Abstract

There is a fundamental discrepancy between the targeted and actual users of current analytics frameworks. Most systems are designed for the data and infrastructure of the Googles and Facebooks of the world—petabytes of data distributed across large cloud deployments consisting of thousands of cheap commodity machines. Yet, the vast majority of users operate clusters ranging from a few to a few dozen nodes, analyze relatively small datasets of up to several terabytes, and perform primarily compute-intensive operations. Targeting these users fundamentally changes the way we should build analytics systems.

This paper describes the design of Tupleware, a new system specifically aimed at the challenges faced by the typical user. Tupleware’s architecture brings together ideas from the database, compiler, and programming languages communities to create a powerful end-to-end solution for data analysis. We propose novel techniques that consider the data, computations, and hardware together to achieve maximum performance on a case-by-case basis. Our experimental evaluation quantifies the impact of our novel techniques and shows orders of magnitude performance improvement over alternative systems.

1 Introduction

The countless possibilities of advanced analytics have elicited more interest than ever in “big data” from companies and researchers alike. Still, current analytics frameworks like Hadoop [11] and Spark [33] are designed specifically to meet the needs of giant Internet companies; they are built to process petabytes of data in cloud deployments consisting of thousands of cheap commodity machines. The widespread popularity of advanced analytics, though, has drastically changed the typical use cases. Nowadays, these frameworks are usually deployed on smaller clusters with more reliable hardware, rather than large cloud deployments. In fact, it was reported in 2011 that the median Hadoop installation was smaller than 30 nodes [3]. Furthermore, common MapReduce-style jobs, even at companies as big as Facebook, rarely exceed a few terabytes in size [8][18], making it possible to fit all data in memory on small clusters for these workloads.

Supporting the typical user, then, fundamentally changes the way we should design analytics tools. Current analytics frameworks are built around the major bottlenecks of large cloud deployments, in which data movement (disk to machine and across the network) is the primary performance bottleneck, machines are slow, and failures are the norm [19]. Conversely, with smaller clusters ranging in size from a few to a few dozen nodes, failures are the exception. Most importantly, whereas single-node performance is largely irrelevant in cloud deployments, it can no longer be ignored when targeting small clusters.

In this paper we describe Tupleware, a new system designed for typical analytics workloads characterized by relatively small data and compute-intensive operations. Tupleware compiles workflows comprised of user-defined functions (UDFs) directly into a self-contained distributed executable, integrating the LLVM [27] compiler framework to provide a language-agnostic API. With Tupleware, we (1) address the unique frontend requirements of complex workflows, (2) apply low-level optimizations on a case-by-case basis by considering specific hardware features (e.g., SIMD vectorization, memory bandwidth), and (3) tailor the deployment architecture to more typical hardware configurations.

Our benchmarks, based on common machine learning tasks, demonstrate that our novel techniques achieve orders of magnitude performance improvements over other systems. In summary, we make the following contributions:

1. We present Tupleware, a general analytics system that considers the data, computations, and underlying hardware together in order to fully synthesize a self-contained and highly optimized distributed program.
2. We propose a new programming model founded in functional programming with monads, allowing for the concise expression of complex workflows while retaining strong optimization potential.
3. We describe a novel code generation strategy that applies optimizations on a case-by-case basis by examining the internals of UDFs.
4. We benchmark Tupleware using several common machine learning tasks and show speedups of up to three orders of magnitude over other systems.
Figure 1: An overview of Tupleware’s architecture, which depicts the three distinct yet interrelated components of the system: (1) frontend, (2) program synthesis, and (3) deployment.

2 System Overview

Tupleware is a distributed, in-memory analytics platform that targets complex computations such as machine learning (ML) and predictive modeling. The system architecture is shown in Figure 1 and is comprised of three distinct parts.

**Frontend:** Similar to other recent frameworks (e.g., Spark [38], Stratosphere [23], DryadLINQ [37]), Tupleware allows users to define complex workflows directly inside a host language by supplying UDFs to API operators like map and reduce. Our new algebra, based on the strong foundation of functional programming with monads, seeks a middle ground between flexibility and optimizability while also addressing the unique needs of ML algorithms. Furthermore, by leveraging the LLVM compiler framework, Tupleware’s frontend is language-agnostic, and users can choose from a wide variety of programming languages (visualized as the top boxes in Figure 1) with little associated overhead. We describe Tupleware’s algebra and API in Section 3.

**Program Synthesis:** When the user submits a job to Tupleware, the *Function Analyzer* examines each UDF to gather statistics for predicting execution behavior. The *Planner* then translates the workflow to an abstract plan and applies high-level optimizations. Finally, the *Code Generator* converts the plan into a self-contained distributed program and applies novel optimizations that specifically target the underlying hardware using the gathered UDF statistics. Program synthesis and the accompanying optimizations are explained in Sections 4-5.

**Deployment:** After Tupleware has generated the distributed program, the job is automatically scheduled, deployed, and executed on the cluster, depicted in Figure 1 as ten nodes (shown as boxes) each with four hyperthreads (circles inside the boxes). Tupleware utilizes a multitiered deployment setup, assigning specialized tasks to dedicated threads, and also takes unique approaches to memory management, load balancing, and recovery. We discuss all of these aspects of deployment further in Section 6.

3 Frontend

Ideally, developers want the ability to concisely express complex workflows in their language of choice without having to consider low-level optimizations or the intricacies of distributed execution. In this section, we describe how Tupleware addresses these points.

3.1 Background

The MapReduce [19] paradigm is a popular programming model for parallel data processing that consists of two primary operators: a *map* that applies a function to every key-value pair, and a *reduce* that aggregates values grouped by key. Many have criticized MapReduce [5], in particular for rejecting the advantages of high-level languages like SQL. However, SQL is unwieldy for expressing many classes of problems, including ML tasks. For instance, the SQL representation of one ML algorithm described in Section 7 required four levels of nested subqueries, compared to only a few short map and reduce operations.

Many ML algorithms are most naturally expressed iteratively, but neither MapReduce nor SQL effectively supports iteration. Unsurprisingly, a number of iterative extensions have been proposed [13, 21, 20], but users frequently need to rethink algorithmic structure to fit the supplied API, often sacrificing some efficiency in the process.

Furthermore, no existing framework incorporates an elegant and efficient solution for the key ingredient of ML algorithms, namely shared state. Many attempts to support distributed shared state within a MapReduce-style framework impose substantial restrictions on how and when global values could be used. For instance, the MapReduce-Update model [12] supplies traditional map and reduce functions with read-only copies of global state values that are recalculated during the update phase after each iteration, but this model is quite restrictive for diverse workflows. Similarly, Spark provides objects called Accumulators, which are only useful for simple count or sum aggregations on a single key, and their values cannot be accessed from within the workflow.
3.2 Programming Model

We therefore need a programming model that strikes a middle ground between the expressiveness of MapReduce and optimizability of SQL while also supporting the unique requirements of ML algorithms. Tupleware introduces a new algebra based on the foundation of functional programming with monads to address this challenge. We define this algebra on a data structure called a TupleSet, which encapsulates both the data and its associated state.

Definition 1 (TupleSet) A TupleSet $T$ 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 of key-value pairs.

In Tupleware’s algebra, operators describe how users can transform a TupleSet:

Definition 2 (Operator) An operator $O$ is a second-order function paired with a user defined first-order function $\lambda$ that takes zero or more TupleSets and produces an output TupleSet.

Tupleware’s programming model allows for automatic and efficient parallel data processing. As shown in Figure 2 each node in the cluster processes a disjoint subset of the data. However, unlike other paradigms, Tupleware’s API incorporates the notion of global state that is logically shared across all nodes.

3.3 Operator Types

We divide operators into four distinct types. Table 1 shows the most common Tupleware operators, as well as the signatures of their associated $\lambda$-functions. The $\lambda$-functions are supplied by the user and specify the workflow’s computation.

Relational: Relational operators include all of the traditional SQL transformations. For example, the user can perform a selection by passing a predicate UDF to the corresponding operator. As given in Table 1 the expected UDF signature has the form: $t \rightarrow b$ where $t \in R$ and $b$ is a Boolean value; that is, the user compiles a predicate using the set of operations $\{=, \neq, >, \geq, <, \leq\}$ that returns true if a given tuple $t$ of the incoming relation $R$ should be selected for the output relation $R'$ and false otherwise. Note that relational operators interact only with the relation $R$ of the TupleSet and cannot modify Context variables. Hence, the Planner described in Section 4.2 can perform the standard query optimization techniques (e.g., predicate pushdown, join reordering). Note, though, that operators such as $\theta$-join and union merge Context variables but do not change their values, performing SQL-style disambiguation of conflicting keys.

Apply: Apply operators invoke the supplied UDF on every tuple in the relation $R$. Tupleware’s API provides three apply operators: map, flatmap, and filter. The map operator requires a UDF that specifically produces a 1-to-1 mapping (i.e. the UDF takes one input tuple and must return exactly one output tuple). The flatmap operator takes a UDF that produces a 1-to-N mapping but is more difficult to optimize. The filter operator takes a UDF that produces a 1-to-0 mapping and is less restrictive than the relational selection operator, permitting arbitrary predicate logic. By distinguishing among these different types of apply operators, our programming model provides the system with additional information about the workflow, thereby allowing for greater optimization.

Aggregate: Aggregate operators perform an aggregation UDF on the relation $R$. Similar to Spark, Tupleware’s reduce operator expects a commutative and associative $\lambda$-function. These semantics allow for the efficient parallelization of computations like sum and count, which return an output relation $R'$ consisting of one or more aggregated values. Users can also specify a key function $\kappa$ that defines the group-by semantics for the aggregation. If no key function is provided, then the computation is a single-key reduce (i.e., all tuples have the same key). Additionally, Tupleware’s reduce operator can modify Context variables, as described in Section 3.3.

Control: So far, we have not specified how to load data, evaluate a workflow, or save the results. We assume that the user loads the data into the filesystem as a separate step and then specifies the workflow using the defined operators. As their names suggest, the evaluate and save
operators actually execute a workflow and store the results, respectively, returning a handle to the result as a new TupleSet that can then be used in a subsequent query. Notice, though, that this programming model can efficiently cache and reuse results across several computations. In order to support iterative workflows, which are common to ML algorithms, Tupleware also incorporates a loop operator. The loop operator models a tail recursive execution of the workflow while the supplied loop invariant holds, and the UDF has access to the Context for maintaining information such as iteration counters or convergence criteria. Finally, Tupleware’s algebra provides an update operator to allow direct modification of Context variables, which we discuss further in Section 3.4.

3.4 Context

Tupleware expresses shared state using monads, which are an elegant way to handle side effects in a functional language. Because of Tupleware’s parallel execution model, monads ensure correct concurrent updates to shared state values. As previously mentioned, this functionality is important for ML algorithms, which can represent models using Context variables.

Changes to the Context as part of a reduce must be commutative and associative. Conceptually, these updates are not directly applied, but rather added to an update set. After the operation completes, the deltas stored in the update sets are applied to the context and made visible. These semantics allow Tupleware to highly parallelize and optimize reduce λ-functions.

The update operator can directly modify Context variables because it executes logically in a single thread.

3.5 Language Integration

As mentioned previously, Tupleware allows users to write workflows and accompanying UDFs in any language with an LLVM compiler, even mixing languages to compose a single job. Presently, C/C++, Python, Julia, and R, and many other languages have LLVM backends.

The system exposes functionality in a given host language via a TupleSet wrapper that implements the Tupleware operator API (see Table 1). As long as the user adheres to the UDF contracts specified by the API, Tupleware guarantees correct parallel execution. A TupleSet’s Context also has a wrapper that provides special accessor and mutator primitives (e.g., get, increment, decrement). With the increasing popularity of LLVM, adding new languages is as simple as writing a wrapper to implement Tupleware’s API.

3.6 Example

Figure 3 shows a Python implementation of the k-means clustering algorithm using Tupleware’s API. K-means is an iterative ML algorithm that classifies each input data item into one of k clusters. In the example, the driver function kmeans defines the workflow using the five specified UDFs, where t1 is an input tuple, t2 is an output tuple, and c is the Context. Note that unlike other approaches, Tupleware can store the cluster centroids as Context variables, thereby providing several optimization opportunities (discussed further in Section 5).
| Function | Type  | Vectorizable | Compute Time (Predicted) | Compute Time (Actual) | Load Time |
|----------|-------|--------------|--------------------------|-----------------------|-----------|
| distance | map   | yes          | 29                       | 32                    | 3.75      |
| minimum  | map   | no           | 17                       | 15                    | 5.62      |
| reassign | reduce| no           | 15                       | 14                    | 4.22      |
| recompute| update| no           | 21                       | 23                    | 0         |

Table 2: Function statistics for the k-means algorithm gathered by the Function Analyzer.

**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 $8 \times 32$-bit floating-point values, offering a potential $8 \times$ speedup. In the k-means example, only the *distance* UDF is vectorizable, as shown in Table 2.

**Compute Time:** One metric for UDF complexity is the number of CPU cycles spent on computation. CPI measurements [4] 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. However, Table 2 shows that these predictions typically differ from the actual 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 operands for a particular UDF faster than the CPU can process them, then the UDF is referred to as *compute-bound*; conversely, if the memory controller cannot provide operands fast enough, then the CPU becomes starved and the UDF is referred to as *memory-bound*. Load time is given by:

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

For example, the load time for the *distance* UDF as shown in Table 2 computed on 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: $3.75 \text{ cycles} = \frac{2.8 \times (2 \times 4 \times B)}{5.97 \times 10^9}$.  

### 4.2 Planner

Next, the Planner converts a user’s workflow into an abstract plan, which is a logical representation of the job. Tupleware’s frontend supplies the Planner with additional information about the workflow, and the system combines this knowledge with data statistics to apply high-level optimizations (e.g., predicate pushdown, join reordering). Additionally, the purely functional programming model allows for the integration of other optimizations from the programming language community. In this regard, we simply apply known techniques, instead focusing on low-level code generation optimizations described in Section 5.

### 4.3 Code Generator

**Code generation** is the process by which compilers translate a high-level language (e.g., Tupleware’s algebra) into an optimized low-level form (e.g., LLVM). As other work has shown [26], SQL query compilation techniques can harness the full potential of the underlying hardware, and Tupleware extends these techniques by applying them to the domain of complex analytics.

The Code Generator translates the abstract plan produced by the Planner into a self-contained distributed program and uses UDF statistics gathered by the Function Analyzer to apply low-level optimizations tailored to the underlying hardware. We discuss these novel optimizations in Section 5.3.

As part of the translation process, the Code Generator produces all of the data structure, control flow, synchronization, and communication code necessary to form a complete distributed program. Unlike other systems that use interpreted execution models, Volcano-style iterators, or remote procedure calls, Tupleware eliminates much associated overhead by compiling in these mechanisms. Additionally, systems that treat UDFs as black boxes need to rely on external function calls, but Tupleware can inline UDFs to seamlessly merge them with the accompanying support code, providing substantial performance speedups.

### 5 Optimizations

Program synthesis involves a wide range of optimizations that occur on both a logical and physical level. We divide these optimizations into three categories.

**DBMS-only:** As described in Section 4.2, Tupleware utilizes well-known query optimization techniques, including predicate pushdown and join reordering. These DBMS-only optimizations rely on metadata and high-level language semantics, information that is unavailable to compilers.

**Compiler-only:** Section 4.3 explains the code generation optimizations that Tupleware leverages, including SIMD vectorization and function inlining. These compiler-only optimizations occur at a much lower level than DBMSs typically consider.

**DBMS & Compiler:** Some systems incorporate DBMS and compiler optimizations separately, and Sections 5.1 & 5.2 describe two such approaches: the (1) *pipeline* and (2) *operator-at-a-time* strategies. On the other hand, Tupleware combines a high-level algebra and data statistics with the ability to generate code, enabling optimizations that would be impossible for either a DBMS or compiler alone. Section 5.3 describes this novel *adaptive strategy*, which applies these hybrid DBMS and compiler optimizations on a case-by-case basis. To illustrate each of these strategies, we again reference the k-means algorithm from Section 5.6.
5.1 Pipeline Strategy

The pipeline strategy [29] aims to maximize data locality by performing as many sequential operations as possible per tuple. Operations referred to as pipeline breakers force the materialization of intermediate results. For example, a reduce requires an aggregation before advancing to the next phase, whereas all consecutive maps can be pipelined. Algorithm 1 shows the pipeline approach to k-means.

Algorithm 1 Pipeline k-means.

\begin{verbatim}
Data[N]
while !converged do
  for i = 1 : N do
    dist ← distance(data_i)
    min ← minimum(dist)
  end for
  reassign(min)
end while
\end{verbatim}

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.

5.2 Operator-at-a-time Strategy

The operator-at-a-time strategy [40] performs a single operation at a time for all tuples. This bulk processing approach maximizes instruction locality and opportunities for SIMD vectorization. Algorithm 2 shows the operator-at-a-time approach to k-means.

Algorithm 2 Operator-at-a-time k-means.

\begin{verbatim}
Data[N], Dist[N], Min[N]
while !converged do
  for i = 1 : N do
    dist ← distance(data_i)
  end for
  for i = 1 : N do
    min ← minimum(dist_i)
  end for
  for i = 1 : N do
    reassign(min_i)
  end for
end while
\end{verbatim}

The operator-at-a-time strategy, however, requires materialization of intermediate results between each operator, resulting in poor data locality. A tiled variant of this strategy [41] performs each operation on a cache-resident subset of the data, thus reducing materialization costs and limiting data transfer to the CPU.

5.3 Adaptive Strategy

In Sections 5.1 and 5.2, we described two different optimization strategies for code generation. While each of these approaches has definite advantages in certain situations, they also possess inherent flaws that prevent their universal applicability. The pipeline strategy boasts excellent data locality but severely limits the advantages of bulk processing, including SIMD vectorization and instruction locality. On the other hand, the operator-at-a-time strategy benefits greatly from bulk processing but fails to consider data locality, and even the cache-aware tiled variant suffers from the need to materialize intermediate results between operator calls.

Systems that dogmatically adhere to one or the other will necessarily generate suboptimal code in many situations. A combination of execution strategies, then, is often most sensible, but traditional systems cannot make case-by-case decisions because they regard UDFs as black boxes. Since Tupleware has the ability to introspect UDFs, we propose a novel adaptive strategy that considers the data, computations, and underlying hardware together to generate optimal code in each individual situation. Furthermore, our strategy can often leverage this knowledge to utilize special data structures when appropriate, offering even greater performance enhancements. We discuss only optimizations for the map and reduce operators, though we are currently developing optimizations for other operators as well.

5.3.1 Map

By default, we group all consecutive maps into a single pipeline to maximize data locality. Our approach then examines each UDF for SIMD processing opportunities and partitions adjacent maps into vectorizable and non-vectorizable groups. Intermediate results are materialized between groups in cache-resident blocks. If the workflow contains no vectorizable UDFs, then the original single-pipeline structure is preserved.

The only exception to this rule arises when a group of one or more vectorizable maps 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 group of maps should remain as part of the original pipeline in order to benefit from data locality, since no additional performance increase can be achieved through vectorization.

Consider again the k-means algorithm. Given the statistics provided by the Function Analyzer in Table 2, we notice that the vectorizable distance UDF is a candidate for pipeline partitioning. However, since the UDF resides at the beginning of the pipeline, we must also ensure that the computation is not already memory-bound. In this case, we see that Compute Time > Load Time, so distance is compute-bound and should be split from the pipeline to yield code resembling Algorithm 3.

5.3.2 Reduce

Generally, a reduce involves maintaining a hash table to store keys and associated aggregates. Since hash ta-
Algorithm 3 Adaptive k-means.

```
Data[N], Dist[N]
while !converged do
  for i = 1 : N do
    disti ← distance(datai)
  end for
  for i = 1 : N do
    min ← minimum(disti)
    reassign(min)
  end for
  recompute()
end while
```

ble lookups require random memory accesses, reduce functions cannot be vectorized. However, the lookup is actually comprised of two distinct parts: the hash calculation and the memory fetch. The hash calculation can therefore be computed in parallel using SIMD registers, followed by serial execution of memory fetches.

In the case of single-key reduces, we utilize special reduction variables to enable the vectorization of aggregation UDFs that are both commutative and associative (e.g., sum). A reduction variable is transformed into a vector of partial aggregates that are then recombined at the end of the loop to derive the final result. This optimization completely avoids the usual process of constructing a hash table during the reduce stage.

Additionally, we add reduces to the end of map pipelines to benefit from data locality, as shown by the reassign function in Algorithm 3.

6 Deployment

After program synthesis (Section 4), the system now has a self-contained distributed executable. Each distributed executable contains all necessary communication and synchronization code, avoiding any overhead associated with external function calls. Tupleware takes a multi-tiered approach to distributed deployment, as shown in Figure 1. The system dedicates 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 the data among nodes and supervising the current stage of the execution. In addition, we dedicate one thread per node as a Local Manager (LM). The Local Manager is responsible for the fine-grained management of the local shared memory, as well as for transferring data between machines. The Local Manager is also responsible for actually deploying compiled programs and does so by spawning new executor threads (E), which actually execute the previously compiled program. During execution, these threads request data from the LM in an asynchronous fashion, and the LM responds with the data and a location for the result.

6.1 Memory Management

Similar to DBMSs, Tupleware manages its own memory pool and tries to avoid memory allocations when possible. Therefore the Local Manager is responsible for keeping track of all active TupleSets and performing garbage collection when necessary. UDFs that allocate their own memory, though, are not managed by Tupleware’s garbage collector. In addition, we avoid unnecessary object creations or data copying. For instance, Tupleware often performs updates in-place if the data is not required in subsequent computations. Additionally, while the Local Manager is idle, it can reorganize and compact the data, as well as free blocks of data that have already been processed.

6.2 Load Balancing

Tupleware’s data request model is multitiered and pull-based, allowing for automatic load balancing with minimal overhead. Each Executor thread requests data in small cache-sized blocks from the LM, and each LM in turn requests larger blocks of data from the GM. All remote data requests occur asynchronously, and blocks are requested in advance to mask transfer latency.

6.3 Fault Tolerance

As our experiments demonstrate, Tupleware can process gigabytes of data with sub-second response times, suggesting that checkpointing would do more harm than good. Extremely long-running jobs on the order of hours or days, though, might benefit from intermediate result recoverability. In these cases, Tupleware performs simple k-safe checkpoint replication.

However, unlike other systems, Tupleware has a unique advantage: since we fully synthesize distributed programs, we can optionally add these recovery mechanisms on a case-by-case basis. If our previously described workflow analysis techniques determine that a particular job will have a long runtime, we combine that estimation with the probability of a failure (given our intimate knowledge of the underlying hardware) to decide whether to include recovery code.

7 Evaluation

Our evaluation compares Tupleware’s performance to alternative systems in Section 7.1, provides a performance breakdown in Section 7.2, analyzes the effects of our optimizations in Section 7.3, and demonstrates the system’s scalability in Section 7.4.

7.1 System Benchmarks

To assess the overall performance of our system, we evaluate Tupleware against three widely-used analytics platforms (Hadoop, Spark, and a commercial column DBMS System X) using four common ML tasks.
7.1 Setup
We benchmark Tupleware in both a distributed cluster and single-machine environment to demonstrate its versatility. For the distributed case, we compare Tupleware against Spark and Hadoop. For the single-machine benchmarks, we compare it against Spark, as well as a commercial column DBMS System X.

To execute iterative algorithms on System X, we iteratively call a SQL stored procedure using a JDBC driver program and perform as much of the computation as possible inside the DBMS. Unless stated otherwise, we used c3.8xlarge instances on Amazon EC2. These instances have Intel E5-2680v2 processors (10 cores, 25MB Cache), 60 GB RAM, 2 × 320 SSDs, and are connected with 10 Gigabit*4 Ethernet. The distributed setup consists of 8×c3.8xlarge instances.

In all cases except Hadoop, we record the total runtime of each algorithm after the input data has been loaded into memory and parsed, with the caches warmed up. For all iterative algorithms, we report the total time to complete 20 iterations of the algorithm.

7.1.2 Workload and Data
For all systems, we implemented a consistent version of each ML task using the same algorithm with a fixed number of iterations. We used a combination of real and synthetic datasets to test across a wide range of data characteristics (e.g., size, dimensionality, skew). For all distributed benchmarks, we generated datasets of 1, 10, and 100GB in size. We benchmarked four ML algorithms.

**K-means**: As described in Section 3.6, k-means is an iterative clustering algorithm that partitions a dataset into k clusters. To evaluate the performance of each system in a distributed setup, we generated a dataset from three distinct means and ran the algorithm on each system. To analyze single node performance, we ran k-means using a 70MB synthetic dataset.

**Logistic Regression**: Logistic regression attempts to find a hyperplane \( w \) that best separates two classes of data by iteratively computing the gradient and updating the parameters of \( w \). We implemented logistic regression with gradient descent on generated data with 1024 features. We also ran the algorithm using a 10MB subset of the Million Song Dataset [11] in the single node setup. The dataset contains 26,330 data elements, each with 90 dimensions denoting various audio features. The algorithm tries to predict whether a given song was produced in 2006 or 2007 based upon these features.

**Linear Regression**: Linear regression produces a model by fitting a linear equation to a set of observed data points. The distributed experiment builds the model by iteratively computing a gradient and updating the weights for a synthetic dataset. We also ran the algorithm on a single node using the same 10MB subset of the Million Song Dataset.

**Naive Bayes**: A naive Bayes classifier is a conditional model that uses feature independence assumptions to assign class labels. Naive Bayes classifiers are extremely popular and are used for a wide variety of tasks, such as spam filtering, text classification, and sentiment analysis. For the distributed setup, we trained a naive Bayes classifier on a generated dataset with 128 features and 10 possible labels. Using a larger subset of the Million Song Dataset, we ran the same naive Bayes algorithm on a single machine in order to predict the release year of a given song. The 100MB subset contains 90 audio features for 288,070 songs, and we preprocessed the data by mapping continuous values to categorical bins.
7.1.3 Discussion

We now discuss the performance of Tupleware against Spark and Hadoop on a small cluster of 8 nodes, and against Spark and System X on a single node. The results are included in Figures 4, 5, and 6. Overall, our benchmarks show speedups between 203 – 6334 × over Hadoop, 10 – 341 × over Spark, and 57 – 1790 × over System X.

Hadoop encounters substantial I/O overhead when materializing intermediate results to disk between iterations. On the other hand, Tupleware intelligently caches intermediate results in memory and focuses on compute-bound workloads. Furthermore, Hadoop’s API is not intended for complex analytics and makes optimization difficult.

Spark improves upon Hadoop by storing the working set in memory and offering a richer API. We therefore measure the greatest speedups for Spark over Hadoop with iterative tasks, whereas the non-iterative naive Bayes runtimes for these two systems are much more similar. Tupleware can achieve an additional 1 to 2 orders of magnitude improvement over Spark by optimizing at a low level for the computation bottleneck.

System X lacks native support for many fundamentals of ML algorithms, in particular iteration. Hence, the SQL implementations incurred major performance penalties for k-means and the regressions but not for naive Bayes.

7.2 Performance Breakdown

No single optimization explains the entire performance gap between Tupleware and other systems. Only the combination of the frontend, program synthesis, and deployment techniques allows Tupleware to achieve speedups of several orders of magnitude. In order to explain these speedups, we provide a performance breakdown to quantify the benefits of Tupleware’s distinguishing features. Each experiment isolates one particular feature to measure the idealized contribution to the overall speedup. This is currently a preliminary breakdown, and we are still investigating other factors.

C++ vs Java: We compared the average runtime for a single iteration of each ML algorithm from Section 7.1 implemented in both C++ and Java. While C++ offers significantly lower level control, the choice of C++ over Java has relatively little overall impact. Our experiments achieved a speedup of up to 2.5×.

Inlining: Compiling queries into executables allows Tupleware to inline functions instead of performing external function calls. This technique eliminates significant overhead associated with UDF-centric workloads, offering around a 2.5× speedup.

No Iterators: Volcano-style iterators recursively call a next function for each operator in the workflow. This pattern is easily generalized and simple to implement, hence its extensive adoption by a wide range of systems, but has substantial performance penalties. By dynamically generating code, Tupleware eliminates all overhead associated with iterator-based execution, which explains a factor of up to 4× in our experiments.

Vectorization: As mentioned in Section 4.1, SIMD vectorization offers significant performance benefits, and Tupleware can automatically take advantage of this hardware feature. On the tested hardware, we can see up to an 8× improvement compared to the scalar version. Even greater speedups are possible with currently available 512- or 1024-bit SIMD registers.

Design & Optimizations: The remainder of the speedup can be attributed to a combination of intangible design differences (e.g., workflow compilation, deployment architecture) and novel optimization techniques. In Section 7.3, we examine the impact of vectorization opportunities made possible by our code generation techniques, as well as our other optimizations.

7.3 Microbenchmarks

In order to get a better understanding of our code generation techniques, we performed a series of microbenchmarks for (1) map optimizations, (2) single-key reduce optimizations, and (3) Context variables in aggregate UDFs.

Map Strategies: We compare our adaptive map optimization strategy to the pipeline, operator-at-a-time, and tiled operator-at-a-time strategies described in Sections 5.1 and 5.2. We implemented each strategy for k-means in C++, compiled with Clang 3.4, and ran 20 iterations using 70MB of input data on a single c3.8xlarge instance. The pipeline strategy benefits from excellent data locality by performing all operations consecutively for each data element. However, this approach prohibits any vectorization due to the fact that the minimum function cannot be vectorized. The operator-at-a-time strategy applies a single operator once to each data element, materializing results between operators. Using this approach, the distance function can be vectorized, but materializing intermediate results between subsequent operators incurs significant overhead. As demonstrated in Figure 8a, our
Figure 8: Tupleware microbenchmarks (a-c) and scalability benchmarks (d).

adaptive strategy performs better than the existing strategies due to the fact that it can take advantage of SIMD vectorization while also pipelining nonvectorizable operations for better data locality.

Reduction Variable: Based on knowledge about the entire workflow, Tupleware is able to generate code that uses a reduction variable rather than a hash table when performing a single-key aggregation, as described in Section 5.3. Our programming model allows for this optimization and permits SIMD vectorization for single-key aggregation UDFs. We performed a microbenchmark for a simple reduce with and without a reduction variable. Figure 8b shows the speedup of the reduction variable over the naive approach for a simple single-key reduce. This code generation technique yields a consistent speedup of around $6.5 \times$ for various data sizes, with a slight drop around 20MB because of the CPU cache size.

Context: Tupleware is able to highly optimize workflows that use Context variables due to the fact that the system knows about the types and sizes of all Context variables when the program is compiled. For updates to the Context, Tupleware is able to use direct indexing instead of hashing to perform aggregation. For example, instead of looking up and incrementing a value for a key in a hash table, Tupleware can directly index into an array and increment the corresponding value. This eliminates the need to perform a hash function for each key as well as search for the correct key once the bucket is found. Tupleware is also able to allocate fixed size containers for these Context variables, allowing the system to better utilize the CPU cache. To measure the impact of this optimization, we compare using hashing to direct indexing for storing a count of 10 distinct keys for datasets of varying size. As shown in Figure 8c, this technique improves performance by around $16 \times$.

7.4 Scalability Benchmarks

To show how Tupleware performs in clusters of various sizes, we run a weak scalability experiment where each machine processes 1GB of input data. Each node in the cluster is a m3.large instance with 2 virtual CPUs and 7.5GB of RAM. Figure 8d shows that Tupleware is able to adapt to different cluster sizes and scale linearly to larger clusters.

8 Related Work

Tupleware’s unique design allows the system to highly optimize complex analytics tasks. While other systems have looked at individual components, Tupleware collectively addresses how to (1) easily and concisely express complex analytics workflows, (2) synthesize self-contained distributed programs optimized at the hardware level, and (3) deploy tasks efficiently on various cluster sizes.

8.1 Programming Model

Numerous extensions have been proposed to support iteration and shared state within MapReduce [9, 13, 21], and some projects (e.g., SystemML [22]) go a step further by providing a high-level language that is translated to MapReduce tasks. Conversely, Tupleware natively integrates iterations and shared state to support this functionality without sacrificing low-level optimization potential. Other programming models, such as FlumeJava [16], Ciel [28], and Piccolo [30] lack the low-level optimization potential that Tupleware’s algebra provides.

DryadLINQ [37] is most similar to Tupleware’s frontend, as it allows the user to incorporate relational transformations into any .NET host language. The biggest difference between this framework and Tupleware is that DryadLINQ cannot easily express updates to shared state and requires a driver-program for iterations, which precludes many optimizations.

Tupleware also has commonalities with the programming models proposed by Spark [38] and Stratosphere [23]. These systems have taken steps in the right direction by providing richer APIs that supply the optimizer with additional information about the workflow, thus permitting standard high-level optimizations. In addition to these more traditional optimizations, Tupleware’s algebra is designed specifically to enable hardware-level
optimizations and efficiently handle distributed shared state.

8.2 Code Generation

Code generation for query evaluation was proposed as early as System R [10], but this technique has recently gained popularity as a means to improve query performance for in-memory DBMSs [31, 26]. Both HyPer [24] and VectorWise [41] propose different optimization strategies for query compilation, but these systems focus on SQL and do not optimize for UDFs. In fact, our experiments in Section 7.2 demonstrate that neither of these strategies is optimal for complex analytics workflows.

LegoBase [25] incorporates a query engine written in Scala that generates specialized C code, allowing for continuous optimization. Again, LegoBase concentrates on SQL and does not consider complex analytics or UDFs.

Scope [15] compiles workflows using the .NET framework. However, Scope primarily focuses on SQL-like queries against massive datasets rather than complex in-memory analytics and applies no hardware-level optimizations. Similarly, Tenzing [17] and Impala [2] are SQL compilation engines that also target simple queries on large datasets.

OptiML [33] offers a Scala-embedded, domain-specific language used to generate execution code that targets specialized hardware (e.g., GPUs) on a single machine. Tupleware on the other hand provides a general, language-agnostic frontend used to synthesize LLVM-based distributed executables for deployment in a cluster.

One key advantage of Tupleware is its ability to adapt to the features of the underlying hardware, including SIMD registers. Significant work has been done on leveraging SIMD instructions for query processing in traditional DBMSs [39, 36, 35], but the focus is on relational operations and does not consider UDFs.

8.3 Distribution Architecture

Hadoop [11] targets datasets many times larger than the available memory, while Tupleware focuses instead on in-memory processing for complex computations. Like Tupleware, Spark and Stratosphere also focus on distributed in-memory analytics, but their deployment architecture and iterator-based data processing model impose considerable overhead, as shown in Section 7.2.

BID Data Suite [14] and Phoenix [34] are high performance single-node frameworks targeting general analytics, but these systems cannot scale to multiple machines or beyond small datasets. Scientific computing languages like R [7] and Matlab [6] have these same limitations. More specialized systems (e.g., Hogwild! [32]) provide highly optimized implementations for specific algorithms on a single machine, whereas Tupleware is intended for general computations in a distributed environment.

9 Conclusion

Advanced analytics workloads have become commonplace for a wide variety of users. However, instead of targeting the hardware to which most of these users have access, existing analytics frameworks are designed exclusively for large cloud deployments with thousands of commodity machines. This paper described the design of Tupleware, a new analytics system geared towards the needs of typical users. Tupleware combines ideas from the database, compiler, and programming language communities to create a user-friendly yet highly efficient end-to-end solution for complex analytics. Our experiments demonstrated that Tupleware achieves remarkable speedups of 10 – 6000× over alternative systems.

References

[1] Apache hadoop. http://hadoop.apache.org
[2] Cloudera impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html
[3] Dbms2. http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/
[4] Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. http://www.agner.org/optimize/instruction_tables.pdf
[5] Mapreduce: A major step backwards. http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html
[6] Matlab. http://www.mathworks.com/products/matlab/
[7] R project. http://www.r-project.org/
[8] Nobody ever got fired for using hadoop on a cluster. In Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing, HotCDP ’12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.
[9] S. Alsubaiee, Y. Altowim, H. Altwajiry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, R. Grover, Z. Heilbron, Y.-S. 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.
[10] 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.
[11] T. Bertin-Mahieux, D. P. W. Ellis, B. Whitman, and F. Lamere. The million song dataset. In ISMIR, pages 591–596, 2011.
