An Architecture for Compiling UDF-centric Workflows

Andrew Crotty Alex Galakatos Kayhan Dursun
Tim Kraska Carsten Binnig Ugur Cetintemel Stan Zdonik
Department of Computer Science, Brown University
{firstname_lastname}@brown.edu

ABSTRACT

Data analytics has recently grown to include increasingly sophisticated techniques, such as machine learning and advanced statistics. Users frequently express these complex analytics tasks as workflows of user-defined functions (UDFs) that specify each algorithmic step. However, given typical hardware configurations and dataset sizes, the core challenge of complex analytics is no longer sheer data volume but rather the computation itself, and the next generation of analytics frameworks must focus on optimizing for this computation bottleneck. While query compilation has gained widespread popularity as a way to tackle the computation bottleneck for traditional SQL workloads, relatively little work addresses UDF-centric workflows in the domain of complex analytics.

In this paper, we describe a novel architecture for automatically compiling workflows of UDFs. We also propose several optimizations that consider properties of the data, UDFs, and hardware together in order to generate different code on a case-by-case basis. To evaluate our approach, we implemented these techniques in TUPLEWARE, a new high-performance distributed analytics system, and our benchmarks show performance improvements of up to three orders of magnitude compared to alternative systems.

1. INTRODUCTION

Motivation: The growing prevalence of big data across all industries and sciences is causing a profound shift in the nature and scope of analytics. Increasingly complex computations, such as machine learning (ML) and advanced statistics, are quickly becoming the norm. Generally, users express these types of tasks as workflows of *user-defined functions* (UDFs), where each UDF represents a distinct step in the algorithm.

Current analytics frameworks that target UDF-centric workflows (e.g., Hadoop [1], Spark [44]) are designed to meet the needs of giant Internet companies; that is, they are built to process petabytes of data in cloud deployments consisting of thousands of cheap commodity machines. Yet non-tech companies like banks and retailers—or even the typical data scientist—seldom operate deployments of that size, instead preferring smaller clusters with more reliable hardware. In fact, recent industry surveys reported that the median Hadoop cluster was fewer than 10 nodes, and over 65% of users operate clusters smaller than 50 nodes [23, 29, 34].

Furthermore, the vast majority of users typically analyze relatively small datasets. For instance, the average Cloudera customer rarely works with datasets larger than a few terabytes in size [19], and commonly analyzed behavioral data peaks at around 1TB [16]. Even companies as large as Facebook, Microsoft, and Yahoo! frequently perform ML tasks on datasets smaller than 100GB [37]. Rather, as users strive to extract more value from their data, the computation itself becomes the true problem.

Targeting complex analytics workloads on smaller clusters fundamentally changes the way we should design analytics tools. Most current systems focus on the major challenges associated with massive datasets and large cloud deployments, where I/O is the primary bottleneck and failures are common [20]. However, the next generation of analytics frameworks should optimize instead for the computation bottleneck.

In order to better utilize the CPU, other work [26, 30] has explored techniques for compiling traditional SQL queries, but these approaches fall short when applied to UDF workflows. Since the properties of individual UDFs can directly impact compile-time optimization, we argue that UDFs must exist at the core of the optimization process and should no longer be treated as black boxes.

Contributions: This paper describes a novel architecture for the automatic compilation of UDF workflows in the context of compute-intensive, in-memory analytics. We propose a method for compiling UDF-centric workflows, as well as several code generation optimization heuristics that jointly consider properties of the data, UDFs, and underlying hardware. The key idea of this work is to integrate high-level query optimization techniques with low-level compiler techniques in order to unlock a new breed of optimizations that were previously impossible.

Our architecture leverages the LLVM [28] 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 [30, 42] and UDF introspection for high-level workflow optimizations [15, 41], we know of no other approach that combines these two ideas in order to optimize UDF workflows at the code generation level.

To implement the proposed architecture, we developed TUPLEWARE, a new high-performance distributed analytics system. Our benchmarks demonstrate that our novel techniques achieve orders-of-magnitude performance improvements over alternative systems. In summary, we make the following contributions:

Proceedings of the VLDB Endowment, Vol. 8, No. 12

- We present a novel architecture that leverages LLVM for compiling UDF-centric workflows into distributed programs.
- We propose several code generation optimizations that consider properties of the data, UDFs, and underlying hardware together.
- We describe a programming model with explicit shared state and semantics that enable low-level code generation optimizations.
- We implemented our techniques in TUPLEWARE, and our benchmarks show speedups of up to three orders of magnitude over other systems for common analytics tasks.

Outline: The remainder of the paper is organized as follows. Section 2 provides a high-level overview of our novel architecture. In Section 3, we describe the important aspects of the user frontend. Section 4 explains our process for compiling UDF-centric workflows. In Section 5, we propose several code generation optimization heuristics. Section 6 mentions some of the primary distributed execution challenges. We then present our evaluation in Section 7, discuss related work in Section 8, and finally conclude in Section 9.

2. OVERVIEW

Architecture: As shown in Figure 1, our proposed architecture consists of three parts. The *Frontend* (Section 3) 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 4). Compiled workflows are then executed automatically on a cluster during the *Execution* phase (Section 6).

This paper primarily focuses on our novel process for compiling UDF workflows. Although other systems [41, 15] 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. Additionally, systems exploring code generation either generate only glue code to connect precompiled operators [24] or focus only on traditional SQL queries [25, 18, 42].

Example: Figure 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 the paper. 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, 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

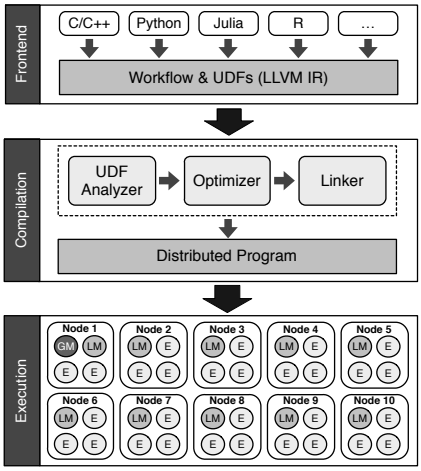


Figure 1: An overview of our proposed architecture. The Frontend allows users to compose UDF workflows in any LLVM-supported language (top boxes). These workflows are then translated to self-contained distributed programs during the three-stage Compilation process. Finally, the Execution phase deploys 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).

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.

3. FRONTEND

In many regards, our Frontend is similar to other recent frameworks (e.g., Spark, Stratosphere [41], DryadLINQ [43]), 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 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 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.

3.1 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

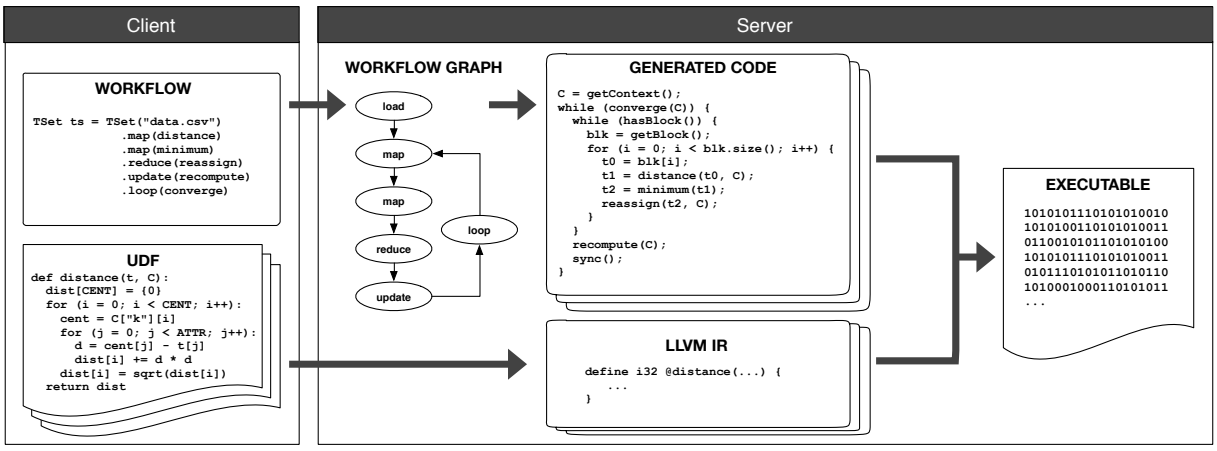


Figure 2: The full lifecycle of a workflow.

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 5.1
Aggregate	reduce$(T)(\lambda, k?)$	$(t_1, t_2) \rightarrow t'$ $(t, C) \rightarrow ()$	Section 5.2
Relational	selection$(T)(\lambda)$ join$(T_1, T_2)(\lambda)$	$t \rightarrow b$ $(t_1, t_2) \rightarrow b$	Section 5.3
Control	loop$(T)(\lambda)$ update$(T)(\lambda)$	$C \rightarrow b$ $C \rightarrow ()$	-

Table 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.

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 5. As shown in Table 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 1, apply UDFs have read-only access to an optionally provided Context $C?$. For example, the k-means `distance` UDF (Figure 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 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 (discussed further in Section 3.2). 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 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.

3.2 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 [13] 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! [33] could use asynchronous updates to improve performance. This paper 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 [27] 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 paper.

3.3 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 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.

4. 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 1, this process consists of three parts: (1) UDF Analyzer, (2) Optimizer, and (3) Linker.

UDF	Type	Vectorizable	Compute Time		Load Time
			Predicted	Actual	
`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: UDF statistics for the k-means algorithm.

4.1 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 shows the UDF statistics for the k-means example from Section 2.

Vectorizability: Vectorizable UDFs can use *single instruction multiple data* (SIMD) registers to achieve data level 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× speedup. We can leverage the operator semantics from Section 3.1 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 [3] provide cycles per instruction estimates for the given hardware. Adding together these estimates yields a rough projection for 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:

$$Load\,Time = \frac{Clock\ Speed \times Operand\ Size}{Bandwidth\ per\ Core} \quad (1)$$

For example, the load time for the `distance` UDF as shown in Table 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: $3.75\ cycles = \frac{2.8GHz \times (2 \times 4B)}{5.97GB/s}$.

4.2 Optimizer

The key idea of this work is to integrate high-level query optimization techniques with low-level compiler techniques in order to apply new optimizations that were previously impossible. As other work has shown [26], SQL query compilation can harness the full potential of the underlying hardware, and we extend these techniques by applying them to UDF-centric workflows. 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. 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 5.

4.3 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.

5. OPTIMIZATIONS

The most interesting and unique opportunities for optimizing a UDF workflow fall into the third category described in Section 4.2, 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 7).

5.1 Program Structure

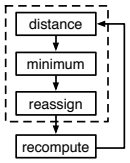
The first optimization we examine considers the most fundamental aspect of any code generation strategy, the overall *program structure*, which refers to the organization of the generated control flow code. This section describes two existing approaches to program structure and then presents our first 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 in Section 5.2.1 specifically for group-by aggregations.

5.1.1 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* [30] strategy (HyPer [24]) 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.

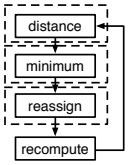
```
data[N];
while (converge()) {
  for (i = 0; i < N; i++) {
    dist = distance(data[i]);
    min = minimum(dist);
    reassign(min);
  }
  recompute();
}
```



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* strategy (MonetDB [46] [1]) 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 variant (Vectorwise [47]) performs 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.

5.1.2 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 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 (Section 4.1) in order to partition P into a set of vectorizable and nonvectorizable 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

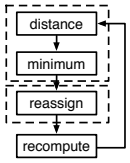
[1]MonetDB does not fully compile queries; rather, the system produces assembly-like language (MAL) for execution by a VM.

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 memory bandwidth bottleneck discussed in Section 4.1. 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), 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 discussed in Section 3.1 (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.

5.2 Aggregation

As described in Section 3.1, 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.

5.2.1 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 (Section 3.1); (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, similar to the example from Section 5.1.2, 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 [36, 31]. Other work [38] 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.

5.2.2 Context Variables

Recall that reduce UDFs can also perform aggregations by updating shared state Context variables (Section 3.2). 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), 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
assign = t1[ATTR];
for (i = 0; i < ATTR; i++)
  c["sum"][assign][i] += t1[i];
c["count"][assign]++;
```

```
//optimized
assign = t1[ATTR];
offset = assign * (ATTR + 1);
for (i = 0; i < ATTR; i++)
  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.

5.3 Selection

Optimizing selections is a well-studied problem, but our 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.

5.3.1 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 [35] 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 the group-by aggregations described in Section 5.2.1, 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.

5.3.2 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 prepass strategy described in Section 5.3.1 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.

5.3.3 Cost Model

We propose a cost model in Equation 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.

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

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 3: Cost model notation.

Each term models the important features of the various strategies, with a summary of notations shown in Table 3.

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).*

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 [35, 38] 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.

6. EXECUTION

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

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.

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.

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.

7. EVALUATION

This section evaluates the techniques described in this paper. First, we compare our TUPLEWARE prototype in a distributed setting against Hadoop and Spark (Section 7.1) and on a single machine against HyPer, MonetDB, and Spark (Section 7.2). We then provide a detailed performance breakdown (Section 7.3) to measure the impact of each optimization from Section 5 in realistic scenarios, further isolating their effects in detailed microbenchmarks (Section 7.4). 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 × 320GB SSDs, and 10 Gigabit*4 Ethernet; or (2) `m3.xlarge` instances with 4 vCPUs, 15GB RAM, and 2 × 40GB SSDs.

7.1 Distributed Benchmarks

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.

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 4: Distributed benchmark runtimes (seconds).

7.1.1 Workloads and Data

We implemented a version of four common ML tasks for each system without using any specialized libraries (e.g., Mahout [4], MLlib [6], BLAS [2]), 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, 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.

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.

7.1.2 Discussion

As shown in Table 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 4) and employing low-level code generation optimizations (Section 5). 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 total distinct pages (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]Hadoop is disk-based, so the cache cannot be warmed up.

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 5: Single node benchmark runtimes (seconds).

7.2 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 5.1, HyPer compiles SQL queries using the pipeline strategy, whereas MonetDB implements the operator-at-a-time strategy.

7.2.1 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 [8] 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 [9].

Million Song Dataset: We used a randomly sampled 1GB subset of the Million Song Dataset [12] 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 the focus of this paper 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 paper.

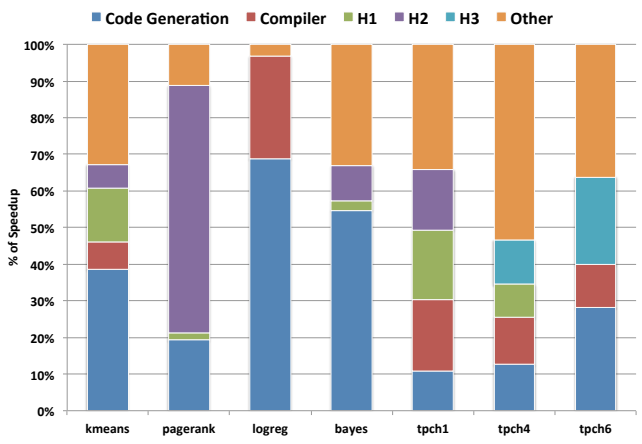


Figure 3: A performance breakdown with percentage speedups achieved by TUPLEWARE over Spark from Section 7.2.

7.2.2 *Discussion*

As shown in Table 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 (Section 7.3).

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.

7.3 Performance Breakdown

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 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 5. To derive the absolute time saved by a particular optimization, one can take the percentage from Figure 3 and multiply this value by Spark's runtime in Table 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 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.

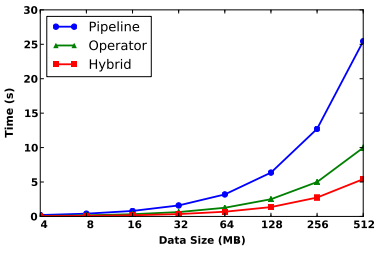
Optimizations: All of the heuristics described in Section 5 combine high-level semantic information about the workflow with low-level UDF statistics gathered by introspecting LLVM IR in order to further improve the performance of the generated code. As shown in Figure 3, these optimizations offer additional 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 4). 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.

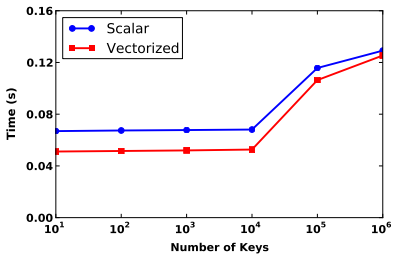
7.4 Microbenchmarks

As previously described, our proposed heuristics (Section 5) use 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 heuristic. All microbenchmarks were implemented in C++, compiled with Clang 3.4, and run on a single `c3.8xlarge` instance.

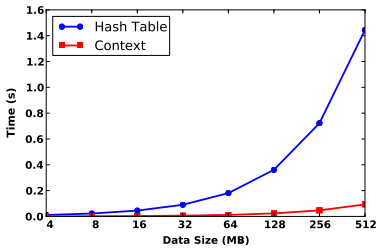
Heuristic 1a: We compared our hybrid strategy to the pipeline and operator-at-a-time strategies using the previously described k-means task. We tested each strategy with varying data sizes and



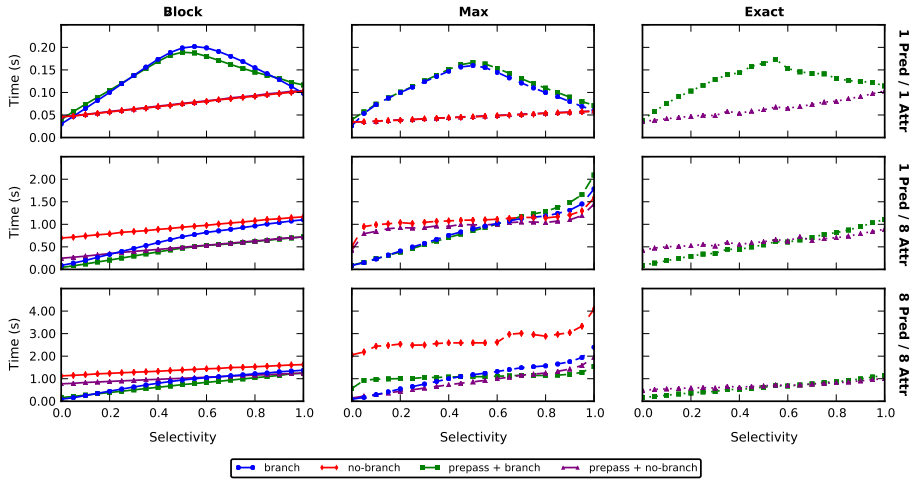
(a) Heuristic 1a



(b) Heuristic 1b



(c) Heuristic 2



(d) Heuristic 3

Figure 4: Heuristic microbenchmarks.

show the results in Figure 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× 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 ($\mathrm{mod}10$). Figure 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 4c shows that Context variables can improve performance by as much as 16×.

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 graphs represents different numbers of predicates and tuple attributes, and each column corresponds to a result allocation strategy (Section 5.3.2). For each graph, we show all possible predicate evaluation strategies (Section 5.3.1). 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 (Section 5.3.3) correctly chose the strategy combination with the lowest runtime.

8. RELATED WORK

Programming Model: Numerous extensions have been proposed to support iteration and shared state within MapReduce [14, 21, 10], and some projects (e.g., SystemML [22]) go a step further by providing a high-level language that is translated into MapReduce tasks. Conversely, TUPLEWARE natively integrates iterations and shared state to support this functionality without sacrificing low-level optimization potential.

DryadLINQ [43] is similar in spirit to TUPLEWARE's frontend and allows users to perform relational transformations directly in any .NET host language. Compared to TUPLEWARE, though, DryadLINQ cannot easily express updates to shared state and requires an external driver program for iterative queries, which precludes any cross-iteration optimizations.

Scope [17] provides a declarative scripting language that is translated into distributed programs for deployment in a cluster. However, Scope primarily focuses on SQL-like queries against massive datasets rather than supporting complex analytics workflows.

TUPLEWARE also has commonalities with the programming models proposed by Spark [44] and Stratosphere [41]. 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. In addition

to these more traditional techniques, TUPLEWARE's algebra is designed specifically to enable low-level optimizations that target the underlying hardware and efficiently support distributed shared state.

Code Generation: Code generation for query evaluation was proposed as early as System R [11], but this technique has recently gained popularity as a means to improve query performance for in-memory DBMSs [32, 26]. Both HyPer [24] and Vectorwise [47] propose different optimization strategies for query compilation, but these systems focus on SQL and do not optimize for UDFs. LegoBase [25] includes a query engine written in Scala that generates specialized C code and allows for continuous optimization, but LegoBase also concentrates on SQL and does not consider UDFs.

DryadLINQ compiles user-defined workflows using the .NET framework but applies only traditional high-level optimizations. Similarly, Tenzing [18] and Impala [42] are SQL compilation engines that also focus on simple queries over large datasets.

OptiML [39] offers a Scala-embedded, domain-specific language used to generate execution code that targets specialized hardware (e.g., GPUs) on a single machine. On the other hand, TUPLEWARE provides a general, language-agnostic frontend and compiles UDF workflows into LLVM-based distributed programs for deployment in a cluster.

Single Node Frameworks: Phoenix++ [40] and BID Data Suite [16] are high performance frameworks that target complex analytics, but they cannot scale to multiple nodes or beyond small datasets. Many scientific computing languages (e.g., R [7], Matlab [5]) have these same limitations. More specialized frameworks (e.g., Hogwild! [33], DimmWitted [45]) provide optimized implementations of specific algorithms on a single node, whereas TUPLEWARE is intended for general workflows in a distributed setting.

9. CONCLUSION

Complex analytics tasks have become commonplace for a wide variety of users. In this paper, we described a novel architecture for compiling UDF-centric workflows, and we believe that this work opens the door for an entire new breed of optimizations that consider data statistics, UDF characteristics, and the underlying hardware to better optimize generated code. Our experiments demonstrated that our TUPLEWARE prototype can achieve orders-of-magnitude speedups over alternative systems and show that the proposed heuristics can further improve performance for complex analytics tasks.

10. ACKNOWLEDGMENTS

This research is funded in part by the Intel Science and Technology Center for Big Data, the NSF CAREER Award IIS-1453171, the Air Force YIP AWARD FA9550-15-1-0144, and gifts from SAP, Mellanox, Amazon, and Oracle. We would like to thank our collaborators from the Parallel Computing Lab at Intel, especially Nadathur Satish, for their valuable input. We would also like to thank the Spark team from the AMPLab at UC Berkeley, particularly Evan Sparks, for their help with our benchmarks.

11. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org.
[2] BLAS. http://netlib.org/blas/.
[3] Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. http://agner.org/optimize/instruction_tables.pdf.
[4] Mahout. http://mahout.apache.org/.
[5] Matlab. http://mathworks.com/products/matlab/.
[6] MLlib. http://spark.apache.org/mllib/.
[7] R Project. http://r-project.org/.
[8] UK Crime Dataset. http://data.police.uk/.
[9] Wikipedia Webgraph. http://dumps.wikimedia.org/.
[10] S. Alsubaiee et al. ASTERIX: An Open Source System for "Big Data" Management and Analysis. In *VLDB*, pages 1898–1901, 2012.
[11] M. M. Astrahan et al. System R: A Relational Data Base Management System. *IEEE Computer*, pages 42–48, 1979.
[12] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and P. Lamere. The Million Song Dataset. In *ISMIR*, pages 591–596, 2011.
[13] V. R. Borkar et al. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. Bull.*, pages 24–32, 2012.
[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB*, pages 285–296, 2010.
[15] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-intensive Programs. In *WebDB*, pages 10:1–10:6, 2010.
[16] J. Canny and H. Zhao. Big Data Analytics with Small Footprint: Squaring the Cloud. In *KDD*, pages 95–103, 2013.
[17] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB*, pages 1265–1276, 2008.
[18] B. Chattopadhyay et al. Tenzing A SQL Implementation On The MapReduce Framework. In *VLDB*, pages 1318–1327, 2011.
[19] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. In *VLDB*, pages 1802–1813, 2012.
[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
[21] J. Ekanayake et al. Twister: A Runtime for Iterative MapReduce. In *HPDC*, pages 810–818, 2010.
[22] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, pages 231–242, 2011.
[23] B. Graham and M. R. Rangaswami. Do You Hadoop? A Survey of Big Data Practitioners. 2013.
[24] A. Kemper et al. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Eng. Bull.*, pages 41–47, 2013.
[25] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. In *VLDB*, pages 853–864, 2014.
[26] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, pages 613–624, 2010.
[27] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, pages 558–565, 1978.
[28] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–88, 2004.
[29] A. Nadkarni and L. DuBois. Trends in Enterprise Hadoop Deployments. 2013.
[30] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *VLDB*, pages 539–550, 2011.
[31] O. Polychroniou and K. A. Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *DaMoN*, pages 6:1–6:6, 2013.
[32] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, page 23, 2006.
[33] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*, pages 693–701, 2011.
[34] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. In *VLDB*, pages 853–864, 2013.
[35] K. A. Ross. Conjunctive Selection Conditions in Main Memory. In *PODS*, pages 109–120, 2002.
[36] K. A. Ross. Efficient Hash Probes on Modern Processors. In *ICDE*, pages 1297–1301, 2007.
[37] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody Ever Got Fired for Using Hadoop on a Cluster. In *HotCDP*, pages 2:1–2:5, 2012.
[38] B. Răducanu, P. Boncz, and M. Zukowski. Micro Adaptivity in Vectorwise. In *SIGMOD*, pages 1231–1242, 2013.
[39] A. K. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*, pages 609–616, 2011.
[40] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *MapReduce*, pages 9–16, 2011.
[41] K. Tzoumas et al. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. In *ICDE*, pages 1292–1295, 2013.
[42] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, pages 31–37, 2014.
[43] Y. Yu et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.
[44] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 15–28, 2012.
[45] C. Zhang and C. Re. DimmWitted: A Study of Main-Memory Statistical Analytics. In *VLDB*, pages 1283–1294, 2014.
[46] M. Zukowski, P. A. Boncz, N. Nes, and S. Hman. MonetDB/X100 - A DBMS in the CPU Cache. *IEEE Data Eng. Bull.*, pages 17–22, 2005.
[47] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*, pages 1349–1350, 2012.