same operators offered by other frameworks, but we impose several additional requirements that play a crucial role in our Compilation process. In contrast to traditional frameworks that do not perform code generation for UDF workflows, our Optimizer leverages the nuances of the operator semantics to generate different code on a case-by-case basis. Each of these subtle yet important differences corresponds to an optimization heuristic described in Section 2.4. As shown in Table 2.1, we divide operators into four categories.

Apply

Apply operators invoke a UDF on every tuple in a T-Set's relation. Traditional MapReduce has a single *map* operator that can return an arbitrary number of output tuples for each input tuple (i.e., *0-to-N* mapping). More recent frameworks already distinguish between a more restrictive map operator for a strict *1-to-1* mapping (i.e., the UDF takes one input tuple t and must return exactly one output tuple t'), a *flatmap* operator for

1-to-(0:N) mappings, and a *filter* operator for *1-to-(0:1)* mappings. Unlike other frameworks that do not compile UDF workflows, we leverage these more detailed semantics to generate more efficient control flow code (Heuristic 1a).

As shown in Table 2.1, apply UDFs have read-only access to an optionally provided Context $C?$. For example, the k-means `distance` UDF (Figure 2.2) reads the current centroid values from `C["k"]`. Since these UDFs have read-only access to the Context, apply operators can execute safely in parallel without conflicts.

Aggregate

Aggregate operators perform a group-by computation on a T-Set's relation. Like Spark, our *reduce* operator expects a commutative and associative UDF (e.g., sum, count) of the form: $(t_1, t_2) \rightarrow t'$ where input tuples t_1 and t_2 are combined to yield an output tuple t'. However, unlike Spark, which operates implicitly on RDDs of key-value pairs, we allow users to specify an explicit key function k that defines the group-by semantics (i.e., k takes a tuple t and returns its group-by key). Explicitly specifying a group-by key function allows us to generate code that better utilizes the hardware based on the characteristics of k (Heuristic 1b).

Table 2.1 also includes an alternative UDF signature for a reduce: $(t, C) \rightarrow ()$ where t is an input tuple and C is the Context. Instead of returning an output tuple t', this alternative reduce UDF can aggregate values by updating Context variables. Context aggregation variables are similar to Spark's accumulator objects, but they additionally (1) permit multiple keys; (2) can be read from within the workflow; and (3) have different distributed update patterns. By performing an aggregation using Context variables when the result cardinality (i.e., the number of distinct keys) is known a priori, we can generate more efficient code that replaces expensive dictionary lookups with static memory addresses at compile time (Heuristic 2).

Relational

Like other frameworks, our Frontend benefits from including traditional SQL transformations. For example, a *selection* expects a predicate UDF of the form: $t \rightarrow b$ where t is an input tuple and b is a Boolean value specifying whether t satisfies the predicate; that is, the user defines a (potentially compound) predicate from the set of operations $\{=, \neq, >, \geq, <, \leq\}$ that returns `true` if t should be selected and `false` otherwise. These semantics allow us to dynamically generate different selection code that considers both UDF complexity and selectivity (Heuristic 3).

Relational operators interact only with a T-Set's relation and can neither read nor update Context variables. This restriction avoids dependencies that would otherwise prevent standard query optimization techniques (e.g., predicate pushdown, join reordering). Note, however, that operators such as *join* merge the keys of two Contexts without changing their values, performing SQL-style disambiguation of conflicting keys at runtime.

Control

In order to support iterative workflows, our API also includes a *loop* operator that models a tail-recursive execution of the workflow; that is, the entire workflow is repeatedly reevaluated while the supplied loop continuation condition holds. For instance, the `converge` UDF from Figure 2.2 returns `true` until the centroids have converged. This UDF has read-write access to Context variables for maintaining values across iterations (e.g., loop counters, convergence criteria). By explicitly handling iterations as part of the workflow, we can perform cross-iteration optimizations (e.g., caching loop invariant data, leveraging data locality) that would be impossible with iterations managed by an external driver program.

Finally, our API provides an *update* operator that executes in a single thread to permit direct modification of Context variables. The k-means example uses the update operator (`recompute`) to calculate the new centroid values by computing the average from the sum and count Context variables.

2.2.2 Explicit Shared State

Shared state is an essential component of complex analytics tasks, but prior attempts to add distributed shared state to existing frameworks restrict how and when UDFs can interact with global variables. For example, Iterative Map-Reduce-Update [31] offers primitives designed for iterative refinement algorithms and cannot model non-convex optimization problems (e.g., neural networks, maximum likelihood Gaussian mixtures), as stated in their paper. Spark also provides several globally distributed primitives (e.g., accumulators, broadcast variables), but these objects are read-only within a workflow and cannot be used to represent shared state that changes frequently (e.g., ML models).

We overcome these limitations by providing three different update patterns for reduce UDFs that use Context aggregation variables: (1) *parallel* (conflicting updates must be commutative and associative); (2) *synchronous* (exclusive locks prevent conflicting updates); and (3) *asynchronous* (the algorithm must ensure correctness). For example, an implementation of stochastic gradient descent can use synchronous updates so that all changes to the shared model are immediately visible to all workers, while an implementation of Hogwild! [114] could use asynchronous updates to improve performance. This work focuses only on optimizations for parallel updates.

Our programming model uses monads, which can be thought of simply as "programmable semicolons," to define the order in which operators that access the shared state must be evaluated. More formally, monads impose a happened-before relation [83] between operators; that is, an operator O that modifies Context variables referenced by another operator O' must be fully evaluated prior to evaluating O'. While interesting, the precise monadic formalisms are not essential for the techniques discussed in this work.

2.2.3 LLVM Foundation

As previously mentioned, our architecture leverages the LLVM compiler framework to make our Frontend language-agnostic, allowing users to compose workflows of UDFs in a variety of programming languages (e.g., C/C++, Python, Julia, R) with minimal overhead. Adding a new LLVM-supported language is as simple as writing the necessary wrappers to implement our API. This approach is in contrast to other frameworks that pay a high boundary crossing penalty to support new languages (e.g., Spark must serialize objects between Java and Python).

LLVM also enables UDF introspection (irrespective of host language) to provide certain correctness

UDF	**Type**	**Vectorizable**	**Compute Time** Predicted	Actual	**Load Time**
`distance`	map	yes	30	28	3.75
`minimum`	map	yes	36	38	7.5
`reassign`	reduce	no	16	24	5.62
`recompute`	update	no	30	26	0

Table 2.2: UDF statistics for the k-means algorithm.

guarantees at compile time (e.g., if a selection UDF returns a Boolean value), though some requirements are impossible to check (e.g., if a reduce UDF is commutative and associative). Like other frameworks, we rely on the user to provide a correct UDF in these undecidable cases. Furthermore, our Optimizer can leverage UDF statistics from the LLVM IR to generate better code, which we describe in the following section.

2.3 Compilation

The main goal of our approach is to improve the performance of compute-intensive, complex analytics tasks by compiling UDF-centric workflows. We jointly consider characteristics of the data, UDFs, and underlying hardware in order to apply low-level optimizations on a case-by-case basis. This section outlines our Compilation process, which generates a distributed program from a workflow of UDFs. As shown in Figure 2.1, this process consists of three parts: (1) UDF Analyzer, (2) Optimizer, and (3) Linker.

2.3.1 UDF Analyzer

Systems that treat UDFs as black boxes have difficulty making informed decisions about how best to execute a given workflow. By leveraging the LLVM framework, we can look inside UDFs to gather statistics that help the Optimizer generate better code. The UDF Analyzer examines the LLVM IR of each UDF to determine several features, including vectorizability, computation cycle estimates, and memory access time predictions. As an example, Table 2.2 shows the UDF statistics for the k-means example from Section 2.1.

Vectorizability

Vectorizable UDFs can use *single instruction, multiple data* (SIMD) registers to achieve data parallelism. For instance, a 256-bit SIMD register on an Intel E5 processor can hold 8×32-bit floating-point values, offering a potential $8\times$ speedup. We can leverage the operator semantics from Section 2.2 to detect two types of vectorizable UDFs: (1) *1-to-1* maps and (2) single-key reduces (i.e., scalar aggregations). In the k-means example, only the `distance` and `minimum` UDFs are vectorizable.

Compute Time

CPI measurements [6] provide cycles per instruction estimates for the given hardware. Adding together these estimates yields a rough overall estimate of the total UDF compute time, but runtime factors (e.g., instruction pipelining, out-of-order execution) can make these values difficult to predict accurately. Furthermore, the compute time for UDFs containing data-dependent control flow code is impossible to predict; in these cases, we make a conservative estimate that assumes the fewest number of cycles. For most UDFs, though, we find that our predictions typically differ from the measured compute times by only a few cycles.

Load Time

Load time refers to the number of cycles necessary to fetch UDF operands from memory. If the memory controller can fetch UDF operands faster than the CPU can process them, then the UDF is *compute-bound*; otherwise, the CPU becomes starved, and the UDF is *memory-bound*. Load time is given by:

$$Load\,Time = \frac{Clock\ Speed \times Operand\ Size}{Bandwidth\ per\ Core} \tag{2.1}$$

For example, the load time for the `distance` UDF as shown in Table 2.2 with 32-bit floating-point (x, y) pairs using an Intel E5 processor with a 2.8GHz clock speed and 5.97GB/s memory bandwidth per core is calculated as follows: $3.75\ cycles = \frac{2.8GHz \times (2\times 4B)}{5.97GB/s}$.

2.3.2 Optimizer

The key idea of operator-centric code generation is to integrate high-level query optimization techniques with low-level compiler techniques in order to apply new optimizations that were previously impossible. The Optimizer translates a workflow into a distributed program by generating all of the necessary control flow, synchronization, and communication code with embedded references to the UDFs, as shown in Figure 2.2. While generating this code, our Optimizer can apply a broad range of optimizations that occur on both a logical and physical level, which we divide into three categories.

High-Level

We utilize well-known query optimization techniques, including predicate pushdown and join reordering. Additionally, our purely functional programming model allows for the integration of other traditional optimizations from the programming language community. All high-level optimizations rely on metadata and algebra semantics, information that is unavailable to compilers, but are not particularly unique to our approach.

Low-Level

Unlike other systems that use interpreted execution models, Volcano-style iterators, or remote procedure calls, our code generation approach eliminates much associated overhead by compiling in these mechanisms. We also gain many compiler optimizations (e.g., inline expansion, SIMD vectorization) “for free” by compiling workflows, but these optimizations occur at a much lower level than DBMSs typically consider.

Combined

Some systems incorporate DBMS and compiler optimizations separately, first performing algebraic transformations and then independently generating code based upon a fixed strategy. On the other hand, our approach combines an optimizable high-level algebra and statistics gathered by the UDF Analyzer with the ability to dynamically generate code, enabling optimizations that would be impossible for either a DBMS or compiler alone. In particular, our Optimizer considers (1) high-level algebra semantics, (2) metadata, and (3) low-level UDF statistics together in order to generate different code on a case-by-case basis. We describe several of these optimizations in Section 2.4.

2.3.3 Linker

After translating the workflow to a distributed program, the generated code has several embedded references to the supplied UDFs. This code then needs to be merged with the LLVM IR for each referenced UDF. The Linker performs the merging process, using an LLVM pass to combine them. It is often beneficial to perform inline expansion, and the Linker replaces call sites directly with the UDF body, providing further performance improvements over frameworks that require external function calls.

2.4 Operator-centric Optimizations

The most interesting and unique opportunities for optimizing a UDF workflow fall into the third category described in Section 2.3, which combines high-level query optimization techniques with low-level compiler techniques to produce a new class of optimizations that were previously impossible. This section describes a novel optimization process that considers the data, UDFs, and underlying hardware together in order to generate different code on a case-by-case basis. As a first step towards exploring this new class of optimizations, we propose heuristics for the following three scenarios: (1) program structure, (2) aggregation, and (3) selection. While these heuristics do not represent an exhaustive list of all optimizations that we perform, they apply to many common use cases and contribute to the overall speedup over other systems in our benchmarks (Section 2.6).

2.4.1 Program Structure

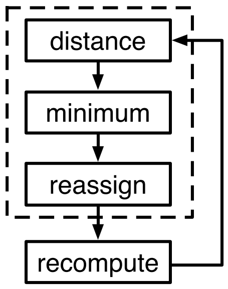
The first optimization we examine considers the most fundamental aspect of any code generation strategy, the overall *program structure*, which refers to the organization of the generated control flow code. This section describes two existing approaches to program structure and then presents our first heuristic, which allows our Optimizer to dynamically construct a hybrid program structure based on low-level UDF characteristics. Then, we propose an extension to this heuristic specifically for group-by aggregations.

Existing Strategies

Existing systems that compile queries rely on a static code generation strategy. These approaches advocate for a single dominant program structure and generate the same code in all situations.

Pipeline: The *pipeline* [95] (i.e., data-centric) strategy maximizes data locality by performing as many sequential operations as possible per tuple. Operations called pipeline breakers force the materialization of intermediate results. For example, a reduce forces materialization of an aggregation result, while consecutive maps can be pipelined. The following pseudocode shows the pipeline approach to the k-means example from Section 2.1.

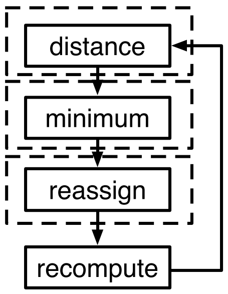
```
data[N];
while (converge()) {
  for (i = 0; i < N; i++) {
    dist = distance(data[i]);
    min = minimum(dist);
    reassign(min);
  }
  recompute();
}
```



The pipeline strategy has the major advantage of requiring only a single pass through the data. Additionally, a tuple is likely to remain in the CPU registers for the duration of processing, resulting in excellent data locality.

Operator-at-a-time: In contrast, the *operator-at-a-time* (i.e., vectorized [30] [1]) strategy performs a single operation at a time for all tuples. This bulk processing approach maximizes instruction locality and opportunities for SIMD vectorization. The pseudocode below shows the operator-at-a-time approach to k-means.

```
data[N]; dist[N]; min[N];
while (converge()) {
  for (i = 0; i < N; i++)
    dist[i] = distance(data[i]);
  for (i = 0; i < N; i++)
    min[i] = minimum(dist[i]);
  for (i = 0; i < N; i++)
    reassign(min[i]);
  recompute();
}
```



However, the operator-at-a-time strategy requires materialization of intermediate results between each operator, resulting in poor data locality. A tiled approach can perform each operation on a cache-resident subset of the data, thus reducing materialization costs and saving memory bandwidth, but does not achieve the same level of data locality as the pipeline strategy.

[1]MonetDB does not fully compile queries; rather, the system produces assembly-like language (MAL) for execution by a VM.

Hybrid Strategy

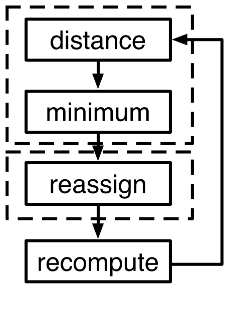
When considering UDF-centric workflows, neither the pipeline nor operator-at-a-time approach is a dominant strategy. Since our Compilation process can introspect UDFs, we propose a hybrid strategy, called operator-centric code generation, that dynamically combines the pipeline and operator-at-a-time approaches based on low-level UDF statistics.

Our strategy first groups all operators into a single pipeline P in order to maximize data locality. Next, for each operator O in P, we leverage the UDF statistics gathered by the UDF Analyzer in order to partition P into a set of vectorizable (i.e., can leverage SIMD) and nonvectorizable (i.e., cannot leverage SIMD) sub-pipelines P'. Intermediate results are materialized between sub-pipelines in cache-resident blocks to reduce the amount of data transferred from memory to the CPU. Note that if the workflow contains no vectorizable UDFs, then the original single-pipeline structure is preserved. By the end of the algorithm, all sub-pipelines should be composed uniformly of either vectorizable or nonvectorizable UDFs.

The only exception to this rule arises when a group of one or more vectorizable UDFs appears at the beginning of a pipeline because of the previously discussed memory bandwidth bottleneck. If the scalar version is already memory-bound, then the vectorizable sub-pipeline should be merged with the adjacent nonvectorizable sub-pipeline in order to benefit from data locality, since no additional performance increase can be achieved with SIMD vectorization.

Consider again the k-means algorithm. Given the statistics provided by the UDF Analyzer (Table 2.2), we notice that the `distance` and `minimum` UDFs are vectorizable because they (1) contain no data-dependent control flow code and (2) have the appropriate apply operator semantics (i.e., they produce a strict *1-to-1* mapping). Therefore, these two UDFs can be split into a separate sub-pipeline, but, since this sub-pipeline resides at the beginning of the workflow, we must also ensure that the computation is not memory-bound. In this case, we see that $Compute\ Time > Load\ Time$, so this sub-pipeline is compute-bound and should therefore be partitioned to yield the following program structure.

```
data[N]; min[N];
while (converge()) {
  for (i = 0; i < N; i++) {
    dist = distance(data[i]);
    min[i] = minimum(dist);
  }
  for (i = 0; i < N; i++)
    reassign(min[i]);
  recompute();
}
```



The pseudocode shown above has the major advantage of being able to vectorize the expensive `distance` and `minimum` UDFs while also minimizing the amount of data materialized between operators. Hence, we propose the following heuristic to summarize our hybrid program structure strategy.

Heuristic 1a *An operator pipeline should always be partitioned into vectorizable and nonvectorizable sub-pipelines, unless the first operator is memory-bound.*

Contribution: The competing pipeline and operator-at-a-time strategies each use a static program structure pattern for compiling traditional SQL queries. However, for complex analytics tasks, UDF characteristics can shift the bottlenecks to favor one of these strategies over the other. For instance, a memory-bound workflow containing many simple UDFs would benefit most from the pipeline approach, whereas the operator-at-a-time approach is better suited for compute-bound workflows with complex UDFs. Often, a combination of these two strategies is optimal, and our approach introspects UDFs to dynamically generate a hybrid program structure that best leverages the underlying hardware.

2.4.2 Aggregation

As described in Section 2.2, a reduce allows users to perform an aggregation in a workflow. Our Optimizer can dynamically generate different code based upon high-level aggregate operator semantics and low-level UDF features. In this section, we present heuristics specific to aggregations.

Group-by

In order to aggregate values grouped by key, reduces normally require a hash table to store keys and associated aggregates. Since hash table lookups contain unpredictable memory accesses, reduce UDFs cannot be vectorized. However, a group-by aggregation is actually comprised of three distinct steps: (1) apply an explicit user-defined key function k; (2) compute the key's hash value using a hash function h; and (3) retrieve/update the associated aggregate value. Since the first two steps have no dependencies, the key/hash functions can be performed in parallel using SIMD vectorization, followed by serial execution of the aggregate value update, as shown below for a sum grouped by key.

```
data[N]; hash[TILE]; sum[M] = {0};
for (i = 0; i < N / TILE; i++) {
  offset = i * TILE;
  for (j = 0; j < TILE; j++) {
    key = k(data[offset + j]);
    hash[j] = h(key);
  }
  for (j = 0; j < TILE; j++)
    sum[hash[j]] += data[offset + j];
}
```

The above pseudocode iterates over the data in cache-sized tiles. The first inner loop applies the key function k and then the hash function h, storing the hash values in a temporary array. The second inner loop performs the hash table lookup using the precomputed hash values and adds the data values to the corresponding sum.

Separating a group-by aggregation into two loops introduces the additional overhead of materializing the computed hash values. In many cases, compute-bound key/hash functions benefit greatly from SIMD vectorization, outweighing this extra cost. However, very simple memory-bound functions will receive no added benefit from SIMD vectorization and should instead be pipelined.

The astute reader may notice that a reduce grouped by key logically consists of two 1-to-1 maps (i.e., the key/hash functions) followed by the aggregation. We can then apply the algorithm from Heuristic 1a in order to determine whether to partition the pipeline. Therefore, we propose the following extension to the original heuristic.

Heuristic 1b *All group-by reduce operations should be decomposed into two 1-to-1 map operations (the key/hash functions) followed by the aggregation and then optimized using Heuristic 1a.*

Contribution: The idea of vectorizing hash computations for group-by aggregations is not new [106, 107, 117]. Other work [106, 108, 111] explores the use of SIMD vectorization for interacting with specialized data structures like Bloom filters. However, our approach can dynamically decide whether SIMD vectorization is beneficial because we allow the user to explicitly provide a key function that our Compilation process can then introspect to determine whether the key/hash functions are compute-bound.

Context Variables

Recall that reduce UDFs can also perform aggregations by updating shared state Context variables. Since the data types and output cardinality (i.e., number of distinct keys) are known a priori, our Optimizer can generate code that uses a form of distinct value encoding at compile time to translate Context variable dictionary lookups into static memory addresses. For example, since the number of centroids is fixed up front in the k-means workflow (Section 2.1), all lookups in the `reassign` UDF can be automatically replaced at compile time with an offset into a one-dimensional array. The pseudocode for the original version is shown below on the left, with the optimized version on the right.

```
//original                           //optimized
assign = t1[ATTR];                   assign = t1[ATTR];
for (i = 0; i < ATTR; i++)           offset = assign * (ATTR + 1);
  c["sum"][assign][i] += t1[i];      for (i = 0; i < ATTR; i++)
c["count"][assign]++;                  c[offset + i] += t1[i];
                                     c[offset + ATTR]++;
```

Notice that in the optimized code, the Context variable lookups `c["sum"]` and `c["count"]` have been automatically replaced with the static memory locations at the specified offsets. Not only do we avoid expensive dictionary lookups every time this UDF is invoked, but we also improve cache line performance by flattening the dictionary to a one-dimensional array. We therefore propose the following heuristic.

Heuristic 2 *All references to Context variables inside UDFs should be replaced with static memory locations at compile time by mapping distinct keys to physical address offsets.*

Contribution: Aggregations grouped by key typically require resizable dictionary structures (e.g., hash tables, binary trees) to handle an arbitrary number of keys. Spark uses a standard hash table to perform `reduce` operations, but the user can achieve better performance in some simple cases with the `aggregate` operator by manually mapping keys to array indices. On the other hand, our API allows users to specify the output cardinality in advance with the Context, and we can leverage this information to generate code that avoids expensive dictionary lookups and automatically handles the array index mapping.

$$\underset{\{(e,r,b)\in(E,R,B)\ |\ \frac{b(1-s)}{d}\leq m\}}{\arg\min} \underbrace{\left(\begin{cases} c_1\left(\frac{1}{2|s-0.5|+\epsilon}\right) & \text{if } e \text{ is } branch \\ c_2\left(\frac{p}{n}\right)+c_3(1-s)a & \text{otherwise}\end{cases}\right)}_{\text{Evaluation Strategy}} + \underbrace{\left(c_4^{\log b}\left\lceil\frac{sd}{b}\right\rceil a + c_5\begin{cases} 0 & \text{if } r \text{ is } max \\ \frac{sd}{b} & \text{if } r \text{ is } exact \\ d & \text{otherwise}\end{cases}\right)}_{\text{Result Strategy}} \tag{2.2}$$

2.4.3 Selection

Optimizing selections is a well-studied problem, but our operator-centric approach goes a step further by pairing code generation techniques with data statistics to get better performance than either a traditional query optimizer or compiler alone. In this section, we present a heuristic for optimizing selections on the code generation level. We separately investigate (1) predicate evaluation and (2) result allocation, and we then propose a cost model that considers several parameters (e.g., number of predicates, estimated selectivities) to dynamically determine the best combination of strategies.

Predicate Evaluation

The initial step in performing a selection is to evaluate whether a particular tuple satisfies the predicate. We first demonstrate existing evaluation strategies and then describe a novel *prepass* strategy. For now, all pseudocode examples assume a sufficiently large result buffer, and we explore efficient result allocation strategies separately in the following section.

Branch: The *branch* strategy is the most straightforward approach. For each input tuple, a conditional statement checks to see whether that tuple satisfies the predicate. If the predicate is satisfied, then the tuple is added to a result buffer; otherwise, the loop skips the tuple and proceeds to the next tuple. The branch strategy is shown below.

```
data[N]; result[M]; pos = 0;
for (i = 0; i < N; i++)
  if (pred(data[i]))
    result[pos++] = data[i];
```

This strategy performs well for both very low and high selectivities, when the CPU can perform effective branch prediction. For intermediate selectivities (i.e., closer to 50%), though, branch misprediction penalties have a severe negative impact on performance.

No-branch: The *no-branch* strategy [116] (sometimes called “predicated” selection [30]) eliminates branch mispredictions by replacing the control dependency with a data dependency. This approach maintains a pointer to the current location in the result buffer that is incremented every time an input tuple satisfies the predicate. If a tuple does not satisfy the predicate, then the pointer is not incremented and the previous value is overwritten. The no-branch strategy is shown below.

```
data[N]; result[M]; pos = 0;
for (i = 0; i < N; i++) {
  result[pos] = data[i];
  pos += pred(data[i]);
}
```

This strategy includes no conditional statements, which yields better performance than the branch strategy for intermediate selectivities by avoiding CPU branch mispredictions.

Prepass: We additionally propose a novel two-phase strategy for selections that improves CPU utilization by performing the predicate test and copy steps independently. Like group-by aggregations, predicate evaluation also logically consists of two distinct steps: (1) testing if a tuple passes the selection criteria (i.e., the branch strategy's `if` conditional and the no-branch strategy's `pos` increment statement); and (2) copying the tuple to the result buffer. Therefore, we can again decompose predicate evaluation into a vectorizable *1-to-1* map followed by a nonvectorizable, data-dependent operation. As shown below, this strategy performs these two steps on cache-sized tiles.

```
data[N]; result[M]; bitmap[TILE]; pos = 0;
for (i = 0; i < N / TILE; i++) {
  offset = i * TILE;
  for (j = 0; j < TILE; j++)
    bitmap[j] = pred(data[offset + j]);
  for (j = 0; j < TILE; j++) {
    result[pos] = data[offset + j];
    pos += bitmap[j];
  }
}
```

The first inner loop performs the predicate test and stores the result in a bitmap, while the second inner loop copies tuples to the result buffer that have passed the selection criteria using either the no-branch (shown above) or the branch strategy. With this technique, predicate evaluation can be partially vectorized because there are no data dependencies in the testing step. Additionally, the resulting code contains tighter loops, thus improving instruction locality.

Result Allocation

Result allocation is particularly difficult for selections, since the output size is not known a priori. We consider three existing strategies and then describe a novel result allocation technique.

Tuple-at-a-time: The most conservative approach to result allocation is to allocate space for only a single output tuple each time an input tuple satisfies the predicate. Tuple-at-a-time allocation minimizes the amount of wasted space, but the overhead associated with allocating in such small increments quickly becomes prohibitive for even relatively small data sizes.

Max: The other extreme would assume a worst-case scenario and allocate all possible necessary space, thereby paying a larger allocation penalty once at the beginning to completely avoid result bounds checking. This approach may work well for very high selectivities but wastes a lot of space for low selectivity cases.

Block: The block allocation strategy is a compromise between the tuple-at-a-time and max strategies. This approach incrementally allocates space for blocks of tuples (e.g., 1024 tuples at a time) in order to balance the required number of allocations and the amount of wasted space.

Exact: All of the previously described allocation strategies make blind decisions regarding result buffer allocation; that is, the max strategy always assumes a selectivity of 100%, while the tuple-at-a-time and block strategies need to decide whether each tuple satisfying the predicate necessitates a new result buffer allocation. Therefore, we can adapt the previously described prepass strategy to also maintain a simple counter when computing the bitmap values, allowing us to generate code that only performs bounds checking if the result buffer could overflow.

Cost Model

We propose a cost model in Equation 2.2 for choosing the optimal combination of evaluation and result strategies given (1) data selectivity, (2) number of predicates, and (3) number of tuple attributes. Each term models the important features of the various strategies, with a summary of notations shown in Table 2.3.

Symbol	Description
e	Evaluation strategy $e \in \{branch, no\text{-}branch\}$
r	Result strategy $r \in \{max, exact, batch\}$
b	Block size $1 \leq b \leq d$
s	Selectivity $0 < s < 1$
m	Wasted memory fraction $0 < m < 1$
d	Number of data elements
p	Number of predicates
a	Number of attributes
n	SIMD parallelism

Table 2.3: Cost model notation.

Every component of the cost model has an associated weight ($c_1, ..., c_5$), which are constants representing the approximate number of cycles for a particular operation (e.g., branch misprediction, comparison, data copy). These constants are architecture-dependent and can be estimated a priori.

In the evaluation strategy component, the term following the c_1 constant (i.e., $\frac{1}{2|s-0.5|+\epsilon}$) considers the cost of CPU branch misprediction using the absolute distance of the data selectivity from 50%. Conversely, since the no-branch strategy cannot perform short-circuit evaluation, the term following the c_2 constant models the cost of evaluating all p predicates, while the term following the c_3 constant considers the cost of extra attribute copying that the no-branch strategy performs for tuples not satisfying the selection criteria.

In the result strategy component, the term containing the c_4 constant expresses the cost of allocating the necessary number of blocks of size b for all attributes a, where the cost of memory allocation scales with b (i.e., allocating a single large piece of memory is less expensive than allocating the same amount of memory in multiple smaller blocks). Lastly, the term following the c_5 constant models the cost of result bounds checking performed by each strategy.

The output of our cost model is a plan (e, r, b) representing the optimal predicate evaluation strategy e, result allocation strategy r, and block size b within the user-specified wasted memory fraction m. We use this cost model to derive our final heuristic.

Heuristic 3 *For all selection operations, choose the combination of evaluation strategy, result strategy, and block size that minimizes the cost for the given parameters (Equation 2.2).*

Contribution: We proposed a cost model that our Optimizer uses to generate different code for selections on a case-by-case basis. Unlike other work [111, 116] that only examines the tradeoffs between the branch and no-branch predicate evaluation strategies, our approach introduces a novel prepass strategy that achieves better performance through SIMD vectorization and tighter loops, and we can perform these optimizations only by combining UDF introspection with traditional DBMS techniques. Moreover, our cost model additionally considers the impact of various result allocation strategies and related optimizations, whereas existing approaches always assume a sufficiently large result buffer. Our cost model can also be easily extended to consider additional operations (e.g., map, reduce) that might follow a selection in a workflow.

2.5 Execution

While the intricacies of distributed workflow execution are important, they are beyond the scope of this work. This section briefly describes our approach to some of the main challenges.

2.5.1 Load Balancing

Our data request model is multitiered and pull-based, allowing for automatic load balancing with minimal overhead. We dedicate a single thread on a single node in the cluster as the *Global Manager* (GM), which is responsible for global decisions such as the coarse-grained partitioning of data across nodes and supervising the current stage of the execution. In addition, we dedicate one thread per node as a *Local Manager* (LM). The LM is responsible for fine-grained management of the local shared memory, as well as transferring data between nodes. The LM also spawns new *Executor* (E) threads for running compiled workflows. These threads request data in small cache-sized blocks from the LM, and each LM in turn requests larger blocks of data from the GM, possibly from remote nodes. All remote data requests occur asynchronously, and blocks are requested in advance to mask network transfer latency.

2.5.2 Memory Management

The LM is responsible for tracking all active T-Sets and performing garbage collection when necessary. UDFs that allocate their own memory, though, are not managed by the LM's garbage collector. TUPLEWARE also avoids unnecessary object creations and data copying (e.g., performing updates in-place if the data is not required in subsequent computations). Additionally, the LM can reorganize and compact the data while idle, potentially even performing on-the-fly compression.

2.5.3 Fault Tolerance

TUPLEWARE further improves performance by forgoing traditional fault tolerance mechanisms for short-lived jobs, where the probability of a failure is low and results are easy to fully recompute. Extremely long-running jobs on the order of hours or days, though, might benefit from intermediate result recoverability. In these cases, TUPLEWARE can perform simple k-safe checkpoint replication. By compiling workflows to distributed programs, TUPLEWARE can optionally generate these checkpointing mechanisms in individual cases based on estimates of the expected runtime and likelihood of a hardware failure.

2.6 Evaluation

This section evaluates the techniques described in this chapter. First, we compare our TUPLEWARE prototype in a distributed setting against Hadoop and Spark and on a single machine against HyPer, MonetDB, and Spark. We then provide a detailed performance breakdown to measure the impact of each optimization from Section 2.4 in realistic scenarios, further isolating their effects in detailed microbenchmarks. We conducted all experiments on Amazon EC2 using either: (1) `c3.8xlarge` instances with Intel E5-2680v2 processors (10 cores, 25MB cache), 60GB RAM, 2 $\times$ 320GB SSDs, and 10 Gigabit*4 Ethernet; or (2) `m3.xlarge` instances with 4 vCPUs, 15GB RAM, and 2 $\times$ 40GB SSDs.

2.6.1 Distributed Benchmarks

Algorithm	10×**c3.8xlarge**									100×**m3.xlarge**								
	Hadoop			**Spark**			TUPLEWARE			**Hadoop**			**Spark**			TUPLEWARE		
	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB
kmeans	1621	5023	36818	4.41	31.3	614	0.451	3.11	29.6	1439	4879	40188	5.89	30.5	274	0.998	4.37	28.9
pagerank	1438	2666	7019	56.6	119	1076	17.1	35.2	102	1456	2623	7290	89.7	183	1719	20.3	31.9	94.2
logreg	1197	1865	6201	2.08	2.45	6.21	0.125	0.431	2.79	1180	1769	6107	3.19	3.71	8.08	0.632	1.34	2.48
bayes	5.18	5.27	6.03	0.532	0.628	0.815	0.047	0.149	0.485	5.29	5.41	5.77	0.603	0.759	1.19	0.184	0.317	0.575

Table 2.4: Distributed benchmark runtimes (seconds).

We compared TUPLEWARE against two prominent distributed analytics frameworks (Hadoop 2.4.0 and Spark 1.1.0) on a (1) small high-end cluster of 10×`c3.8xlarge` instances and (2) large commodity cluster of 100×`m3.xlarge` instances.

Workloads and Data

We implemented a version of four common ML tasks for each system without using any specialized libraries (e.g., Mahout [7], MLlib [9], BLAS [4]), as we wanted to evaluate the performance of the core frameworks. We used synthetic datasets of 1GB, 10GB, and 100GB in order to test across a range of data characteristics (e.g., size, dimensionality, skew). Our results report the total runtime of each algorithm after the input data has been loaded into memory and parsed, with the caches warmed up [2]. For all iterative algorithms, we always perform exactly 20 iterations.

K-means: As described in Section 2.1, k-means is a clustering algorithm that iteratively partitions a dataset into *k* clusters. Our test datasets were generated from four randomly selected centroids.

PageRank: PageRank is an iterative link analysis algorithm that assigns a weighted rank to each page in a web graph to measure its relative significance. Our test dataset was generated with 1 million pages and a normal distribution of links.

[2]Hadoop is disk-based, so the cache cannot be warmed up.

Logistic Regression: Logistic regression aims to find a hyperplane that best separates two classes. Our implementation uses batch gradient descent to classify generated data with 1024 features.

Naive Bayes: A naive Bayes classifier is a conditional model that uses feature independence assumptions to predict class labels. Our generated dataset had 1024 features, each with 10 possible values.

Discussion

As shown in Table 2.4, TUPLEWARE outperforms Hadoop by up to three orders of magnitude and Spark by up to two orders of magnitude for the tested ML tasks. In general, we find that the absolute runtimes are generally lower for the 100-node cluster due to the larger number of cores (400 vCPUs vs. 320 vCPUs) and higher aggregate memory bandwidth than the 10-node cluster, although the distributed coordination costs for more machines impose a larger, constant overhead most noticeable in the small 1GB experiments.

Hadoop incurs substantial I/O overhead from materializing intermediate results to disk between iterations. On the other hand, TUPLEWARE caches intermediate results in memory and performs hardware-level optimizations to improve CPU efficiency. For these reasons, we measure the greatest speedups over Hadoop on the iterative tasks (i.e., k-means, PageRank, logistic regression), whereas the performance gap for naive Bayes is much smaller.

Spark also outperforms Hadoop for iterative tasks by keeping the data memory-resident. Additionally, Spark offers a richer API that allows the runtime to pipeline operators, improving data locality. However, TUPLEWARE achieves additional speedups over Spark by compiling workflows into distributed programs (Section 2.3) and employing low-level code generation optimizations (Section 2.4). These optimizations are most beneficial in CPU-intensive tasks (e.g., k-means) because they allow TUPLEWARE to more efficiently use the available hardware resources. For instance, Spark shows particularly poor performance in the 100GB k-means case because the default internal data representation exceeds the aggregate 600GB memory. Other tasks (e.g., logistic regression, naive Bayes) operate close to the memory bandwidth limit, but TUPLEWARE's code generation techniques can still show improvements over Spark's approach.

Finally, we noticed that the more network-bound tasks (e.g., PageRank) show absolute runtimes for both Spark and TUPLEWARE that tend to increase sublinearly compared to data size. We observed this effect because the PageRank workloads used an increasing number of page links but a fixed number of distinct pages (i.e., 1 million), and the ranks for those pages needed to be redistributed to all workers in the cluster on every iteration. For algorithms like PageRank, TUPLEWARE's Context variables are highly efficient for representing an ML model that needs to be iteratively redistributed.

2.6.2 Single Node Benchmarks

We also compared TUPLEWARE on a single `c3.8xlarge` instance to a DBMS that uses query compilation (HyPer), a column-store (MonetDB 5), and Spark. As mentioned in Section 2.4, HyPer compiles SQL queries using the pipeline strategy, whereas MonetDB implements the operator-at-a-time strategy.

Algorithm	1×`c3.8xlarge`			
	Spark	**HyPer**	**MonetDB**	TUPLEWARE
kmeans	6.34	2440	8639	0.615
pagerank	212	1220	272	19.5
logreg	1.96	118	153	0.259
bayes	0.107	6.34	2.11	0.042
tpch1	3.29	0.127	1.71	0.341
tpch4	9.69	0.388	0.382	1.42
tpch6	0.971	0.048	0.128	0.105

Table 2.5: Single node benchmark runtimes (seconds).

Workloads and Data

In addition to the four previously described ML tasks, we also included three TPC-H queries (Q1, Q4, Q6). Since the scale is smaller, we wanted to evaluate the performance of the ML tasks using real-world datasets. For the DBMSs, we implemented the ML algorithms in SQL without UDFs, and all reported runtimes exclude compilation time.

UK Crime: We ran k-means on a 240MB dataset [15] containing GPS coordinates of crimes in the UK over the past five years.

Wikipedia Web Graph: We ran PageRank on a randomly sampled 1GB subset of Wikipedia's complete dump of articles, containing about 6 million pages and 130 million links [16].

Million Song Dataset: We used a randomly sampled 1GB subset of the Million Song Dataset [27] containing 90 audio features and the release year for each song. Logistic regression and naive Bayes were used to predict each song's release year.

TPC-H: TPC-H is a popular OLAP benchmark that contains BI queries. Even though TUPLEWARE's focus is optimizing for UDF-centric workflows (e.g., ML), we wanted to show that some of the optimization techniques we developed also apply to more traditional SQL analytics queries. We implemented three TPC-H queries (Q1, Q4, Q6) in TUPLEWARE and compared the performance to HyPer, MonetDB, and SparkSQL using a scale factor of 10. We selected these three queries because they do not focus on join optimizations, which we have not explored in this chapter.

Discussion

As shown in Table 2.5, TUPLEWARE outperforms Spark, HyPer, and MonetDB for all ML tasks. In particular, HyPer and MonetDB both perform poorly on the ML workloads because they are not designed to express or optimize complex, iterative UDF workflows. On the other hand, the DBMSs can execute the TPC-H queries very efficiently by applying well-known OLAP optimization techniques, including indexing, sorting, and columnar compression, all of which TUPLEWARE does not currently implement. However, by tuning the level of parallelism (i.e., number of threads) to optimally saturate memory bandwidth, TUPLEWARE can achieve better performance than both HyPer and MonetDB on Q6, which is a simple scalar aggregation, but we chose instead to match Spark's level of parallelism in order to ensure a fair comparison for the performance breakdown in the following section.

Spark similarly outperforms the DBMSs for the ML tasks, but SparkSQL is slower than all other systems for the TPC-H queries. Although Spark can handle UDF workflows better than the DBMSs, our previously described code generation techniques and optimizations enable TUPLEWARE to achieve about an order-of-magnitude speedup over Spark for all workloads. We further explore the source of these speedups in the following section.

2.6.3 Performance Breakdown

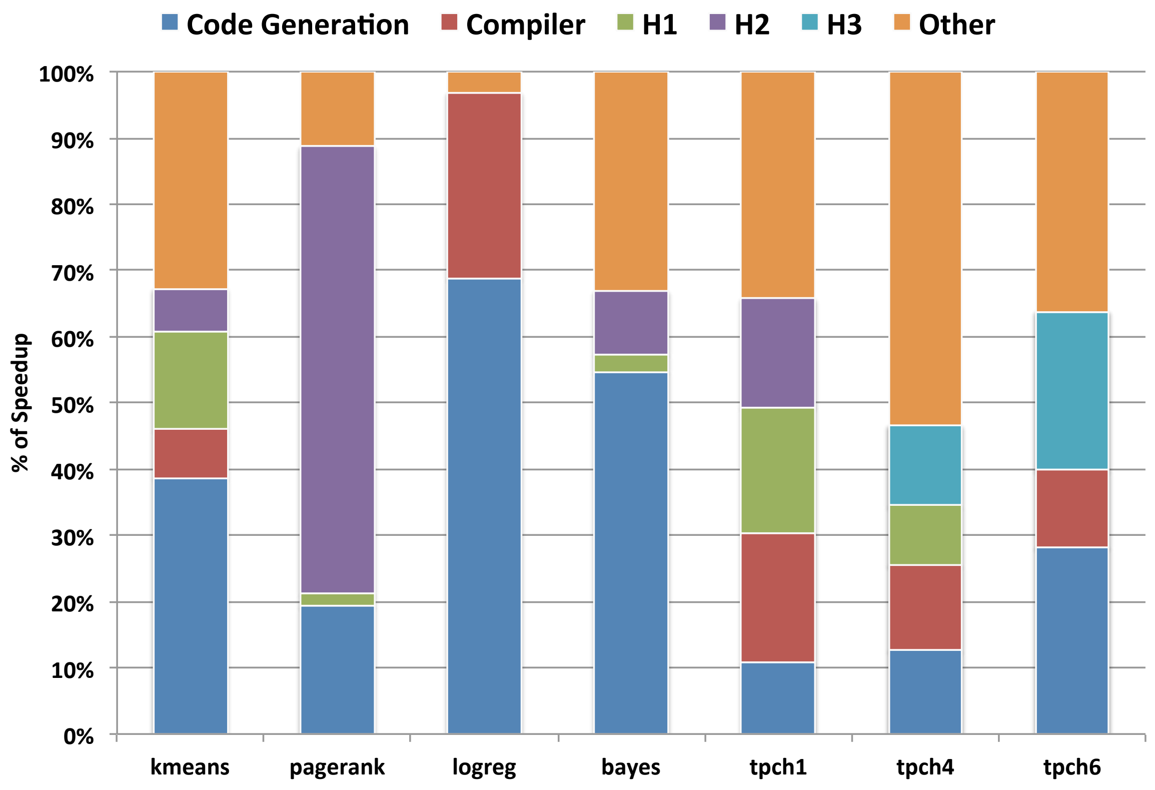


Figure 2.3: A performance breakdown with percentage speedups achieved by TUPLEWARE over Spark from the single node benchmarks.

Since Spark is the closest in spirit to TUPLEWARE, we provide a detailed breakdown that highlights the impact of different components on overall workflow runtime. We conducted all breakdown experiments on a single `c3.8xlarge` instance to exclude factors that impact distributed performance (e.g., network object serialization). Figure 2.3 shows the isolated percentage contribution of each component to the total speedup of TUPLEWARE over Spark for the single node benchmarks from the previous section shown in Table 2.5. To derive the absolute time saved by a particular optimization, one can take the percentage from Figure 2.3 and multiply this value by Spark's runtime in Table 2.5 (e.g., Heuristic 1 saves approximately $15\% \times 6.34s = 0.951s$).

Code Generation

In contrast to Spark's JVM-based implementation that uses polymorphic iterators, TUPLEWARE directly generates LLVM code for workflow execution. Code generation provides substantial speedups for compute-intensive UDF workflows by avoiding many sources of runtime overhead associated with high-level abstractions, including dynamic dispatch of function calls, object creation penalties, and unnecessary loop bounds checking. As shown in Figure 2.3, the performance impact of code generation is most noticeable in workflows containing tight loops or complex instructions (e.g., `sqrt` in k-means). These speedups represent the baseline performance improvements that can be achieved by applying our techniques to generate code that explicitly manages memory, resolves polymorphic function calls at compile time, and performs type specialization, among other advantages. We measured each of these components by comparing against a baseline of generated code to isolate the various inefficiencies introduced by high-level abstractions like Java and iterators.

Compiler

By generating code, TUPLEWARE can also leverage a wide variety of modern compiler optimizations "for free," including inline expansion and SIMD vectorization. Inline expansion particularly benefits workflows comprised of many UDFs (e.g., k-means, all TPC-H queries) by eliminating extra instructions associated with external function calls and minimizing register spilling. On the other hand, SIMD vectorization can improve the performance of UDFs that use vectorizable loops internally. For example, the compiler can automatically vectorize the `for` loop used to compute the dot product for each data element in logistic regression, yielding a substantial performance improvement for this task. We evaluated the impact of this component by comparing runtime performance of the generated code to a baseline that used compiler flags to disable individual optimizations.

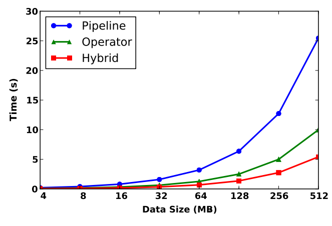
Operator-centric Optimizations

All of the heuristics described in Section 2.4 combine high-level semantic information about the workflow with low-level UDF statistics gathered by introspecting LLVM IR in order to further improve the performance of the generated code. As shown in Figure 2.3, these optimizations offer additional speedups targeted to characteristics of individual workloads. For example, Heuristic 1 (H1) selects a hybrid program structure to alleviate the CPU bottlenecks in several workflows, which substantially outperforms Spark's static pipeline strategy in these cases. Similarly, the Context variable optimizations from Heuristic 2 (H2) help the most in workflows with aggregate values that are frequently updated in a random order. Finally, for workflows containing selections (i.e., the TPC-H queries), Heuristic 3 (H3) allows TUPLEWARE to generate more efficient code by also considering data selectivities. We evaluated each individual heuristic by comparing the optimized generated code to a baseline version that does not apply that heuristic.

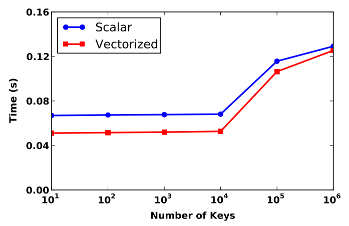
Other

The remaining speedups in each workload can be attributed to various factors, including scheduling overhead in Spark, garbage collection pauses, and intangible engineering differences. In some workloads (e.g., PageRank, logistic regression), the observed speedups are explained almost entirely by the application of our workflow compilation techniques (Section 2.3). However, the performance improvements in other workloads are much more difficult to quantify. In particular, TUPLEWARE's speedup over SparkSQL in the TPC-H queries is less straightforward, since inefficient data structure implementations or suboptimal plan selection for complex SQL queries can have a significant performance impact.

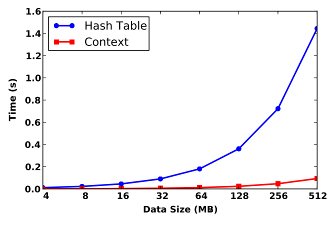
2.6.4 Microbenchmarks



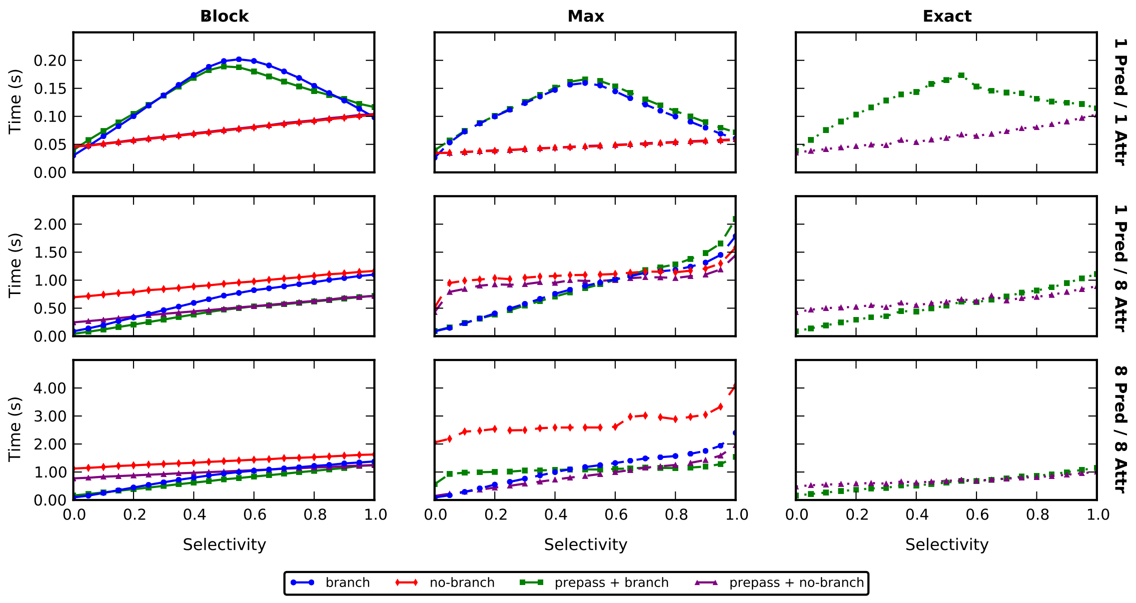
(a) Heuristic 1a



(b) Heuristic 1b



(c) Heuristic 2



(d) Heuristic 3

Figure 2.4: Heuristic microbenchmarks.

As previously described, our operator-centric code generation strategy (Section 2.4) uses information about the workflow, data, and UDFs in order to generate better code. In this section, we use a series of microbenchmarks to study the benefits of each of our proposed heuristics. All microbenchmarks were implemented in C++, compiled with Clang 3.4, and run on a single `c3.8xlarge` instance.

Heuristic 1a

We compared our hybrid strategy to the pipeline and operator-at-a-time strategies using the previously described k-means task. We tested each strategy with varying data sizes and show the results in Figure 2.4a. The pipeline strategy provides excellent data locality but prohibits any SIMD vectorization due to the fact that the `reassign` UDF cannot be vectorized. Conversely, the operator-at-a-time strategy benefits greatly from bulk processing but fails to consider data locality. Using this approach, the `distance` and `minimum` UDFs can be vectorized separately, but materializing intermediate results between each operator incurs significant overhead. Our hybrid strategy outperforms both existing strategies by taking advantage of SIMD vectorization when possible while also pipelining consecutive vectorizable operations for better data locality, achieving a 2-$5\times$ speedup.

Heuristic 1b

We compared standard scalar hashing to our vectorized hashing approach by performing a sum grouped by key on 512MB of data. We varied the number of distinct, uniformly distributed keys and used a simple hash function ($\text{mod}10$). Figure 2.4b shows a 20% performance increase in this simple case for vectorized hashing. However, using a more complex hash function would achieve greater speedups with SIMD vectorization.

Heuristic 2

We compared our Context variable implementation to a standard hash table in order to compute a count grouped by 10 distinct keys with varying data sizes. Figure 2.4c shows that Context variables can improve performance by as much as $16\times$.

Heuristic 3

In order to evaluate our cost model, we conducted an extensive series of microbenchmarks that test all strategies across a range of three parameters: (1) minimum predicate selectivity, (2) total number of predicates, and (3) number of tuple attributes. We show the results of these experiments in Figure 4d, which we limit to only the most salient cases. Each row of charts represents different numbers of predicates and tuple attributes, and each column corresponds to a result allocation strategy. For each chart, we show all possible predicate evaluation strategies. We measure the total runtime on the y-axis for the varying selectivities shown on the x-axis. In all tested cases, our cost model correctly chose the strategy combination with the lowest runtime.

Chapter 3

Query-centric Code Generation

In the previous chapter, we discussed two different approaches to code generation: (1) data-centric and (2) operator-centric. However, both of these categories share one key commonality, namely that they optimize and generate code by reasoning at the level of traditional query operators. On one hand, the data-centric approach optimizes across operator boundaries by merging distinct query operators into combined statements to maximize data locality. On the other hand, operator-centric strategies group query operators together to better enable low-level optimizations (e.g., SIMD, prefetching).

Unfortunately, by utilizing this traditional abstraction, existing approaches place an arbitrary restriction on the variety of code that they can produce, and generated queries end up resembling amalgamations of clearly delineated operator templates. In reality, though, systems that use code generation do not actually need to adhere to this abstraction, since they have complete freedom to create ad hoc operators as necessary.

Therefore, in this chapter, we propose a new *query-centric* code generation strategy, called BESPOKE. Unlike existing strategies that optimize for and generate code at the level of traditional query operators, BESPOKE optimizes across traditional operator boundaries to produce code optimized at the level of the entire query. In particular, we apply the concept of "predicate pullups" in order to produce more streamlined code that outweighs the penalties of performing wasted work.

The remainder of this chapter is organized as follows. Section 3.1 presents BESPOKE, our query-centric code generation strategy, and describes the key techniques we developed for this approach. Then, we provide an extensive evaluation of all of these techniques in Section 3.2.

3.1 Bespoke

The "predicate pushdown" has long existed as a fundamental query optimization heuristic. The key idea of this technique is to filter out tuples as early as possible during query processing to avoid having later operators perform work that is ultimately discarded. In traditional disk-based systems, predicate pushdowns are an effective way to reduce the amount of data that must be retrieved from disk, as well as to limit the amount of processing performed by expensive operators (e.g., joins).

However, for in-memory processing, the relatively simple relational comparisons (i.e., ==, !=, >, >=, <, <=) that normally occur in selection predicates coupled with aggressive columnar compression calls the traditional wisdom of the predicate pushdown into question. If the CPU has free cycles in many cases, then, might a "predicate pullup" actually be better?

Predicate pullups have been previously proposed, including for delaying evaluation of expensive predicates [66] and sharing work among continuous queries [38]. We extend this work and explore the application of the predicate pullup technique to code generation. In the following, we discuss these extensions for (1) scalar aggregation, (2) group-by aggregation, and (3) join.

3.1.1 Scalar Aggregation

The simplest form of aggregation is scalar aggregation, where the query has no group-by clause and outputs a single aggregate value. Figure 3.1a shows TPC-H Q6 from Section 1.1.1, which calculates the sum of `l_extendedprice * l_discount` after applying a predicate over the relation.

Again, the basic data-centric version of the generated code that performs a predicate pushdown for this query is outlined in Figure 3.1b. As shown, this version evaluates the predicate before adding the product of `l_extendedprice * l_discount` to the running sum, eliminating any wasted work for tuples that do not satisfy the predicate.

Value Masking

On the other hand, our proposed predicate pushup version is shown in Figure 3.1c. In this version, the single loop of the basic vectorized query processing approach is broken into a separate inner and outer loop. The outer loop iterates over the input vector eight tuples at a time, while the inner loop stores the result of each application of the predicate in the temporary `cmp` output mask. This is particularly efficient because the selection can benefit from SIMD due to the lack of both control and data dependencies in the generated code.

Since predicate pullups perform wasted work (i.e., subsequent operators still process tuples that do not pass the predicate), predicate selectivity plays an important role. More specifically, if the query contains one or more highly selective predicates, the `cmp` bitmask in Figure 3.1c will contain mostly zeros, and the follow up operator will perform a significant amount of wasted work for the tuples that fail the predicate. Therefore, in such cases, it is valuable instead to switch to an offset-based approach that fetches only the tuples in the bitmap that have a one instead of fetching all tuples and adding the result of the predicate to the mask.

Indexed Access

Based on this tradeoff, we propose a fall-back that directly considers and attempts to eliminate the data copying from the operator-centric approach. Instead of copying the values to intermediate result vectors between operators, we access them by their indexes, similar to the MonetDB/X100 approach [30]. While this leads to non-uniform accesses, it is more economical than the traditional data copying approach taken by operator-centric code generation strategies (e.g., Tupleware, ROF).

```
select sum(l_extendedprice * l_discount)
from lineitem
where l_shipdate >= '1994-01-01'
  and l_shipdate < '1995-01-01'
  and l_discount between 0.05 and 0.07
  and l_quantity < 24
```

(a) TPC-H Q6

```
size_t i; sum = 0.0;
for (i = 0; i < l_len; i++)
  if (l_shipdate[i] >= '1994-01-01'
      && l_shipdate[i] < '1995-01-01'
      && l_discount[i] >= 0.05
      && l_discount[i] <= 0.07
      && l_quantity[i] < 24)
    sum += l_extendedprice[i] * l_discount[i];
```

(b) Pushdown

```
size_t i, j, len; cmp[]; sum = 0.0;
for (i = 0; i < l_len; i += TILE) {
  len = l_len - i < TILE ? l_len - i : TILE;
  for (j = 0; j < len; j++)
    cmp[j] = l_shipdate[i+j] >= '1994-01-01'
           & l_shipdate[i+j] < '1995-01-01'
           & l_discount[i+j] >= 0.05
           & l_discount[i+j] <= 0.07
           & l_quantity[i+j] < 24;
  for (j = 0; j < len; j++)
    sum += (l_extendedprice[i+j] * l_discount[i+j])
         * cmp[j];
}
```

(c) Pushup

```
size_t i, j, len, pos; cmps[]; idxs[]; sum = 0.0;
for (i = 0; i < l_len; i += TILE) {
  len = l_len - i < TILE ? l_len - i : TILE;
  for (j = 0; j < len; j++)
    cmp[j] = l_shipdate[i+j] >= '1994-01-01'
           & l_shipdate[i+j] < '1995-01-01'
           & l_discount[i+j] >= 0.05
           & l_discount[i+j] <= 0.07
           & l_quantity[i+j] < 24;
  pos = 0;
  for (j = 0; j < len; j++) {
    idx[pos] = j;
    pos += cmp[j];
  }
  for (j = 0; j < pos; j++)
    sum += l_extendedprice[i+idx[j]]
         * l_discount[i+idx[j]];
}
```

(d) Indexed Access

Figure 3.1: Scalar Aggregation

```
size_t i, j, len; cmps[]; tmps[]; sum = 0.0;
for (i = 0; i < l_len; i += TILE) {
  len = l_len - i < TILE ? l_len - i : TILE;
  for (j = 0; j < len; j++)
    cmp[j] = l_shipdate[i+j] >= '1994-01-01'
           & l_shipdate[i+j] < '1995-01-01'
           & l_quantity[i+j] < 24;
  for (j = 0; j < len; j++)
    tmp[j] = l_discount[i+j]
           * (l_discount[i+j] >= 0.05
            & l_discount[i+j] <= 0.07);
  for (j = 0; j < len; j++)
    sum += l_extendedprice[i+j] * tmp[j] * cmp[j];
}
```

(e) Access Consolidation

Figure 3.1: Scalar Aggregation (cont.)

In order to decide between the value masking and the indexed approach to accessing data items, we use a cost model similar to the one previously described in Section 2.4. Based on the data and operations in the query, we first determine if the computation is compute-bound or memory-bound. If the operation is compute-bound, it is beneficial to use an index-based approach to pass data items between operators since it reduces any unnecessary work. On the other hand, if the computation is memory-bound, the extra work is masked by the data transfer and predicate pullups are beneficial. In Section 3.2, we show cases where each of these approaches are beneficial.

Access Consolidation

OLAP queries frequently reference the same attributes in multiple places in a query. One notable example is TPC-H Q1, which has the most computationally expensive `SELECT` clause of any TPC-H query.

Common Subexpression Elimination (CSE) is a common compiler technique to remove redundant computations by materializing intermediate values in temporary variables. CSE is straightforward when optimizing a particular component of the query (e.g., the select clause), but can we go a step further by removing redundant data accesses across the entire query?

Again, in TPC-H Q6, notice that `l_discount` appears in both the `SELECT` and `WHERE` clauses of the query. Based on this observation, we propose a new rewrite, called Access Consolidation, that removes redundant data accesses across operator boundaries by directly merging together operators that work on the same data. More specifically, in this query, the generated code shown in Figure 3.1e eliminates the need to access the `l_discount` attribute in two separate parts of the query (i.e., the `SELECT` and `WHERE` clauses). Instead, this query only accesses the attribute once, reducing the overhead associated with unnecessary load/store instructions and increased memory pressure.

3.1.2 Group-By Aggregation

Unlike scalar aggregation, a group-by aggregation produces multiple aggregate values that are grouped by the attributes listed in the query's `GROUP BY` clause.

```
select l_returnflag, sum(l_quantity)
from lineitem
where l_shipdate < '1998-09-02'
group by l_returnflag
```

(a) TPC-H Q1

```
size_t i, j, len; cmp[]; sums = {0};
for (i = 0; i < l_len; i += TILE) {
   len = l_len - i < TILE ? l_len - i : TILE;
   for (j = 0; j < len; j++)
      cmp[j] = l_shipdate[i+j] < '1998-09-02';
   for (j = 0; j < len; j++) {
      key = l_returnflag[i+j];
      sums[key] += l_quantity[i+j] * cmp[j];
   }
}
```

(b) Mask the Value

```
size_t i, j, len; key[]; counts = {0};
for (i = 0; i < l_len; i += TILE) {
   len = l_len - i < TILE ? l_len - i : TILE;
   for (j = 0; j < len; j++) {
      cmp = l_shipdate[i+j] < '1998-09-02';
      key[j] = l_returnflag[i+j] * cmp + cmp;
   }
   for (j = 0; j < len; j++)
      sums[key[j]] += l_quantity[i+j];
}
```

(c) Mask the Bucket

Figure 3.2: Group-by Aggregation

The traditional hash-based aggregation first applies the query predicates using branching. Then, the system looks up the bucket for the group by key for any tuples that satisfy the predicate and finally updates the corresponding aggregate value. Although straightforward, this approach suffers from the same downfalls as scalar aggregation (i.e., contains data and control dependencies) and therefore precludes SIMD vectorization. In the following, we propose two novel branch-free implementations of a group-by operator that carefully balance the predicate evaluation, group-by lookup, and aggregate computation.

Value Masking

The first branch-free variant of group-by aggregation involves masking the value to be combined with a particular group-by aggregate. More specifically, for every data item, this approach first unconditionally retrieves the current aggregate value from the hash table. Then, similar to the previously described value masking technique for scalar aggregation, the returned value is updated by performing the aggregation and multiplying by the predicate (zero or one).

As an example, consider the simplified version of TPC-H Q1 shown in Figure 3.2a and the resulting code for the branch-free variant that masks the hash bucket value in Figure 3.2b. In this case, the predicate is first evaluated and stored in the `cmp` bitmap. The code then increments the sum for a specific `l_returnflag` with `l_quantity` multiplied by 1 (i.e., true) if the tuple passed the predicate and 0 (i.e., false) otherwise.

Bucket Masking

Another approach to generating code for a branch-free group-by aggregation can alternatively mask the aggregation bucket instead of the value. The core idea is that our approach maps any keys that do not satisfy the predicate into a single bucket, similar to Voodoo's predicated lookup technique [105].

Figure 3.2c shows how our approach calculates the bucket to update after evaluating the predicate and storing the result in the `cmp` variable. After calculating the bucket, the generated code performs the aggregation on the value for the calculated bucket. Any key that does not pass the predicate (i.e., `cmp` is equal to 0) still updates the returned bucket, but it is important to note that all keys which fail the predicate map to the same bucket, which is then discarded.

This approach is more efficient because, oftentimes, the computations performed in the `SELECT` clause can be overlapped with the hash table lookup. However, a similar problem to value masking occurs when a query contains very complex project clauses (e.g., TPC-H Q1) which result in a large amount of wasted work for a tuple that does not satisfy the predicate. To remedy this problem, we can again apply the same optimization in the previously described scalar aggregation case, whereby we can fall back to an offset-based aggregation approach, thereby avoiding the problem of reading in a lot of values only to discard them.

3.1.3 Join

Similar to group-by aggregations, joins are one of the most widely used operators in query processing. This section describes predicate pullup optimizations for the well-known hash join algorithm. Although many

```
int hmap_contains(hmap, key, cmp)
{
   key = mask_key(key, cmp);
   hash = hashfn(key);
   k = hmap->ents[hash].key;
   return k == key - (k != EMPTY) & (k != key);
}
```

(a) hmap contains

```
int bmap_contains(bmap, key, cmp)
{
   return (bmap->bits[key/8] >> (key%8)) & cmp;
}
```

(b) bmap contains

Figure 3.3: Joins

complicated join algorithms have been proposed, simple hash joins are widely used and offer excellent performance in many cases [77].

Fast Lookups

A similar technique of masking the output bucket also applies to operations, shown in Figure 3.3a. The core idea behind this optimization is that instead of masking an aggregation bucket, we mask hash table lookups for missing keys to a bucket that will later be discarded, similar to previously described technique for bucket masking in group-by aggregations.

However, a key source of inefficiency in hash tables is chaining, since it requires conditional branching statements as well as pointer following. Therefore, the hash tables that our generated code uses employ several optimizations to ensure that common patterns are as fast and efficient as possible.

For example, Figure 3.3a shows the `hmap_contains` function that the generated code uses in order to check if an element is contained in a hash table. First, the function masks the bucket for the given key, using the same techniques as group-by aggregations (i.e., masking the `key` value with a computed `cmp` value from the query predicate). Then, once the correct bucket is found, the generated code uses as series of comparisons and masks to detect whether the key is contained at the head of the chain in a branch-free way. More specifically, the function returns 0 if the key is not equal (i.e., the bucket is empty), 1 if the key is equal (i.e., the key is at the head of the chain), or -1 if neither is true, such that the key might be somewhere in the chain.

Therefore, importantly, the resulting code does not contain any branches in the two most common cases (i.e., the hash bucket is empty or key is equal to the value at the head of the chain). In the unlikely event that any tuple in the block requires traversing the chain, the generated code separately processes the block using conditional statements.

Simple set inclusion techniques will be much more efficient in this case, since it eliminates the need to traverse hash bucket chains to find elements that are not present. For example, HyPer uses a compact tag similar to a Bloom filter to avoid having to scan hash bucket chains, since it can use the tag to first determine if an element might be present somewhere in the chain [85]. However, the key idea of our approach is that we can completely remove branching in the common case (i.e., the bucket is empty or the key is located at the head of the chain), which significantly improves performance.

Bit Vectors

Although the previously described join technique will work for any type of join, for the common case of primary/foreign key joins, we can apply a more efficient approach based on an old technique [26] that was recently rediscovered [102]. This join algorithm has three distinct steps, which we describe in the following.

The first step takes place at either data loading or schema definition time before any query processing has even begun. During this time, the DBMS must enforce both primary and foreign key constraints.

A primary key constraint requires that a key must uniquely identify a tuple, which means that no two tuples in a relation may share the same primary key. As such, we can internally (and transparently) replace references to primary keys with the physical offsets of the memory addresses of tuples, which also uniquely identify each tuple (i.e., each tuple is identified by the offset in the table where it is stored), similar to dictionary encoding.

On the other hand, foreign key constraints require that a key value in a foreign key column needs to exist in the primary key column of another table. DBMSs typically perform this check when loading a tuple (or when scanning an existing foreign key column) by probing an index on the primary key of the referenced table in order to see if the value exists. In addition to this check for existence, our algorithm (transparently) converts the foreign key value to the physical offset value that uniquely identifies each tuple on the primary key side. Again, these actions occur during primary and foreign key constraint checking time and therefore do not incur any additional overhead, since these checks need to be performed regardless.

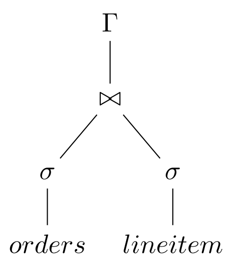
The next step of our approach involves filtering down to only the selected tuples for the relation in what would be considered the “build” phase of a traditional hash join algorithm. Essentially, we create a mask bit vector, just as in our other predicate pullup techniques, on the smaller side of the join by applying a selection predicate, where the value of a tuple is 1 if the tuple belongs in the join and 0 otherwise. The initial reaction to this approach might be that, during the subsequent “probe” phase when the the bit vector is used, the same cache miss problems that occur in a traditional hash join will come up. Surprisingly, though, even for relatively large tables, the corresponding selection bit vector is still a very manageable size. For example, a table with 100M tuples requires only about 12.5MB of space, which is only half the size of most modern L3 CPU caches. Furthermore, compression is another option for reducing size by run-length encoding long sequences of 0’s (or 1’s) in the selection bit vector.

The final step of the join algorithm is to actually perform the join by testing whether or not a particular foreign key value appears in the corresponding mask array that contains the filtered set of primary keys, which proceeds as follows. First, for every tuple in the probe (i.e., larger) side of the join, we check the corresponding index in the join bit vector based on the offset of the join key. As shown in Figure 3.3b, this test returns a 1 if the key is included and a 0 otherwise. After using bit manipulation operations to extract the value, the result is stored back into an output bit vector that could either be (1) a bit vector intended to be passed locally to the next operator or (2) a larger bit vector for the entire table to be used in a subsequent join.

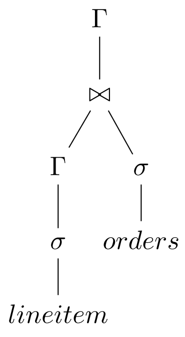
Again, although space might seem like a concern, especially for queries with large numbers of joins, the resulting bit vectors are relatively compact, and the space requirement in bytes never grows larger than the size

```
select o_shippriority, count(*)
from orders, lineitem
where l_orderkey = o_orderkey
  and o_orderdate < '1995-03-15'
  and l_shipdate > '1995-03-15'
group by o_shippriority
```

(a) TPC-H Q3



(b) Original Plan



(c) Rewritten Plan

Figure 3.4: Join Pullups

of the largest table divided by eight (i.e., the number of keys that can be represented by a single byte). For example, TPC-H Q5 contains a six-way join. First, after a pass over the "probe" side of the join is complete (`orders` in this example), the mask array constructed for the "build" side of the join can be safely freed if it is no longer needed, making the memory again available. On the other hand, if the mask array is required for later processing or another join, or if the system wishes to perform some kind of intelligent caching for intermediate results, they can be paged out to main memory to make more space available in the CPU cache until they are needed again.

Furthermore, this algorithm is much easier to parallelize than complex parallel algorithms that use radix partitioning. Specifically, rather than having to radix partition both sides of the join, they can simply be range partitioned, with the optional additional optimization of radix partitioning the "probe" side in cases when the bit vector exceeds the size of the CPU cache in order to remain cache-conscious.

Join Pullups

A join between two tables is essentially a selection that returns all tuples on the probe side that have a match on the build side. Traditionally, an equijoin first builds a hash table over one of the tables using the primary key as the key and the query's group-by key as the value. Then, for every tuple in the probe side, the algorithm retrieves the value for the foreign key and performs a group-by over that retrieved value.

However, the main problem with this approach is that it requires a materialization of the group-by keys on the build join in order to perform the final group-by on the probe side. Therefore, since a join is essentially a selection, we propose instead to perform the same pullup technique.

More specifically, our proposed approach first performs the query's aggregate for each foreign key over the build side of the join. Then, for each tuple on the probe side that passes any predicates, our approach looks up the aggregated value for the tuple's primary key and performs the final aggregation for the query's group by key.

As an example, consider the simplified version of TPC-H Q3 and the accompanying query plans in Figure 3.4. In the original plan (Figure 3.4b), the group-by key `o_shippriority` needs to be materialized on the build side of the join, and then it is accessed by the probe side for the final aggregation. On the other hand, the optimized plan (Figure 3.4c) switches the join order, performing a group-by on the build side by the join key before the final aggregation takes place on the probe side.

3.2 Evaluation

This section presents a detailed experimental evaluation of the techniques described throughout this chapter. To test the efficacy of our approach, we compare each of the techniques that we presented against baselines of both data- and operator-centric code generation. We use a series of detailed microbenchmarks designed to specifically isolate the different operators described throughout this chapter.

For fairness, all strategies are hand-implemented and use the same library code (e.g., hash table implementations) to ensure a fair, apples-to-apples comparison. This type of comparison is meant to eliminate any of the overheads associated with system implementation differences or additional features (e.g., concurrency control, recovery logging) that might be expected in a full-fledged DBMS.

Our hand-coded solutions use some common optimizations that have been proven to have a huge impact for column stores (e.g., dictionary encoding, null suppression, fixed point arithmetic). For tiled approaches, we use an intermediate vector size of 1024, as suggested by other studies [77, 90]. Additionally, our code does not use any indexing or materialized data structures for query processing other than primary/foreign key indexes; we only build data structures on the fly as needed during query processing.

All experiments were conducted on a single server with an Intel E5-2660 CPU (2.2GHz, 10 cores, 25MB cache) and 256GB RAM.

3.2.1 Microbenchmarks

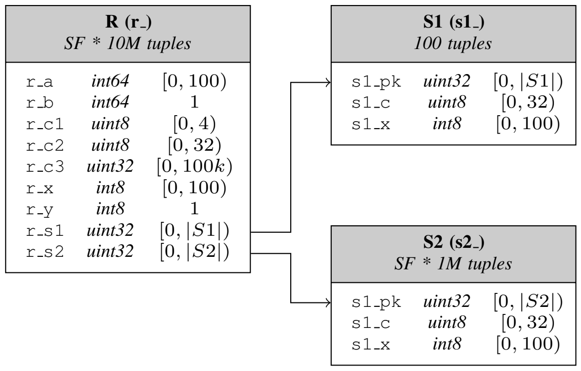
To evaluate the efficacy of our techniques, we created a microbenchmark that isolates each of the operations described in the previous section. Figure 3.5 shows the full microbenchmark specification, including the table schemas (3.5a) and four queries (3.5b).

In the following, we provide a detailed description of the microbenchmark results. Although many variations of the benchmark queries are possible, we limit each query to two configurations that test each of the extremes of the relevant operations.

Q1

This query (Figure 3.6a) tests simple scalar aggregation over different columns.

In three of the four query flights, data-centric (DC) code generation exhibits the well-known curve due to CPU branch misprediction for intermediate selectivities [116]. Flight Q1.2 replaces the multiplication



(a) Schema

Query #	SQL	Args

Q1	`select sum(COL1 OP COL2)`
	`from R`
	`where r_x < SEL and r_y = 1`
		OP, COL1, COL2, SEL

Q2	`select C, sum(r_a * r_b)`
	`from R`
	`where r_x < SEL and r_y = 1`
	`group by C`
		OP, C, SEL

Q3	`select sum(r_a * r_b)`
	`from R, S`
	`where r_S = S_pk and PRED`
		S, PRED

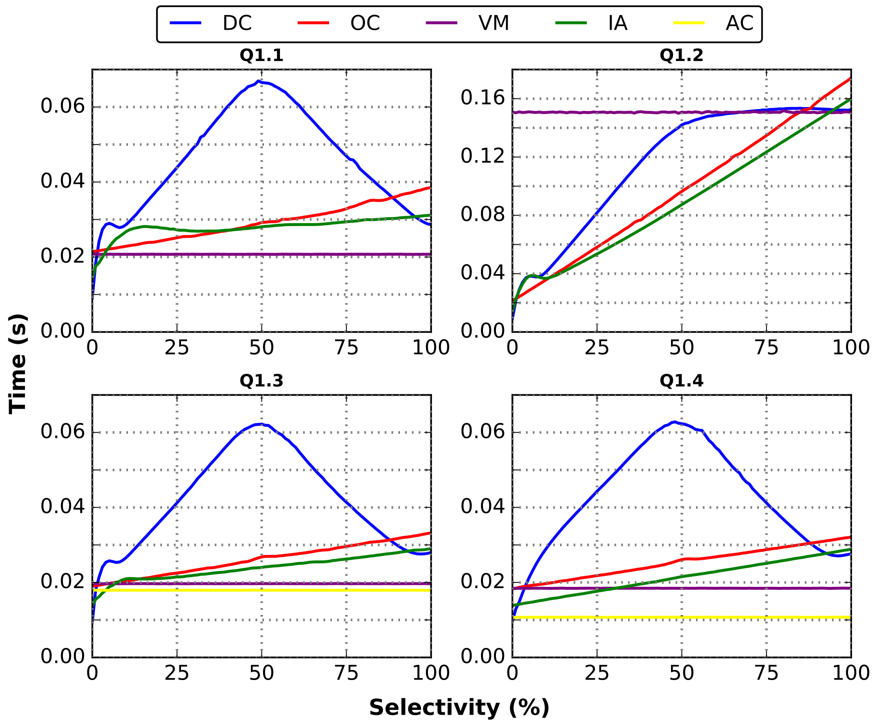
Q4	`select C, sum(r_a * r_b)`
	`from R, S`
	`where r_S = S_pk and r_x < 70`
	`  and S_x < SEL`
	`group by C`
		S, C, SEL

(b) Queries

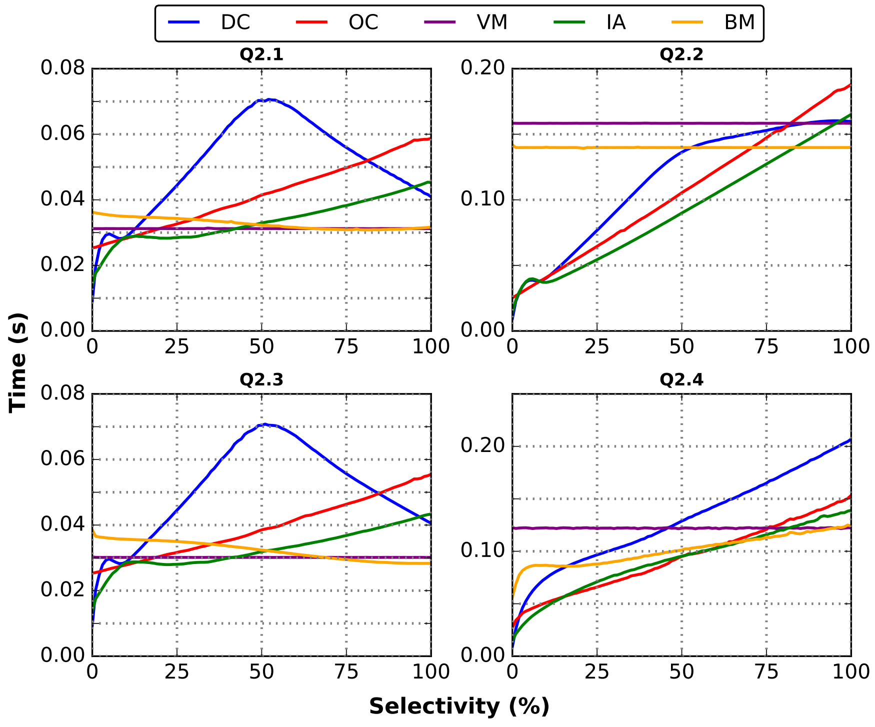
Flight	Args
Q1.1	OP=$*$, COL1=r_a, COL2=r_b
Q1.2	OP=$/$, COL1=r_a, COL2=r_b
Q1.3	OP=$*$, COL1=r_x, COL2=r_b
Q1.4	OP=$*$, COL1=r_x, COL2=r_y
Q2.1	OP=$*$, C=r_c1
Q2.2	OP=$/$, C=r_c1
Q2.3	OP=$*$, C=r_c2
Q2.4	OP=$*$, C=r_c3
Q3.1	S: s1, PRED: `s1_x < ?`
Q3.2	S: s1, PRED: `s1_x < 70 and r_x < ?`
Q3.3	S: s2, PRED: `s2_x < ?`
Q3.4	S: s2, PRED: `s2_x < 70 and r_x < ?`
Q4.1	S: s1, C: s1_c
Q4.2	S: s2, C: s2_c
Q4.3	S: s1, C: s1_pk
Q4.4	S: s2, C: s2_pk

(c) Query Flights

Figure 3.5: Microbenchmark

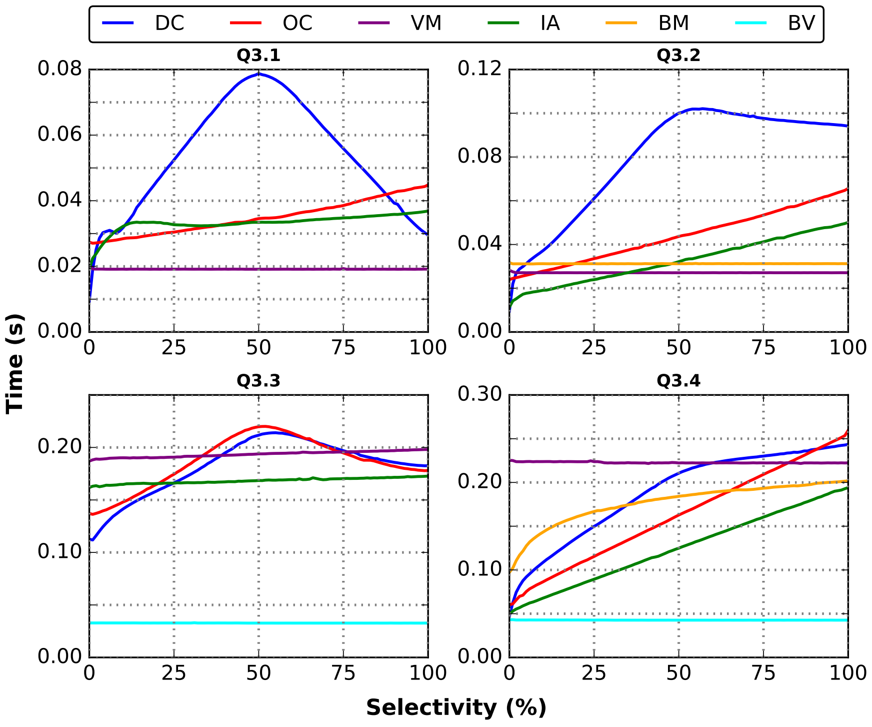


(a) Q1

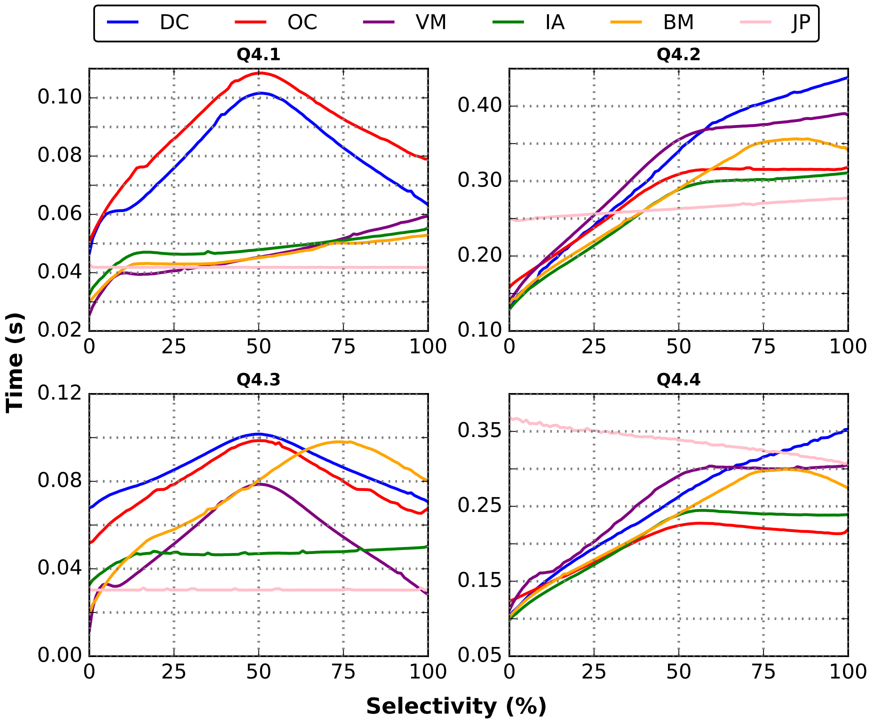


(b) Q2

Figure 3.6: Microbenchmark Results



(c) Q3



(d) Q4

Figure 3.6: Microbenchmark Results (cont.)

operator with a division, which is about an order-of-magnitude more expensive to compute. Therefore, the line continues upward as more data is selected.

Operator-centric (OC) avoids the branch misprediction problem using branch-free data copying, resulting in a steadily increasing runtime as selectivity increases. However, DC is slightly better on very low and high selectivity queries due to good CPU branch prediction.

As expected, value masking (VM) runs in constant time, irrespective of the selectivity. For the simple multiplication operator, it outperforms all others. However, the more expensive division operator in Q1.2 is an order of magnitude more expensive to compute, making VM prohibitively expensive for all but the least selective predicates. The additional access consolidation (AC) optimization applied in flights Q1.3 and Q1.4 provides a non-negligible speedup by merging redundant accesses in the selection and projection.

Finally, indexed access (IA) exhibits similar performance to OC, since it uses the same idea of passing cache-resident vectors of intermediate results between operators. Unlike OC, though, IA avoids the overhead associated with data copying for intermediates, thus allowing it to have better performance in almost all cases. IA generally performs worse than the predicate pullup techniques (i.e., VM and AC) except, notably, in Q1.2, where it exhibits the best performance because it avoids performing the expensive division unnecessarily.

Q2

This query (Figure 3.6b) tests group-by aggregation with a varying number of keys (i.e., 4, 32, and $100k$).

Flights Q2.1, Q2.2, and Q3.3 strongly resemble the results from Q1. DC is somewhat better relative to the alternative approaches because it can leverage the CPU's out-of-order execution capabilities to interleave the key lookup with the aggregate computation.

Unlike Q1, though, there is now a much clearer tradeoff between predicate pullups (i.e., VM and BM) and IA. These two approaches switch places at approximately 50% selectivity, which makes sense given the amount of wasted work.

Moreover, Q2.3 looks nearly identical to Q2.1, such that a relatively small number of buckets (in this case, 4 vs 32) has almost no impact on query performance. Group-by keys regularly fluctuate in this range from a few to a few dozen distinct keys.

Flight Q2.4, which tests a large number of group-by keys ($100k$), is notably different than the other flights. With a much larger number of keys, OC and IA start to blend together because query runtime is dominated by random memory accesses into the hash table for both approaches. Again, there is a narrow tradeoff between IA being better at lower selectivites while predicate pullups are better at higher selectivities, but BM outperforms all other approaches in the high (i.e., $> 75\%$) range. OC is slightly better than IA for some lower selectivities (about 15-20) because of a more consistent access pattern.

Q3

This query (Figure 3.6c) tests a two-way equijoin with a subsequent scalar aggregation. On the probe side, the query selects most (i.e., 75%) of the tuples in order to stress the join operator; otherwise, the relative importance of the join operator diminishes as more tuples are filtered out before the join.

Yet again, the results of Q3.1 strongly resemble Q1.1, with VM clearly outperforming all other approaches. As shown, for a small number of keys, the query-centric code generation strategy substantially outperforms the generic hash table version, both for the smaller and larger sizes of S. On the other hand, the technique tends to perform less well relative to the hash join-based algorithms for a larger number of keys. The exact number of keys where this technique no longer pays off is, however, hardware- and data-dependent, requiring the application of the memory-bandwidth calculations from Tupleware.

At larger scale factors, the join hash table falls out of the CPU cache and all lines blend together, as the join time becomes bound by random memory access on the probe side. This pattern does not hold true for BESPOKE, however, which remain inside the CPU cache due to the fact that they are much more compact. Even at these larger than cache sizes, the fast-lookup inclusion test still achieves a moderate (around 10%) speedup over alternatives in the middle (25% to 75%) selectivity range.

Q4

This query tests a two-way equijoin followed by a group-by aggregation. Again, 75% of tuples are selected on the probe side to stress the join operator.

Interestingly, for flight Q4.1, DC outperforms OC, due to tighter code that is more amenable to certain CPU features (e.g., out-of-order execution, prefetching). Otherwise, DC and OC are not contenders, except where OC outperforms all other approaches for moderate to high selecitivies in Q4.4. This is because of OC's access patterns, where it separates hash table lookups from the actual aggregation step, allowing time for the values to be fetched rather than stalling on them. Other approaches exhibit behavior similar to their performance in previous queries.

In all flights except Q4.4, the join pullup (JP) technique performs better for almost all selectivities. It performs worst, however, in Q4.4, since the disparity between distinct join keys and group-by keys (1M vs 100) is quite large.

Chapter 4

Rethinking Code Generation

As we have demonstrated throughout this dissertation, code generation can provide excellent performance compared to traditional query execution approaches. Based on our discussion in Section 1.1.2, though, it is relatively straightforward to achieve the type specialization and low overhead benefits of code generation in a vectorized model with precompiled operators simply by using careful design techniques. However, the unique advantage that code generation still holds over traditional vectorized query processing is the ability to apply holistic optimizations, where the DBMS can optimize across query operator borders.

In this chapter, we present ERSATZ, an extension of the vectorized query processing model that attempts to mimic the advantages of holistic optimization without needing to perform code generation in order to overcome the key limitations discussed in Section 1.1.3. ERSATZ starts with highly tuned implementations of traditional vectorized query processing operators and then adds a new class of precompiled fused operators, which reflect commonly occurring operator patterns. Simply by adding a handful of these new fused operators, ERSATZ unlocks a number of opportunities for holistic optimization that would otherwise be impossible in traditional vectorized approaches.

The remainder of this chapter is organized as follows. Section 4.1 summarizes the state of the art for code generation by revisiting the example from Chapter 1. Then, in Section 4.2, we describe our vectorized query processing model, called ERSATZ, that can mimic several of the optimizations of code generation without having to resort to using code generation. Each subsection of Section 4.2 takes an in-depth look at individual query operators that are important for OLAP queries, including (1) select, (2) join, and (3) project & aggregate.

4.1 Revisiting TPC-H Q6

Recall again TPC-H Q6, which was the example query from Section 1.1.1. While we presented a simple data-centric version of this query to illustrate the key ideas behind code generation, we have seen throughout this dissertation that many improvements on this basic approach are possible.

Figure 4.1 provides a comparison of three code generation versions of Q6 that we have discussed throughout this dissertation: (1) data-centric [95] from HyPer [75]; (2) operator-centric from Chapter 2; and (3) query-centric from Chapter 3. In the following, we explain the key differences between these approaches.

4.1.1 Data-centric Code Generation

Figure 4.1a shows the data-centric generation strategy, which produces a single `for` loop over all tuples stored in the `lineitem` table in order to compute the `revenue` (i.e., `sum(l_extendedprice * l_discount)`) on the subset of tuples that pass the predicate. Whereas an iterator-based approach would apply each operation as a separate `next` call, each of the different predicate operations (i.e., five comparison and four branching instructions) is merged together into a single `if` conditional statement, just as the `sum` and `multiply` operations are merged into a single imperative statement.

Merging operations together in this way maximizes the benefits of short-circuit evaluation and data locality while removing the overheads of the iterator model. However, the data-centric strategy unfortunately prevents many useful low-level optimizations (e.g., SIMD vectorization) due to the control dependency introduced by the branching code.

4.1.2 Operator-centric Code Generation

Unlike data-centric code generation, operator-centric strategies apply heuristics or build cost models to generate different code to group together operators on a case-by-case basis. Figure 4.1b shows the operator-centric code generation strategy presented in Chapter 2. The outer `for` loop iterates over the lineitem table in tiles (`i += TILE`), while the inner `for` loops handle the processing for each tile. Based on the results of the prepass predicate evaluation (first loop), tuple values that pass the predicate are copied to intermediate result vectors (second loop), and then the final computation of `revenue` is performed using those intermediate vectors (third loop).

By removing the control flow (i.e., `if` statement) from the loop, the compiler can now apply SIMD autovectorization on the first and third loops, which yields a 3x performance improvement over the data-centric approach for Q6.

4.1.3 Query-centric Code Generation

Figure 4.1c shows the query-centric code generation strategy from Chapter 3. Here, we see the techniques of (1) predicate pullups and (2) access consolidation. Specifically, the first inner loop stores a mask for every tuple in the `cmp` buffer by taking the bitwise `&` of the predicate comparisons, each of which will produce a 1 (i.e., true) or 0 (i.e., false). Then, the second loop uses the `cmp` values to mask the result of each multiplication before updating `revenue`. Again, as shown in our experimental evaluation, these techniques achieve almost a 5.5x and 2x performance improvement over the data- and operator-centric variants, respectively.

```
size_t i; revenue = 0.0;
for (i = 0; i < l_len; i++)
   if (l_shipdate[i] >= '1994-01-01'
         && l_shipdate[i] < '1995-01-01'
         && l_discount[i] >= 0.05
         && l_discount[i] <= 0.07
         && l_quantity[i] < 24)
      revenue += l_extendedprice[i] * l_discount[i];
```

(a) Data-centric

```
size_t i, j, len, pos; cmp[TILE];
tmp1[TILE]; tmp2[TILE]; revenue = 0.0;
for (i = 0; i < l_len; i += TILE) {
   len = l_len - i < TILE ? l_len - i : TILE;
   for (j = 0; j < len; j++)
      cmp[j] = l_shipdate[i+j] >= '1994-01-01'
             & l_shipdate[i+j] < '1995-01-01'
             & l_discount[i+j] >= 0.05
             & l_discount[i+j] <= 0.07
             & l_quantity[i+j] < 24;
   pos = 0;
   for (j = 0; j < len; j++) {
      tmp1[pos] = l_extendedprice[i+j];
      tmp2[pos] = l_discount[i+j];
      pos += cmp[j];
   }
   len = pos;
   for (j = 0; j < len; j++)
      revenue += tmp1[j] * tmp2[j];
}
```

(b) Operator-centric

```
size_t i, j, len; cmp[TILE]; tmp[TILE]; revenue = 0.0;
for (i = 0; i < l_len; i += TILE) {
   len = l_len - i < TILE ? l_len - i : TILE;
   for (j = 0; j < len; j++)
      cmp[j] = l_shipdate[i+j] >= '1994-01-01'
             & l_shipdate[i+j] < '1995-01-01'
             & l_quantity[i+j] < 24;
   for (j = 0; j < len; j++)
      tmp[j] = l_discount[i+j]
             * (l_discount[i+j] >= 0.05
              & l_discount[i+j] <= 0.07);
   for (j = 0; j < len; j++) {
      tmp[j] *= l_extendedprice[i+j] * cmp[j];
      revenue += tmp[j];
   }
}
```

(c) Query-centric

Figure 4.1: Code generation variants of Q6.

4.2 Ersatz

This section describes our ERSATZ approach, which is a vectorized query processing model that mimics the benefits of holistic code generation while avoiding the downsides of actually performing code generation. All of the optimization techniques described as part of ERSATZ have been presented in earlier parts of this dissertation, and the core contribution of this work is to demonstrate how to integrate these optimizations into a vectorized query processing model, thereby achieving state-of-the-art OLAP performance while avoiding the downsides of code generation. Since the techniques for holistic optimization were originally proposed in the context of code generation, translating them to a vectorized query processing model is not straightforward.

This section presents a detailed description of ERSATZ. First, we begin by providing an overview of the vectorized query processing model, as well as how we tune traditional vectorized operators (Section 4.2.1). Then, we describe how ERSATZ extends the traditional vectorized query processing model in order to mimic the holistic optimizations applied by code generation approaches for both core (Section 4.2.2) and less frequently used (Section 4.2.3) vectorized operators.

4.2.1 Vectorized Query Processing

ERSATZ is based on the concept of vectorized query processing, which refers to the bulk application of one operator at a time to vectors of tuples, materializing the intermediate results between each operator. In order to evaluate a query, a vectorized query processing engine translates SQL into either a tree of vectorized iterators [138] or assembly-like bytecode to be executed by an interpreter (e.g., MonetDB's MAL interpreter [8]).

Every operation necessary for query processing is encapsulated by a precompiled, type-specialized operator. We show a subset of the vectorized query operators most relevant to TPC-H in Table 4.1, with the operator name and operands (left) followed by the computation performed on those operands (right).

As an example, consider again the first date comparison from Q6 (`l_shipdate >= '1994-01-01'`), which ERSATZ would execute using the vectorized greater than or equal to operator (`ge`) from Table 4.1. In the following, we start by first examining a basic implementation of the `ge` function and then building upon this basic version with additional layers of optimizations to produce a heavily tuned version. Then, we discuss the limitations that this traditional vectorized approach imposes on the query optimization process.

A Basic Version

Figure 4.2a shows a basic version of `ge` from Table 4.1. The function has three parameters: (1) a pointer to the column `col` on which to perform the comparison (e.g., `l_shipdate`); (2) a constant value `val` against which to compare values from `col` (e.g., '1994-01-01'); and (3) a pointer to a selection vector `vec` that stores the indexes (i.e., offsets) of values from `col` that pass the selection condition. The `vec` parameter is then passed to subsequent operators as a way to communicate which tuples passed the predicates applied by earlier selection operators.

The `for` loop iterates over each of the indexes stored by `vec` and copies that index back to `vec` at offset `pos`, which will always be less than or equal to `i`. Then, the comparison `col[idx] >= val` tests the

Type	Name	Operation
Select	`eq(x,y,v)`	`v = x == y`
	`ne(x,y,v)`	`v = x != y`
	`gt(x,y,v)`	`v = x > y`
	`ge(x,y,v)`	`v = x >= y`
	`lt(x,y,v)`	`v = x < y`
	`le(x,y,v)`	`v = x <= y`
	`le(x,y,v)`	`v = x <= y`
	`bet(x,lo,hi,v)`	`v = lo <= x <= hi`
Join & Group	`build(m,k,v)`	`m.put(k,hashfn(k))`
	`probe(m,k,v)`	`v = m.in(k,hashfn(k))`
	`group(k,g,v)`	`g = get(k,h)`
Project & Aggregate	`add(x,y,t,v)`	`t = x + y`
	`sub(x,y,t,v)`	`t = x - y`
	`mul(x,y,t,v)`	`t = x * y`
	`div(x,y,t,v)`	`t = x / y`
	`concat(x,y,t,v)`	`t = x::y`
	`cast(x,t,v)`	`t = (typeof(t)) x`
	`count(n,t,v)`	`n++`
	`sum(n,x,v)`	`n += x`
	`min(n,x,v)`	`n = x < n ? x : n`
	`max(n,x,v)`	`n = x > n ? x : n`
	`gcount(n,g,v)`	`n[g] += v`
	`gsum(n,x,g,v)`	`n[g] += x * v`
	`gmin(n,x,g,v)`	`n[g] = x < n[g] ? x : n[g]`
	`gmax(n,x,g,v)`	`n[g] = x > n[g] ? x : n[g]`

Table 4.1: List of traditional vectorized operators.

```
void ge_dt(dt_t *col, dt_t val, vec_t *vec)
{
   size_t i, pos = 0;
   for (i = 0; i < vec->len; i++) {
      size_t idx = vec->idxs[i];
      vec->idxs[pos] = idx;
      pos += col[idx] >= val;
   }
   vec->len = pos;
}
```

(a) Basic Implementation

```
void ge_dt(dt_t *col, dt_t val, vec_t *vec)
{
   size_t i, pos = 0;
   if (vec->len == TILE) {
      int cmp[TILE];
      for (i = 0; i < TILE; i++)
         cmp[i] = col[i] >= val;
      for (i = 0; i < TILE; i++) {
         vec->idxs[pos] = i;
         pos += cmp[i];
      }
   }
   else
      for (i = 0; i < vec->len; i++) {
         size_t idx = vec->idxs[i];
         vec->idxs[pos] = idx;
         pos += col[idx] >= val;
      }
   vec->len = pos;
}
```

(b) Optimized Implementation

Figure 4.2: Basic and optimized versions of the vectorized greater than or equal to operator (`ge`) specialized for the date data type (`dt_t`).

data value at the specified index, which produces an integer output of either 1 (i.e., true) or 0 (i.e., false). If the outcome of the comparison is true, then pos is incremented to point to the next index value in vec on the following iteration. Otherwise, if the outcome of the comparison is false, then pos is not incremented and the index value stored in vec at that offset is overwritten on the following iteration. This "no-branch" implementation of a selection [116] (sometimes referred to as "predicated" selection [30]) transforms the control dependency from the branching code in Figure 4.1a into a data dependency, thereby eliminating the overhead incurred by CPU branch mispredictions.

In this chapter, we assume that all vectorized operators utilize the no-branch approach. However, by implementing both branch and no-branch alternatives, a query optimizer could make even lower-level decisions based on selectivity estimates during physical plan construction or adaptively at query runtime [111].

A cursory examination of the code in Figure 4.2a might suggest that the vectorized implementation will actually perform extra work relative to the pipelined code from Figure 4.1a, since no obvious if statement exists to enable short-circuit predicate evaluation. In the vectorized approach, however, as more tuples are filtered out by each successive selection operator, later operators will need to perform less work than earlier ones, thereby achieving a similar effect to short-circuit evaluation. While a simple concept, this observation forms the basis for several of the optimizations proposed by ERSATZ, which we describe in later sections.

Additionally, note that the function is specialized for the date data type (dt_t), which is crucial for maximizing query performance. As suggested in Section 1.1.2, we use C macros to write a single, generic version of each operator that then expands into the various type-specialized versions.

An Optimized Version

While the code in Figure 4.2a depicts a straightforward implementation of the vectorized ge operator, Figure 4.2b instead shows a more optimized version of the same operator, which incorporates a variant of the prepass optimization described in Chapter 2.

In this implementation, the if statement first checks to see whether the input tile is full by comparing the length of vec to TILE. Again, since this operator happens to evaluate the first selection condition applied in Q6, the input vector will always be full unless the length of the lineitem table is not evenly divisible by TILE, which would result in a partially full final tile. Similarly, if the operation were to appear later in the query plan, then earlier operators may have already filtered some tuples, which would also result in a partially full tile. The else branch therefore handles the processing of these partially full tiles and is equivalent to the basic variant from Figure 4.2a.

The first for loop in the if branch iterates over every value in the tile, performing the comparison and storing the result (i.e., 0 or 1) in the temporary array cmp. Then, the second for loop stores the index i in vec at offset pos, and then pos is incremented with the corresponding value from cmp. By using two separate loops over a tile with statically known length, the optimized version of the ge operator permits important low-level optimizations (e.g., SIMD vectorization, loop unrolling).

Type	Name	Operation
Select	`lbet(x,lo,hi,v)`	`v = lo <= x < hi`
	`rbet(x,lo,hi,v)`	`v = lo < x <= hi`
	`xbet(x,lo,hi,v)`	`v = lo < x < hi`
	`nbet(x,lo,hi,v)`	`v = !(lo <= x <= hi)`
Join & Group	`hash(k,h,v)`	`h = hashfn(k)`
	`hbuild(m,k,h,v)`	`m.put(k,h)`
	`hprobe(m,k,h,v)`	`m.in(k,h)`
	`hgroup(k,h,g,v)`	`g = get(k,h)`
Project & Aggregate	`addOP(x,y,z,t,v)`	`t = (x + y) OP z`
	`subOP(x,y,z,t,v)`	`t = (x - y) OP z`
	`mulOP(x,y,z,t,v)`	`t = (x * y) OP z)`
	`divOP(x,y,z,t,v)`	`t = (x / y) OP z)`
	`OPsum(n,x,y,v)`	`n += x OP y`
	`OPmin(n,x,y,v)`	`n = (x OP= y) < n ? x : n`
	`OPmax(n,x,y,v)`	`n = (x OP= y) > n ? x : n`
	`OPgsum(n,x,y,g,v)`	`n[g] += (x OP= y)`
	`OPgmin(n,x,y,g,v)`	`n[g] = (x OP= y) < n[g] ? x : n[g]`
	`OPgmax(n,x,y,g,v)`	`n[g] = (x OP= y) > n[g] ? x : n[g]`

Table 4.2: List of added ERSATZ **operators.**

Query Optimization Limitations

Given the optimized vectorized operator variant, we are well on our way to matching the performance of state-of-the-art code generation systems. However, unlike the code generation versions, the vectorized approach cannot benefit from holistic query optimization. Since operators are implemented as distinct functions, and they are capable of being strung together to execute an arbitrary SQL query, the optimizer cannot perform any optimizations that merge or rewrite the distinct operators, as is possible with code generation. This inability to optimize across operators can consequently add additional overhead to vectorized query processing that is not present in the code generation version, accounting for much of the remaining performance gap.

4.2.2 Core Operators

In order to overcome this major limitation of traditional vectorized query processing, our ERSATZ approach attempts to unlock opportunities for holistic optimization without resorting to code generation. As mentioned, ERSATZ adds a handful of new fused operators that implement explicitly merged versions of commonly occurring operator patterns, as shown in Table 4.2. The addition of these new operators consequently enables ERSATZ to apply holistic optimization rewrite rules previously only possible for code generation systems.

This section describes how ERSATZ integrates holistic optimization techniques for the three core operator types shown in Tables 4.1 and 4.2, each using an example TPC-H query to illustrate the key differences between the traditional vectorized approach and ERSATZ. Section 4.2.3 explains ERSATZ's optimizations for other, less frequently used operators separately.

The sample code shown in Figures 4.3-4.5 appears as assembly-like bytecode that represents the query plan produced by (1) the traditional vectorized approach and (2) ERSATZ. For each example, we explain how

```
vec = init()
tmp = init()
agg = 0.0
ge(l_shipdate, '1994-01-01', vec)
lt(l_shipdate, '1995-01-01', vec)
bet(l_discount, 0.05, 0.07, vec)
lt(l_quantity, 24, vec)
mul(l_extendedprice, l_discount, tmp, vec)
sum(agg, tmp, vec)
```

(a) Vectorized

```
vec = init()
agg = 0.0
ge(l_shipdate, '1994-01-01', vec)
lt(l_shipdate, '1995-01-01', vec)
ge(l_discount, 0.05, vec)
le(l_discount, 0.07, vec)
lt(l_quantity, 24, vec)
mulsum(agg, l_extendedprice, l_discount, vec)
```

(b) Ersatz

Figure 4.3: TPC-H Q6

ERSATZ conceptually applies rewrite rules to the basic vectorized version to produce the holistically optimized result.

Select

The select operator retrieves a filtered subset of tuples from a table based on a specified predicate. In this section, we focus on conjunctive selection predicates comprised of basic relational comparisons (i.e., ==, !=, >, >=, <, <=), which form the overwhelming majority of predicates in TPC-H queries. We discuss disjunctive predicates, as well as other more complex predicate operators, in Section 4.2.3.

As an example, recall TPC-H Q6 from Section 1.1, which involves applying several selection predicates before computing a simple scalar aggregation over the `lineitem` table. Figure 4.3 shows the vectorized and ERSATZ versions of this query.

The bytecode for the vectorized version (Figure 4.3a) is a fairly straightforward translation from the SQL for Q6 to the operators in Table 4.1. The code first initializes some variables (`vec`, `tmp`, `agg`), applies four selection operators (`ge`, `lt`, `bet`, `lt`), and finally performs the scalar aggregation on the filtered values (`mul`, `sum`).

Figure 4.3b shows the ERSATZ version of the bytecode for Q6. The differences between the ERSATZ bytecode and the vectorized version from Figure 4.3a are subtle yet important. Foremost, the between operator (`bet`) is split into separate greater than or equal to (`ge`) and less than or equal to (`le`) operators. This rewrite may seem minor, but the observation that an operator can be split in this way allows ERSATZ to make finer-grained decisions about whether to use a fused operator (for memory-bound cases by doing more work per data item) or to use split operators (for CPU-bound cases by doing less work per data item), as proposed in Chapters 2 and 3.

While the combination of this query and data distribution benefit from a split `bet` operator (i.e., separate `le` and `ge` operators), a different data distribution might change the optimization decision. For example, consider the `ge` and `lt` operators that filter tuples based on `l_shipdate`; they are currently CPU-bound and benefit from being split. However, if the data distribution were different causing them to become memory-bound, we would want to replace them with a fused operator, but none exists for this operator pattern.

ERSATZ therefore introduces a new left-inclusive between operator (`lbet`) that could be used to replace the `ge` and `lt` operators to optimize this memory-bound case. Similarly, we also add a right-inclusive (`rbet`), exclusive (`xbet`), and negated (`nbet`) variant of between, thereby expanding the types of query rewriting that we can perform. While not as flexible as code generation, this ability to optimize across operators enables ERSATZ to apply some of the most important holistic query optimizations without the downsides of code generation.

Additionally, note that the ERSATZ version merges the `mul` and `sum` operators from Figure 4.3a into a single operator: `mulsum`. This type of optimization is conceptually similar to the example of the between operator, and we discuss this type of optimization for projection and aggregation in greater detail later in this section.

Join & Group

Next, we consider the join and group operators, which organize tuples in a hash table. Specifically, we focus on equijoin, which is a binary operator that combines tuples from two relations that match the join criteria. Although other types of joins appear infrequently in OLAP queries, equijoins are by far the most common, as is demonstrated by their inclusion in almost every TPC-H query.

As a concrete example of the join and group operators, consider Q3, which retrieves the highest valued unshipped orders. This query represents the prototypical example of a join. First, the `customer` table is filtered to only the `'BUILDING'` market segment. Then, the `orders` table is filtered based on `o_orderdate` and joined with the previously filtered `customer` table. Finally, the `lineitem` table is filtered based on `l_shipdate`, with a `sum` aggregation grouped by `l_orderkey`, `o_orderdate`, and `o_shippriority`. Note that, in this last aggregation step, the aggregation hash table does not require a compound key, since `l_orderkey` functionally determines the other two group-by attributes.

Figure 4.4 shows the two alternative implementations for Q3. The vectorized approach (Figure 4.4a) again uses the familiar select and project operators seen previously in Q6. However, Q3 requires the introduction of some new operators: (1) `build` and `probe` to handle the build and probe phases, respectively, of a hash join; and (2) `group` and `gsum` to perform the final group-by sum aggregation.

The operators for the build side of the hash join first insert the key into a hash table. Likewise, on the probe side, the operator again tests to see whether that key exists in the hash table. Note that, unlike a code generation version, the vectorized approach must perform each selection and join probe separately rather than merging them into a single conditional statement. Similar to the join operators, the `group` operator retrieves the appropriate value to pass to the group-by sum operator `gsum`.

Figure 4.4b shows the ERSATZ alternative, which again tries to mimic the holistic optimization opportunities of code generation. This time, by splitting the `hbuild`, `hprobe`, and `hgroup` operators each into two distinct operators (e.g., `hbuild` is replaced with separate `hash` and `build`), ERSATZ has finer-grained

```
vec = init()
joina = init()
joinb = init()
tmpa = init()
tmpb = init()
tmpc = init()
agg = init()
eq(c_mktsegment, 'BUILDING', vec)
build(joina, c_custkey, vec)
lt(o_orderdate, '1995-03-15', vec)
probe(joina, o_custkey, vec)
build(joinb, o_orderkey, vec)
gt(l_shipdate, '1995-03-15', vec)
probe(joinb, l_orderkey, vec)
group(l_orderkey, tmpa, vec)
sub(1, l_discount, tmpb, vec)
mul(l_extendedprice, tmpc, vec)
gsum(agg, tmpc, tmpa, vec)
```

(a) Vectorized

```
vec = init()
joina = init()
joinb = init()
tmpa = init()
tmpb = init()
tmpc = init()
tmpd = init()
tmpe = init()
agg = init()
eq(c_mktsegment, 'BUILDING', vec)
hash(c_custkey, tmpa, vec)
hbuild(joina, c_custkey, tmpa, vec)
lt(o_orderdate, '1995-03-15', vec)
hash(o_custkey, tmpb, vec)
hprobe(joina, o_custkey, tmpb, vec)
hash(o_orderkey, tmpc, vec)
hbuild(joinb, o_orderkey, tmpc, vec)
gt(l_shipdate, '1995-03-15', vec)
hash(l_orderkey, tmpd, vec)
hprobe(joinb, l_orderkey, tmpd, vec)
hgroup(l_orderkey, tmpd, vec)
submul(1, l_discount, l_extendedprice, tmpe, vec)
gsum(agg, tmpe, tmpd, vec)
```

(b) Ersatz

Figure 4.4: TPC-H Q3

control over how the operators are organized and can make different decisions based on whether an operation is CPU- or memory-bound. In particular, by separating these operators in this way, we can take advantage of SIMD to speed up the hash computations, which would otherwise have been prevented by data dependencies introduced by needing to access hash-based data structures, as described in Chapter 2. If, on the other hand, the hash computations are not so expensive, ERSATZ can instead use the default combined operators.

Another benefit of this approach becomes clear in this query, where the key from the probe side of the `lineitem` join is later used as the key for the group-by aggregation. By separating the hash function, ERSATZ can reuse the intermediate hash values of the (potentially expensive) hash computation from the probe side of the join without having to re-hash the keys for the final group-by aggregation (somewhat similar to the idea of groupjoin [91]).

As mentioned, equijoins are by far the most common join operator, both in the research literature and in practice. Moreover, many equijoins use integers that represent a surrogate primary key. Compound and string-based keys are not uncommon, but with minor adjustments (e.g., multi-dimensional hash tables, hashing to an integer), the same algorithms used for integer key joins can be used in these cases as well. Other join operators (e.g., theta join, outer join, anti-join) also use well-defined algorithms and data structures, making them amenable to highly efficient, precompiled implementations, with type-specific optimizations where appropriate.

Finally, the `EXISTS` operator, which tests to see if at least one matching tuple exists in another relation, is highly similar to how ERSATZ would handle a join, and query optimizers will often rewrite an `EXISTS` clause as a join (e.g., in TPC-H Q4). Therefore, ERSATZ can apply the same exact techniques previously described for join optimizations.

Project & Aggregate

Finally, the last two operators we examine are projection and aggregation.

To concretely explore these operators, we reference Q1, which provides a sales summary report aggregated over all lineitems that had shipped by the specified date. Q1 is a good example of a query with relatively complex project and aggregate operations. The query involves only a single table (`lineitem`) but filters very few tuples such that almost all of the tuples are contained in the final result. Most importantly, Q1 includes a very complex and expensive to compute projection/aggregation component, which helps to highlight the differences between the different approaches shown in Figure 4.5.

By default, both versions perform common subexpression elimination (CSE) to avoid the wasted work of recomputing multiple equivalent intermediate values. The benefits of the holistic optimization are particularly noticeable in this query. For example, consider the `l_discount` attribute used to compute the subexpression `(1 - l_discount)`, which can be materialized and reused in both `sum_disc_price` and `sum_charge`. An approach using code generation could easily compute the subexpression concurrently with the average, which would be much more efficient. Similarly to `l_discount`, `l_extendedprice` would need to be looked at by a total of four different operators (i.e., `sum_base_price`, `sum_disc_price`, `sum_charge`, `avg_price`) in an interpreted version, since no reuse is possible.

ERSATZ again addresses this problem by mimicking holistic optimization and merging projection operators

```
vec = init()
tmpa = init()
tmpb = init()
tmpc = init()
tmpd = init()
tmpe = init()
agg = init()
le(l_shipdate, '1998-12-01' - interval '90' day, vec)
concat(l_returnflag, l_linestatus, tmpa, vec)
group(tmpa, tmpb, vec)
gcount(agg, tmpb, vec)
gsum(l_extendedprice, tmpb, vec)
gsum(l_quantity, tmpb, vec)
gsum(l_discount, tmpb, vec)
sub(1, l_discount, tmpc, vec)
mul(l_extendedprice, tmpc, tmpd, vec)
gsum(agg, tmpd, tmpb, vec)
add(1, l_tax, tmpe, vec)
mul(tmpd, tmpe, tmpf, vec)
gsum(agg, tmpf, tmpb, vec)
```

(a) Vectorized

```
vec = init()
tmpa = init()
tmpb = init()
tmpc = init()
tmpd = init()
agg = init()
le(l_shipdate, '1998-12-01' - interval '90' day, vec)
concat(l_returnflag, l_linestatus, tmpa, vec)
group(tmpa, tmpb, vec)
gcount(agg, tmpb, vec)
gsum(l_extendedprice, tmpb, vec)
gsum(l_quantity, tmpb, vec)
gsum(l_discount, tmpb, vec)
submul(1, l_discount, l_extendedprice, tmpc, vec)
gsum(agg, tmpc, tmpb, vec)
addmul(1, l_tax, tmpc, tmpd, vec)
gsum(agg, tmpd, tmpb, vec)
```

(b) Ersatz

Figure 4.5: TPC-H Q1

```
void or_ge_dt(dt_t *col, dt_t val, vec_t *pass, vec_t *fail)
{
   size_t i, pos = ppos = 0, fpos = 0;
   if (vec->len == TILE) {
      int cmp[TILE];
      for (i = 0; i < TILE; i++)
         cmp[i] = col[i] >= val;
      for (i = 0; i < TILE; i++) {
         pass->idxs[ppos] = i;
         ppos += col[i] >= val;
         fail->idxs[fpos] = i;
         fpos += col[i] < val;
      }
   }
   else
      for (i = 0; i < vec->len; i++) {
         size_t idx = pass->idxs[i];
         pass->idxs[ppos] = idx;
         ppos += col[idx] >= val;
         fail->idxs[fpos] = idx;
         fpos += col[idx] < val;
      }
   pass->size = ppos;
   fail->size = fpos;
}
```

Figure 4.6: OR Operator

where possible, cutting the number of required intermediate value materializations for Q1 in half. As shown in Figure 4.5b, ERSATZ merges the adjacent operators into the fused `submul` and `addmul` operators.

4.2.3 Other Operators

In addition to the most prominent OLAP operators we have covered throughout this section, there are many other operators available in SQL. Some of these operators have been historically difficult for interpreted execution paradigms to handle, so we focus in particular on those operators in this section in order to squeeze out the remaining performance difference between vectorized and code generated approaches.

Or

The previous section covered in detail our techniques for handling conjunctive predicates composed using the `AND` logical operator, but we have not yet discussed how to deal with disjunctive predicates that use the `OR` operator. Although not as common as conjunctive predicates (the `OR` operator is used in only three of the 22 TPC-H queries, specifically Q7, Q12, and Q19), efficient strategies for dealing with disjunctive predicates are crucial for implementing a general-purpose DBMS.

Historically, disjunctive predicates have been a problem for vectorized query processing models, where they often suffer from the need to perform extra work. Essentially, each predicate in the `OR` clause is evaluated separately on the same tile-sized input vector, and the results are then merged together to produce the union of all of the individual outputs. This way of individually processing each of the predicates is in stark contrast to the data-centric code generation case, which has the opportunity to merge all of the predicates into a single conditional statement and benefit from short-circuit evaluation.

The ERSATZ approach avoids performing this redundant work by only processing each tuple in the input vector exactly once, as shown in Figure 4.6. The basic idea is based on the bypass method [74], but we have tailored a simplified version that keeps separate true and false streams for expressions in a predicate.

The `or_ge_dt` function is structurally very similar to the conjunctive `ge_dt` implementation shown in Figure 4.2b. Again, the `if` statement checks whether the input tile is full, iterating either over the entire tile or only the value indexes `idxs` that were selected in the `else`. However, now instead of a single selection vector, note that there are two selection vectors: `pass` and `fail`. As their names suggest, the `pass` selection vector stores the indexes of the tuples that pass the predicate, in this case the >= operator, and the `fail` selection vector stores the indexes of the tuples that do not pass the predicate (i.e., <). Finally, the sizes of each of the selection vectors are set to the ending position.

Using this approach, we will end up doing some extra work but at the benefit of avoiding having to reprocess tuples that have already been examined. This tradeoff presents an interesting optimization opportunity, depending on the selectivity of the predicate. For instance, in the case of a predicate where many tuples pass, this technique will save considerable work by avoiding reprocessing a large part of the input vector. On the other hand, if the predicate selects very few tuples, most of the tuples in the input vector will need to be processed again anyway, so the extra overhead incurred by this technique would not make sense; rather, the traditional separate evaluation followed by merge approach would be superior. Therefore, ERSATZ can decide which version of the `OR` operator to use at query planning time in a type of holistic optimization, similar to how a traditional query optimizer would order predicates depending on relative selectivities.

In

The `IN` operator tests whether an attribute value is included in a set that is either statically defined in the query or dynamically determined during query execution. If the `IN` clause refers to a small, statically defined set of values (e.g., Q12, Q16, Q19), then the query optimizer can instead use several related `OR` statements to cover each possible value, applying exactly the same techniques we use to optimize disjunctive predicates. On the other hand, if the `IN` clause is large (e.g., generated by an automatic querying tool) or the values are dynamically determined (e.g., Q18, Q20), the query optimizer can instead use a hash-based set data structure to test for inclusion, similar to the previously described treatment of the `EXISTS` operator.

Case

The `CASE` operator is a logical switch statement that applies different processing logic based on the result of evaluating a Boolean condition. Interestingly, the `CASE` statement also bears strong similarities to the processing of multiple `OR` operators (or a statically defined `IN` clause). These similarities become readily apparent if one imagines each condition of the `CASE` statement as an individual predicate in an `OR` clause, where we need to determine whether each tuple passes that predicate. Just as with multiple `OR` predicates, we can reprocess the entire tile-sized vector of tuples to evaluate each `CASE` condition before doing the processing for that particular branch. However, by applying the previously described work-saving techniques, we avoid redundant processing for tuples that have already been examined and dealt with.

Chapter 5

Related Work

This chapter provides a detailed review of the literature. In the first section, we discuss work related to query processing for data analytics, which is the core focus of this dissertation. Then, in the second section, we highlight some of the more tangentially related areas that overlap with specific parts of our work.

While many of the references are prior art, a number of cited publications appeared either concurrently with or subsequent to the work conducted as part of this dissertation. We indicate cases where these publications were inspired by or directly built upon our work.

5.1 Query Processing

The two primary query processing models addressed in this dissertation are (1) code generation and (2) vectorization. In the following, we discuss each of these approaches, as well as attempts to combine them into a hybrid query processing model.

5.1.1 Code Generation

Using code generation for query evaluation is not a recent proposal. In fact, System R [25] originally compiled SQL statements directly to machine code by stitching together code fragments from a fragment library [36]. While beneficial for avoiding the overhead associated with interpreted query execution, this approach was eventually abandoned due to issues with software maintenance and debuggability [113], as well as poor portability [90].

Over two decades later, Daytona [65] pioneered the translation of queries into C programs, which could then be compiled into an executable and linked with the necessary libraries. Compared to machine code, the generation of relatively high-level C code avoided many of the main issues encountered with System R.

Similarly, JAMDB [113] translated queries to Java code while applying operator type specialization and aggressive expression inlining. Unlike the statically compiled C programs produced by Daytona, though, JAMDB sought to leverage the capability of the Java Virtual Machine to dynamically apply low-level optimizations at runtime.

HIQUE [82] used template-based code generation to compile queries into C programs with explicit data staging. Unlike past attempts at code generation, HIQUE's approach removed the overheads associated with traditional data access methods and iterator-based execution, which emerged as a key bottleneck for in-memory DBMSs. Later systems that also used template-based code generation include Hekaton [49, 59], H2O [21], and RAW [73].

Data-centric [95] code generation was proposed as the query execution framework for HyPer [75]. The key idea is to use a push-based, pipelined query processing model that attempts to maximize data locality, with data items remaining in CPU registers as long as possible. LLVM glue code is generated to stitch together precompiled operators, and then the query is JIT-compiled. Several newer systems (e.g., Tenzing [80], Impala [81, 126], ViDa [72], Proteus [71]) have similarly leveraged LLVM for query compilation, and LLVM-based JIT compilation has even been integrated into PostgreSQL [34, 89, 120].

LegoBase [78] includes a query engine written in Scala that generates specialized C code and allows for continuous optimization. This approach is based on the idea of "abstraction without regret," whereby a system implemented in a high level language does not sacrifice performance by leveraging generative programming.

The popularity of the Spark [3] distributed computing framework has prompted a number of attempts to augment performance using code generation. Initial versions of Spark SQL [23], for example, compiled query expressions into Java bytecode. Project Tungsten [18, 130] extended this capability by generating data-centric Java bytecode for executing the entire query. More recently, Flare [55, 56] was developed as a drop-in accelerator for Spark that generates more efficient C code.

Code generation has also been explored in a variety of other settings, including stream processing (e.g., SPADE [61, 62], DBToaster [19, 20, 76]), distributed execution (e.g., DryadLINQ [131], Steno [93], Radish [94]), and specialized hardware (e.g., MapD [92, 115], YDB [128], Hawk [32], HetExchange [41]).

Vectorization

Vectorization attempts to overcome the inefficiency of iterator-based execution models by amortizing the costs of this abstraction over a vector of tuples. The idea originated as part of the MonetDB/X100 project [30], which would eventually be commercialized as Vectorwise [138] (currently known as Actian Vector [17]).

The vectorized query processing model naturally lends itself to single-instruction, multiple data (SIMD) processing, where the same operation is applied simultaneously to different data elements. Widely available architectures currently support 128-, 256-, and 512-bit SIMD operations. A vast body of work has explored how to leverage SIMD processing for data analytics [40, 67, 69, 70, 106–109, 117, 118, 121, 123, 129, 136]. In particular, many of the techniques presented throughout Chapters 2 and 3 were inspired by the seminal work [136] on using SIMD instructions to implement query operators.

Other systems that utilize vectorized query processing include DB2 BLU [112], SQL Server 11 (Denali) [86], HANA [57, 58], and Quickstep [102].

5.1.2 Combined Approaches

Since code generation and vectorization are both highly efficient query processing models, the idea of combining them has received considerable attention. The first systematic comparison [122] of these two approaches examined three core query operators: (1) projection, (2) selection, and (3) hash join. The study concluded that neither approach was dominant and advocated instead for a combination of the two in order to achieve the best performance.

TUPLEWARE [42–45] proposed an operator-centric code generation strategy as a hybrid between data-centric code generation and vectorization. This strategy jointly considers properties of the operators, data, and underlying hardware in order to generate code that minimizes intermediate result materialization while maximizing opportunities for SIMD processing. We described these techniques at length in Chapter 2 and extended them to the query-centric strategy in Chapter 3.

Data Blocks [85] are a compressed columnar storage format in which code generation and vectorization are each used for different parts of query execution. Specifically, precompiled vectorized scan operators perform certain operations (e.g., decompression, selection) before passing the results to generated data-centric code for the remainder of query processing. Whereas TUPLEWARE introduced the prepass strategy (Chapter 2) to enable SIMD processing for selection operators, the Data Blocks approach uses a precomputed lookup table to shuffle selected values in a SIMD register into contiguous lanes.

Voodoo [105] is a vector algebra with a backend that generates OpenCL kernels. This abstraction allows for both portability and tunability, such that the backend can produce highly specialized code irrespective of the target hardware. Additionally, Voodoo's predicated lookup technique inspired several of the optimizations presented in Chapter 3.

Relaxed operator fusion [90] (ROF) is a different operator-centric code generation strategy that, like TUPLEWARE, blends data-centric code generation and vectorization by materializing intermediate results at staging points during query execution. Compared to TUPLEWARE, ROF has three main differences: (1) operating on full intermediate result vectors rather than partially full vectors; (2) using the precomputed lookup table technique from Data Blocks to enable SIMD for selection operators; and (3) embedding explicit prefetch instructions to mask data access latencies, in particular for operators that need to access a hash table (e.g., join, group-by aggregation).

An even more recent study [77] provided an in-depth, apples-to-apples comparison of data-centric code generation and vectorized query processing. Like previous studies, the main conclusion is that no clear winner between the two exists, with each approach performing better in certain cases. However, the choice to consider only the data-centric approach to code generation is a major limitation of this study. As this dissertation clearly demonstrates, more advanced code generation strategies like operator- (Chapter 2) and query-centric (Chapter 3) code generation can significantly outperform the simple data-centric strategy, thereby calling this main conclusion into question.

5.2 Other Related Areas

This section provides an overview of some areas of related work that are not at the heart of this dissertation but have some overlap with our proposed techniques.

5.2.1 Analytics Frameworks

The widespread success of the Hadoop [2] framework, which is based on the MapReduce [48] programming model, cemented the role of distributed data processing. Numerous extensions have been proposed to support iteration and shared state within MapReduce [22, 33, 53], and some projects (e.g., SystemML [63]) go a step further by providing a high-level language that is translated into MapReduce tasks.

Spark [3] and Flink [1] (formerly Stratosphere) built on Hadoop's success by making this type of processing more efficient in memory. These systems have taken steps in the right direction by providing richer APIs that can supply an optimizer with additional information about the workflow, permitting high-level workflow optimization [68].

5.2.2 Bit-Level Storage & Indexing

BitWeaving [88] is an alternative data storage format intended to alleviate memory bandwidth pressure and work better with modern CPUs. Specifically, BitWeaving comes in two varieties: (1) BitWeaving/V and (2) BitWeaving/H. BitWeaving/V physically stores data values as transposed bit vectors (i.e., columnar storage for the bits of values within a column) such that selection operators can operate on more tuples at once (however many bits fit into a register) and abort early if no matches are found, thus reducing the required memory bandwidth by excluding tuples without necessarily having to bring in the entire attribute values. BitWeaving/H better aligns non-transposed attribute values around word boundaries to prevent values from being split across words, thereby slowing down efficient CPU processing. These techniques are collectively implemented in Quickstep [102].

Bitmap indexes are used internally for query processing by many popular DBMSs in order to test existence or set inclusion, and various techniques that leverage bitmap indexes for query processing have also been proposed [97, 98].

5.2.3 Operator Specialization & Fusion

Operator specialization refers to tailoring a query operator for specific data types and usages, as we described in Chapter 1. MonetDB/X100 [30], for instance, leveraged precompiled vectorized primitives highly tuned for individual data types. Similarly, micro-specialization [133–135] replaces generic query processing functions with much more efficient, type-specialized code.

Another form of operator specialization involves choosing different low-level implementations on a case-by-case basis, such as choosing between a branch vs no-branch [116] implementation of a selection operator. Micro Adaptivity [111] takes this idea of specialization a step further and switches operator implementations on the fly depending on query statistics.

Operator fusion is an optimization that involves merging two separate operators into a single operator, which plays a key role in many of the techniques presented in this work. Although we have discussed this technique specifically in the context of code generation, operator fusion is widely leveraged in a variety of data processing systems. For example, the Weld [99, 100] project enables cross-library operator fusion. Similarly, SPOOF [29, 54] proposes rewrite rules to fuse linear algebra operators.

Chapter 6

Conclusion and Future Work

Throughout this dissertation, we have explored how to push code generation to the limit for in-memory data analytics. We presented several novel code generation techniques, as well as a new vectorized query processing framework that can mimic the benefits of these techniques in many instances using only precompiled operators. Overall, our work has inspired many extensions of our techniques for code generation [18,55,56,90,94,105,130] and operator fusion [29,54,99,100] in this new context.

First, we described TUPLEWARE, a novel architecture for compiling the types of complex analytics tasks that have become commonplace for a wide variety of users. As part of this project, we developed the operator-centric code generation strategy, which generates code to fuse query operators on a case-by-case basis. This work opened the door for an entire new breed of optimizations that consider data statistics, UDF characteristics, and the underlying hardware in order to better optimize generated code. Our experiments demonstrated that our TUPLEWARE prototype can achieve orders-of-magnitude speedups over alternative systems and showed that the proposed heuristics can further improve performance.

We then expanded upon our operator-centric code generation strategy with BESPOKE in Chapter 3. Instead of organizing the generated code at the granularity of traditional operators, BESPOKE takes a more holistic, query-centric approach that freely optimizes across operator boundaries. In particular, our techniques heavily leverage "predicate pullups" to reorder operators in unconventional ways, and our experiments demonstrated clear performance improvements over both data- and operator-centric code generation strategies.

However, despite the clear performance benefits, there are also many obvious drawbacks associated with code generation for query evaluation. We summarized several of the main pros and cons of code generation techniques in Chapter 1, but the major concern that could prevent widespread adoption is the need for a complete rearchitecting of current systems. Moreover, the need to modify query optimizers to handle edge cases in the code generation process introduces significantly more complexity.

After a critical reexamination of code generation techniques, which have recently become a perceived necessity for implementing a state-of-the-art DBMS, we concluded that the overhead of starting from scratch to build a code generation system is not worth the return on investment for many traditional OLAP workloads. Informed by these concerns, we developed ERSATZ (Chapter 4), a vectorized query processing model that achieved performance close to code generation on the well-known TPC-H decision support benchmark. We hope that this work serves as a resource for researchers and system designers to achieve state-of-the-art OLAP performance without having to pay the costly price of implementing a code generation system completely from scratch.

Yet, while we have shown that ERSATZ can (nearly) match code generation performance for OLAP queries, it remains unclear whether this finding generalizes to other types of workloads or problem domains. Further investigation will therefore be required in order to determine whether our techniques apply in these other settings. As such, obvious avenues for future work include reexamining our techniques for specialized hardware (e.g., RDMA [28], coprocessors [50]) and new workloads (e.g., interactive data exploration [46,47,60,132], time series analysis [51,52]).

Besides these obvious extensions of our techniques, we believe that code generation has a bright future in the overall data management landscape. In the following, we discuss what we consider to be three of the most impactful directions for future work.

6.1 Dynamic Code Generation

Just-in-time (JIT) compilation allows programming languages like Java to apply adaptive optimizations [24] based on runtime profiling. In contrast, nearly all state-of-the-art code generation approaches for query evaluation, including those presented in this dissertation, involve a static process of first creating an optimized query plan and then translating that plan to executable code in full before query execution begins. Even for techniques that incrementally compile and optimize their generated code [79], optimization passes apply only low-level optimizations and do not fundamentally impact the high-level code structure.

However, the ability to dynamically switch between different operator implementations based on runtime statistics (e.g., Micro Adaptivity [111]) has demonstrable benefits for traditional query processing paradigms. Therefore, we believe that applying the same idea to dynamically regenerate and redeploy query execution code optimized using runtime statistics (i.e., on-stack replacement) could open the door for an entirely new class of code generation techniques.

Going a step further, a system could even incrementally generate code in a lazy fashion, using statistics gathered during earlier stages of query execution to optimize later stages. For instance, queries with blocking operators (e.g., joins) could gather runtime statistics that were unknown prior to execution and leverage those statistics to generate highly specialized code. This type of lazy code generation also presents many interesting opportunities to directly embed instance-specific information (e.g., literal values) based not only on metadata from the system catalog but even from derived intermediate results that are unknown until runtime.

6.2 Multi-query Code Generation

Techniques to optimize across multiple concurrently executing OLAP queries have been extensively explored in traditional query processing settings. For example, multiple-query optimization [119] attempts to minimize redundant computation by finding common subexpressions or other opportunities to share intermediate results across different queries. Similarly, shared scans try to best utilize the available bandwidth by attaching multiple queries with possibly disjoint predicates to the same cooperative scan over the data, and they are implemented in a number of DBMSs (e.g., DB2 [84], MonetDB [137], Blink [110], Crescando [125]). Not surprisingly, these techniques have also been explored in the context of MapReduce-style systems [96, 124, 127].

However, state-of-the-art code generation strategies, including those proposed as part of this dissertation, optimize for the execution of a single query running in isolation. While single-query performance is important, most real-world settings have multiple users issuing queries concurrently. Furthermore, modern visual data exploration tools (e.g., Tableau [13], Vizdom [46]), might issue several dozen queries at the same time triggered by just one single user interaction. While many of the code generation techniques we have discussed throughout this dissertation will apply equally well when considering concurrently executing queries, we believe that an entirely new research challenge emerges related to generating code that optimizes *across* multiple queries.

Code generation may, in fact, permit significantly more fine-grained opportunities for sharing work across multiple queries than is possible with traditional query processing models. For example, since code generation systems are no longer bound by traditional operator boundaries, they can arbitrarily reorder and interleave operations from different queries, merging computations as necessary.

Moreover, code generation presents a solution for a common shortcoming of traditional multi-query optimization approaches, namely the need to wait for multiple queries to arrive in order to perform efficient batch execution. Instead, by combining these ideas with dynamic code generation, we could redeploy multi-query code as new queries arrive, rather than waiting for batches of queries, while still exploiting the maximum optimization potential.

6.3 Online Transaction Processing

This dissertation has addressed code generation exclusively for data analytics, including OLAP and complex UDF workflows. Similarly, the other two areas proposed as promising directions for future work (i.e., dynamic and multi-query code generation) focus on data analytics workloads.

We believe, though, that code generation holds enormous potential for other core data management problems, including online transaction processing (OLTP). Whereas data analytics workloads involve computing high-level summaries over large amounts of data, OLTP deals instead with short-lived, repetitive updates to small pieces of the database state (e.g., banking transactions, retail purchases).

This completely different setting presents a multitude of opportunities for low-level code generation optimizations. For example, a system could leverage code generation to streamline execution by including only the required functionality in a RISC-style fashion [37], or even by producing a highly tuned executable that is specific to the workload and data [39]. Additionally, while this dissertation has focused on the problem of generating code, OLTP workloads may benefit from actually generating different data layouts or specialized data structures tailored to data access patterns.

Bibliography

[1] Apache Flink. `https://flink.apache.org/`.

[2] Apache Hadoop. `https://hadoop.apache.org`.

[3] Apache Spark. `https://spark.apache.org/`.

[4] BLAS. `https://netlib.org/blas/`.

[5] How SQLite Is Tested. `https://www.sqlite.org/testing.html`.

[6] Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `https://www.agner.org/optimize/instruction_tables.pdf`.

[7] Mahout. `https://mahout.apache.org/`.

[8] MAL Interpreter. `https://www.monetdb.org/Documentation/Manuals/MonetDB/interpreter`.

[9] MLlib. `https://spark.apache.org/mllib/`.

[10] MySQL. `https://www.mysql.com/`.

[11] PostgreSQL. `https://www.postgresql.org/`.

[12] SQLite. `https://www.sqlite.org/index.html`.

[13] Tableau. `https://www.tableau.com/`.

[14] TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[15] UK Crime Dataset. `https://data.police.uk/`.

[16] Wikipedia Webgraph. `https://dumps.wikimedia.org/`.

[17] Actian vector. `https://www.actian.com/analytic-database/vector-analytic-database/`, 2019.

[18] S. Agarwal, D. Liu, and R. Xin. Apache spark as a compiler: Joining a billion rows per second on a laptop. `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`, 2016.

[19] 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.

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

[21] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114, 2014.

[22] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, R. Grover, Z. Heilbron, Y. Kim, C. Li, N. Onose, P. Pirzadeh, R. Vernica, and J. Wen. ASTERIX: an open source system for ”big data” management and analysis. *PVLDB*, 5(12):1898–1901, 2012.

[23] 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, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394, 2015.

[24] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeño JVM. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 47–65, 2000.

[25] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, P. Tiberio, I. L. Traiger, B. W. Wade, and R. A. Yost. System R: A relational data base management system. *IEEE Computer*, 12(5):42–48, 1979.

[26] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst.*, 4(1):1–29, 1979.

[27] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011, Miami, Florida, USA, October 24-28, 2011*, pages 591–596, 2011.

[28] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It’s time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[29] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *PVLDB*, 11(12):1755–1768, 2018.

[30] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237, 2005.

[31] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.

[32] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.*, 27(6):797–822, 2018.

[33] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[34] D. Butterstein and T. Grust. Precision performance surgery for postgresql: Llvm-based expression compilation, just in time. *PVLDB*, 9(13):1517–1520, 2016.

[35] M. J. Cafarella and C. Ré. Manimal: Relational optimization for data-intensive programs. In *Proceedings of the 13th International Workshop on the Web and Databases 2010, WebDB 2010, Indianapolis, Indiana, USA, June 6, 2010*, 2010.

[36] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. 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(10):632–646, 1981.

[37] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 1–10, 2000.

[38] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 345–356, 2002.

[39] A. Cheung. Towards generating application-specific data management systems. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[40] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[41] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, 12(5):544–556, 2019.

[42] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[43] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014.

[44] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: "big" data, big analytics, small clusters. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[45] A. Crotty, A. Galakatos, and T. Kraska. Tupleware: Distributed machine learning on small clusters. *IEEE Data Eng. Bull.*, 37(3):63–76, 2014.

[46] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2027, 2015.

[47] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The case for interactive data exploration accelerators (ideas). In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 11, 2016.

[48] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[49] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[50] K. Dursun, C. Binnig, U. Çetintemel, and R. Petrocelli. Silicondb: rethinking dbmss for modern heterogeneous co-processor environments. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 10:1–10:4, 2017.

[51] P. Eichmann, A. Crotty, A. Galakatos, and E. Zgraggen. Discrete time specifications in temporal queries. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017, Extended Abstracts.*, pages 2536–2542, 2017.

[52] P. Eichmann and E. Zgraggen. Evaluating subjective accuracy in time series pattern-matching using human-annotated rankings. In *Proceedings of the 20th International Conference on Intelligent User Interfaces, IUI 2015, Atlanta, GA, USA, March 29 - April 01, 2015*, pages 28–37, 2015.

[53] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. C. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010*, pages 810–818, 2010.

[54] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[55] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Native compilation for heterogeneous workloads in apache spark. *CoRR*, abs/1703.08219, 2017.

[56] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 799–815, 2018.

[57] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[58] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[59] C. Freedman, E. Ismert, and P. Larson. Compilation in the microsoft SQL server hekaton engine. *IEEE Data Eng. Bull.*, 37(1):22–30, 2014.

[60] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[61] B. Gedik, H. Andrade, and K. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 847–856, 2009.

[62] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1123–1134, 2008.

[63] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 231–242, 2011.

[64] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[65] R. Greer. Daytona and the fourth-generation language cymbal. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 525–526, 1999.

[66] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 267–276, 1993.

[67] S. Héman, N. Nes, M. Zukowski, and P. A. Boncz. Vectorized data processing on the cell broadband engine. In *Workshop on Data Management on New Hardware, DaMoN 2007, Beijing, China, June 15, 2007*, page 4, 2007.

[68] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J. Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1292–1295, 2013.

[69] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[70] H. Inoue and K. Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, 2015.

[71] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[72] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[73] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014.

[74] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, pages 336–347, 1994.

[75] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[76] O. Kennedy, Y. Ahmad, and C. Koch. Dbtoaster: Agile views for a dynamic data management system. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 284–295, 2011.

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

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

[79] A. Kohn, V. Leis, and T. Neumann. Adaptive execution of compiled queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 197–208, 2018.

[80] S. T. C. Konigsmark, L. K. Hwang, D. Chen, and M. D. F. Wong. System-of-pufs: Multilevel security for embedded systems. In *2014 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2014, Uttar Pradesh, India, October 12-17, 2014*, pages 27:1–27:10, 2014.

[81] 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, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[82] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 613–624, 2010.

[83] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[84] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1136–1145, 2007.

[85] 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. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[86] P. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1177–1184, 2011.

[87] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[88] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 289–300, 2013.

[89] D. Melnik. Speeding up query execution in postgresql using llvm jit compiler. `https://llvm.org/devmtg/2016-09/slides/Melnik-PostgreSQLLLVM.pdf`, 2016.

[90] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

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

[92] T. Mostak. An overview of mapd (massively parallel database), 2013.

[93] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 121–131, 2011.

[94] B. Myers, M. Oskin, and B. Howe. Compiling queries for high-performance computing, 2016.

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

[96] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[97] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[98] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 38–49, 1997.

[99] S. Palkar, J. J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *PVLDB*, 11(9):1002–1015, 2018.

[100] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkopf, S. P. Amarasinghe, and M. Zaharia. A common runtime for high performance data analysis. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[101] 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`.

[102] 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. *PVLDB*, 11(6):663–676, 2018.

[103] A. Pavlo. Building a New Database Management System in Academia. `http://www.cs.cmu.edu/~pavlo/blog/2017/03/building-a-new-database-management-system-in-academia.html`.

[104] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

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

[106] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1493–1508, 2015.

[107] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 6, 2013.

[108] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 6:1–6:6, 2014.

[109] O. Polychroniou and K. A. Ross. Efficient lightweight compression alongside fast scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN 2015, Melbourne, VIC, Australia, May 31 - June 04, 2015*, pages 9:1–9:6, 2015.

[110] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[111] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1231–1242, 2013.

[112] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[113] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 23, 2006.

[114] B. Recht, C. Ré, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 693–701, 2011.

[115] C. Root and T. Mostak. Mapd: a gpu-powered big data analytics and visualization platform. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference, SIGGRAPH '16, Anaheim, CA, USA, July 24-28, 2016, Talks*, pages 73:1–73:2, 2016.

[116] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 109–120, 2002.

[117] K. A. Ross. Efficient hash probes on modern processors. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1297–1301, 2007.

[118] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 351–362, 2010.

[119] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[120] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher. Runtime specialization of postgresql query executor. In *Perspectives of System Informatics - 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers*, pages 375–386, 2017.

[121] E. A. Sitaridi, O. Polychroniou, and K. A. Ross. Simd-accelerated regular expression matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 8:1–8:7, 2016.

[122] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 33–40, 2011.

[123] D. Song and S. Chen. Exploiting SIMD for complex numerical predicates. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pages 143–149, 2016.

[124] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[125] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

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

[127] G. Wang and C. Chan. Multi-query optimization in mapreduce framework. *PVLDB*, 7(3):145–156, 2013.

[128] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *PVLDB*, 7(11):1011–1022, 2014.

[129] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[130] R. Xin and J. Rosen. Project tungsten: Bringing apache spark closer to bare metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`, 2015.

[131] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 1–14, 2008.

[132] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. How progressive visualizations affect exploratory analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(8):1977–1987, 2017.

[133] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-specialization: dynamic code specialization of database management systems. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pages 63–73, 2012.

[134] R. Zhang, R. T. Snodgrass, and S. Debray. Application of micro-specialization to query evaluation operators. In *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 315–321, 2012.

[135] R. Zhang, R. T. Snodgrass, and S. Debray. Micro-specialization in dbmses. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 690–701, 2012.

[136] 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, Madison, Wisconsin, USA, June 3-6, 2002*, pages 145–156, 2002.

[137] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 723–734, 2007.

[138] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1349–1350, 2012.