Tupleware: Distributed Machine Learning on Small Clusters

Andrew Crotty, Alex Galakatos, Tim Kraska
Brown University
{crottyan, agg, kraskat}@cs.brown.edu

Abstract

There is a fundamental discrepancy between the targeted and actual users of current analytics frameworks. Most systems are designed for the challenges 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 in size, and perform primarily compute-intensive operations. Targeting these users fundamentally changes the way we should build analytics systems.

This paper describes our vision for the design of Tupleware, a new system specifically aimed at performing complex analytics (e.g., distributed machine learning) on small clusters. Tupleware's architecture brings together ideas from the database and compiler communities to create a powerful end-to-end solution for data analysis. Our preliminary results show orders of magnitude performance improvement over alternative systems.

1 Introduction

The growing prevalence of big data across all industries and sciences is causing a profound shift in the nature and scope of analytics. Increasingly complex computations such as machine learning (ML) are quickly becoming the norm. However, current analytics frameworks (e.g., Hadoop [1], Spark [36]) 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 clusters 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 [20, 28].

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 [15], and commonly analyzed behavioral data peaks at around 1TB [11]. Even companies as large as Facebook, Microsoft, and Yahoo! frequently perform ML tasks on datasets smaller than 100GB [32]. Rather, as users strive to extract more value than ever from their data, computational complexity becomes the true problem.

Targeting more complex workloads on smaller clusters fundamentally changes the way we should design analytics tools. Current frameworks disregard single-node performance and instead focus on the major challenges of large cloud deployments, in which data I/O is the primary bottleneck and failures are common [17]. In fact, as

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

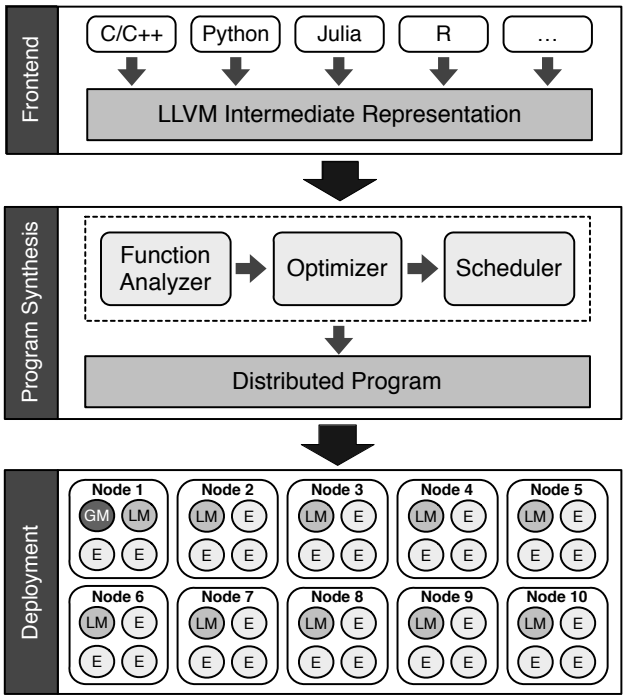


Figure 1: An overview of Tupleware's architecture, which depicts the three distinct yet interrelated components of the system: (1) frontend, (2) program synthesis, and (3) deployment.

our experiments show, it is surprising to see how inefficiently these frameworks utilize the available computing resources.

In this paper, we describe our vision for the design of Tupleware, a high-performance distributed system built specifically for performing complex analytics on small clusters. The key idea behind Tupleware is to integrate high-level DBMS optimizations with low-level compiler optimizations in order to best take advantage of the underlying hardware. Tupleware compiles workflows comprised of *user-defined functions* (UDFs) directly into a distributed program. More importantly, workflow compilation allows the system to apply optimizations on a case-by-case basis that consider properties about the data, computations, and hardware together.

2 System Overview

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

Frontend: Tupleware allows users to define ML workflows directly inside a host language by supplying UDFs to API operators such as 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 [25] 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 workflow to Tupleware, the *Function Analyzer* first examines each UDF to gather statistics for predicting execution behavior. The *Optimizer* then converts the workflow into

a self-contained distributed program and applies low-level optimizations that specifically target the underlying hardware using the previously gathered UDF statistics. Finally, the *Scheduler* plans how best to deploy the distributed program on the cluster given the available resources. The program synthesis process is explained in Section 4.

Deployment: After compiling the workflow, the distributed program is automatically deployed 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 deployment aspects further in Section 5.

3 Frontend

Ideally, developers want the ability to concisely express ML 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 Motivation

Designing the right abstraction for ML tasks that balances usability and functionality is a tricky endeavor. We believe that a good programming model for ML has three basic requirements.

Expressive & Optimizable: MapReduce [17] 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. Yet, many have criticized MapReduce, in particular for rejecting the advantages of high-level languages like SQL [4]. However, SQL is unwieldy for expressing ML workflows, resulting in convoluted queries that are difficult to understand and maintain. Other recent frameworks (e.g., Spark, Stratosphere [21], DryadLINQ [35]) have started to bridge the gap between expressiveness and optimizability by allowing users to easily compose arbitrary workflows of UDFs.

Iterations: Many ML algorithms are most naturally expressed iteratively, but neither MapReduce nor SQL effectively supports iteration [26, 16]. The most straightforward solution to this problem is to handle iterations via an external driver program. Both Spark and DryadLINQ take this approach, but the downside is that a completely independent job must be submitted for each iteration, making cross-iteration optimization difficult. In contrast, a number of iterative extensions to MapReduce have been proposed (e.g., Stratosphere, HaLoop [10], Twister [18]), but these approaches either lack low-level optimization potential or do not scale well.

Shared State: No existing framework incorporates an elegant and efficient solution for the key ingredient of ML algorithms: shared state. Many attempts to support distributed shared state within a MapReduce-style framework impose substantial restrictions on how and when programs can interact with global variables. For instance, the Iterative Map-Reduce-Update [9] model supplies traditional map and reduce functions with read-only copies of global state values that are recalculated during the update phase after each iteration. However, this paradigm is designed for iterative refinement algorithms and could be difficult to adapt to tasks that cannot be modeled as convex optimization problems (e.g., neural networks, maximum likelihood Gaussian mixtures). Furthermore, Iterative Map-Reduce-Update precludes ML algorithms that explore different synchronization patterns (e.g., Hogwild! [31]). Spark is another framework that supports shared state via objects called accumulators, which can be used only for simple count or sum aggregations on a single key, and their values cannot be read from within the workflow. Spark also provides broadcast variables that allow machines to cache large read-only values to avoid redistributing them for each task, but these values can never be updated. Therefore, broadcast variables cannot be used to represent ML models, which change frequently.

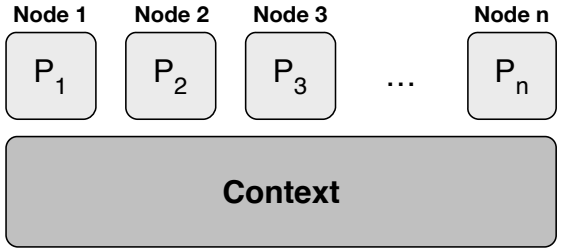


Figure 2: A visualization of a TupleSet's logical data model. Partitions $P_1, ..., P_n$ of the relation R are spread across n nodes, whereas the Context is logically shared across all nodes.

Type	Operator	λ-Function
Relational	selection$(T)(\lambda)$	$t \rightarrow b$
	projection$(T)(\lambda)$	$t \rightarrow t'$
	cartesian(T_1, T_2)	-
	θ-join$(T_1, T_2)(\lambda)$	$(t_1, t_2) \rightarrow b$
	union(T_1, T_2)	-
	difference(T_1, T_2)	-
Apply	map$(T)(\lambda)$	$(t, C) \rightarrow t'$
	flatmap$(T)(\lambda)$	$(t, C) \rightarrow \{t'\}$
	filter$(T)(\lambda)$	$(t, C) \rightarrow b$
Aggregate	reduce$(T)(\lambda)(\kappa?)$	$(t, C) \rightarrow (\Delta_\kappa, C')$
Control	load()	-
	evaluate(T)	-
	save(T)	-
	loop$(T)(\lambda)$	$C \rightarrow b$
	update$(T)(\lambda)$	$C \rightarrow C'$

Figure 3: A subset of TupleSet operators, showing their types and λ-function contracts.

3.2 Programming Model

Based on these requirements, we envision a new programming model for distributed ML that: (1) strikes a middle ground between the expressiveness of MapReduce and optimizability of SQL; (2) natively handles iterative workflows in order to optimize transparently across iterations; and (3) provides flexible shared state primitives with configurable synchronization patterns that can be directly accessed from within the workflow. Tupleware introduces an algebra based on the foundation of functional programming with monads to address all three of these points. We define this algebra on a data structure called a *TupleSet* comprised of a data relation and its associated *Context*, which is a dictionary of key-value pairs that stores the shared state. *Operators* define the different ways in which users can transform a TupleSet, returning a new TupleSet as output.

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. Figure 3 shows the most common Tupleware operators, most of which take as input one or more TupleSets T, as well as the signatures of their associated λ-functions. The λ-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 Figure 3, the expected UDF signature has the form: $t \rightarrow b$ where $t \in R$ and b is a Boolean value; that is, the user composes a (potentially compound) 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 shared state variables maintained by the Context C. Relational operators therefore introduce no dependencies, so we can perform the standard query optimization techniques (e.g., predicate pushdown, join reordering). Note, though, that operators such as *θ-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. 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-(0:1)* 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. Similar to Spark, Tupleware's *reduce* operator expects a commutative and associative λ-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 optionally specify a key function κ 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, reduce λ-functions can modify Context variables in different ways, which we describe further in Section 3.4.

Control: As their names suggest, the *load*, *evaluate*, and *save* operators actually load data into memory, execute a declared workflow, and persist the results to the specified location, respectively, returning a handle to the result as a new TupleSet that can then be used in a subsequent workflow. 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 that executes logically in a single thread and allows for direct modification of Context variables.

3.4 Context

Shared state is an essential component of distributed ML algorithms, which frequently involve iterative refinement of a global model. Tupleware expresses shared state using monads, which are an elegant way to handle side effects in a functional language. Monads impose a happened-before relation between operators; that is, an operator O that modifies Context variables referenced by another operator O' must be fully evaluated prior to evaluating O'.

Each type of operator has different rules that govern interaction with the Context. As previously mentioned, relational operators cannot access Context variables. Apply operators have read-only access to Context variables, allowing them to be fully parallelized. On the other hand, the update operator can directly modify Context variables because it executes logically in a single thread.

For aggregate operators, we define three different Context variable update patterns that can express a broad range of algorithms.

Parallel: Parallel updates 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 aggregated first locally and then globally, possibly using an aggregation tree to improve performance.

Synchronous: Synchronous updates require that each concurrent worker obtain an exclusive lock before modifying a Context variable. This pattern ensures that each worker always sees a consistent view of the shared state and that no updates are lost.

Asynchronous: Asynchronous updates allow workers to read from and write to a Context variable without first acquiring a lock. This pattern makes no guarantees about when changes to a Context variable will become visible to other concurrent workers and updates may be lost. Many algorithms can benefit from relaxing synchronization guarantees. For example, Hogwild! shows that asynchronous model updates can dramatically improve the performance of stochastic gradient descent. While asynchronous updates will occur nondeterministically

```
ATTR = 2                                  #2 attributes (x,y)
CENT = 3                                  #3 centroids
ITER = 20                                 #20 iterations

def kmeans(c):
 ts = TupleSet('data.csv', c)             #load file 'data.csv'
 ts = ts.map(distance)                    #get distance to each centroid
        .map(minimum)                     #find nearest centroid
        .reduce(reassign)                 #reassign to nearest centroid
        .update(recompute)                #recompute new centroids
        .loop(iterate)                    #perform 20 iterations
        .evaluate()                       #trigger computation
 return ts.context()['k']                 #return new centroids

def distance(t1, t2, c):
 t2.copy(t1, ATTR)                        #copy t1 attributes to t2
 for i in range(CENT):                    #for each centroid:
  t2[ATTR+i] = sqrt(sum(map(lambda        # compute and store distance
      m,n:(n-m)**2,c['k'][i],t1))

def minimum(t1, t2):
 t2.copy(t1, ATTR)                        #copy t1 attributes to t2
 m,n = min(m,n for n,m                    #find index of min distance
   in enumerate(t[:CENT]))
 t2[ATTR] = n                             #assign to nearest centroid

def reassign(t1, c):
 assign = t1[ATTR]                        #get centroid assignment
 for i in range(ATTR):                    #for each attribute:
  c['sum'][assign][i] += t1[i]            # compute sum for assign
 c['ct'][assign] += 1                     #increment count for assign

def recompute(c):
 for i in range(CENT):                    #for each centroid:
  for j in range(ATTR):                   # for each attribute:
   c['k'][i][j] =                         #  calculate average
      c['sum'][i][j]/c['ct'][i]

def iterate(c):
 c['iter'] += 1                           #increment iteration count
 return c['iter'] < ITER                  #check iteration count
```

Figure 4: A Tupleware implementation of k-means in Python.

during the execution of an individual operator, the final result of that operator should always be deterministic given valid inputs.

3.5 Language Integration

As mentioned previously, Tupleware allows users to compose workflows and accompanying UDFs in any language with an LLVM compiler. Presently, C/C++, Python, Julia, 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 Figure 3). 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, set). 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 4 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.

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

4 Program Synthesis

Once a user has submitted a workflow, the system (1) examines and records statistics about each UDF, (2) generates an abstract execution plan, and (3) translates the abstract plan into a distributed program. We refer to this entire process as *program synthesis*. In this section, we outline the different components that allow Tupleware to synthesize highly efficient distributed programs.

4.1 Function 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, Tupleware can look inside UDFs to gather statistics useful for optimizing workflows during the code generation process. The Function Analyzer examines the LLVM intermediate representation of each UDF to determine vectorizability, computation cycle estimates, and memory bandwidth predictions. As an example, Table 2 shows the UDF statistics for the k-means algorithm from Section 3.6.

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 $8\times$ speedup. In the k-means example, only the `distance` and `minimum` UDFs are vectorizable, as shown in Table 2.

Compute Time: One metric for UDF complexity is the number of CPU cycles spent on computation. 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. 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:

$$Load\ Time = \frac{Clock\ Speed \times Operand\ Size}{Bandwidth\ per\ Core} \tag{7}$$

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: $3.75\ cycles = \frac{2.8GHz\times(2\times4B)}{5.97GB/s}$.

4.2 Optimizer

Tupleware's optimizer can apply a broad range of optimizations that occur on both a logical and physical level. We divide these optimizations into three categories.

High-Level: Tupleware utilizes 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 Tupleware.

Low-Level: *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 [24], 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. As part of the translation process, Tupleware generates 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. Tupleware also gains many compiler optimizations (e.g., SIMD vectorization, function inlining) "for free" by compiling workflows, but these optimizations occur at a much lower level than DBMSs typically consider.

Hybrid: 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, Tupleware combines an optimizable high-level algebra and statistics gathered by the Function Analyzer with the ability to dynamically generate code, enabling optimizations that would be impossible for either a DBMS or compiler alone. In particular, we consider (1) high-level algebra semantics, (2) metadata, and (3) low-level UDF statistics together to synthesize optimal code on a case-by-case basis. For instance, we are investigating techniques to generate different code for selections based upon the estimated selectivity and computational complexity of predicates.

4.3 Scheduler

The Scheduler determines how best to deploy a job given the available computing resources and physical data layout in the cluster. Most importantly, the Scheduler takes into account the optimum amount of parallelization for a given operation in a workflow. Operations on smaller datasets, for instance, may not benefit from massive parallelization due to the associated deployment overhead. Additionally, the Scheduler considers data locality to minimize data transfer between nodes.

5 Deployment

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

Memory Management: Similar to DBMSs, Tupleware manages its own memory pool and tries to avoid memory allocations when possible. Therefore, the LM 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 LM is idle, it can reorganize and compact the data, as well as free blocks of data that have already been processed.

Load Balancing: Tupleware's data request model is multitiered and pull-based, allowing for automatic load balancing with minimal overhead. Each of the executor threads 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.

Fault Tolerance: As our experiments demonstrate, Tupleware can process gigabytes of data with subsecond 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 checkpointing code.

6 Evaluation

We compared an early Tupleware prototype against Hadoop 2.4.0 and Spark 1.0.1 using a small cluster with high-end hardware. This cluster consisted of 10×`c3.8xlarge` instances with Intel E5-2680v2 processors (10 cores, 25MB Cache), 60GB RAM, 2 × 320GB SSDs, and 10 Gigabit*4 Ethernet.

6.1 Workloads and Data

The benchmarks included five common ML tasks. We implemented a consistent version of each algorithm across all systems with a fixed number of iterations and used synthetic datasets in order to test across a range of data characteristics (e.g., size, dimensionality, skew). All tasks operated on datasets of 1, 10, and 100GB in size. We recorded the total runtime of each algorithm after the input data was loaded into memory and parsed except in the case of Hadoop, which had to read from and write to HDFS on every iteration. For all iterative algorithms, we report the total time taken to complete 20 iterations. We now describe each ML task.

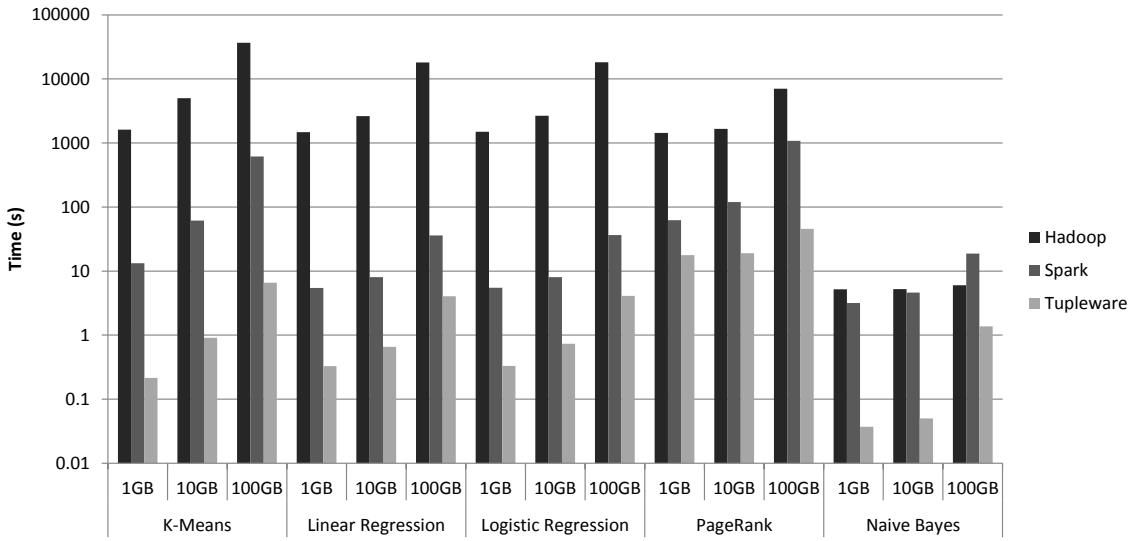
K-means: As described in Section 3.6, k-means is an iterative clustering algorithm that partitions a dataset into k clusters. Our test datasets were generated from four distinct centroids with a small amount of random noise.

Linear Regression: Linear regression produces a model by fitting a linear equation to a set of observed data points. We build the model using a parallelized batch gradient descent algorithm that computes updates locally on each worker from a disjoint subset of the dataset. These local updates are then averaged and applied to the model globally before the next iteration. The generated data had 1024 features.

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 also with batch gradient descent on generated data with 1024 features.

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. Based on the observation that important pages are likely to have more inbound links, search engines such as Google use the algorithm to order web search results.

Naive Bayes: A naive Bayes classifier is a conditional model that uses feature independence assumptions to assign class labels. Naive Bayes classifiers are used for a wide variety of tasks, such as spam filtering, text classification, and sentiment analysis. We trained a naive Bayes classifier on a generated dataset with 1024 features and 10 possible labels.



Algorithm	10×`c3.8xlarge`								
	Hadoop			**Spark**			**Tupleware**		
	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB
K-means	1621.36	5023.64	36818.58	13.243	61.528	614.355	0.214	0.910	6.594
Linear Regression	1479.7	2622.68	18149.8	5.433	8.013	36.101	0.327	0.657	4.046
Logistic Regression	1497.66	2665.7	18200.54	5.507	8.030	36.630	0.329	0.734	4.107
PageRank	1438.44	1666.8	7019.52	62.348	119.854	1076.834	17.797	19.027	45.543
Naive Bayes	5.18	5.228	6.025	3.162	4.622	18.703	0.037	0.050	1.371

Figure 5: Distributed benchmark results and runtimes (in seconds).

6.2 Discussion

As shown in Figure 5, Tupleware outperforms Hadoop by up to three orders of magnitude and Spark by up to two orders of magnitude for the tested ML tasks on a small cluster of high-end hardware.

Tupleware is able to significantly outperform Hadoop because of the substantial I/O overhead required for materializing intermediate results to HDFS between iterations. On the other hand, Tupleware is able to cache intermediate results in memory and performs hardware-level optimizations to improve CPU efficiency. For this reason, we measure the greatest speedups over Hadoop on k-means, linear and logistic regression, and PageRank, whereas the performance difference for Naive Bayes is not as pronounced. Furthermore, Hadoop's simple API is not intended for complex analytics and the system is not designed to optimize workflows for single-node performance.

Spark performs better than Hadoop for the iterative algorithms because it allows users to cache the working set in memory, eliminating the need to materialize intermediate results to disk. Additionally, Spark offers a richer API that allows the runtime to pipeline operators, further improving data locality and overall performance. For CPU-intensive ML tasks such as k-means, though, Tupleware is able to significantly outperform Spark by

synthesizing distributed programs and employing low-level code generation optimizations. Furthermore, Tupleware's unique frontend and Context variables described in Section 3 provide efficient, globally-distributed shared state for storing and updating a model. The benefit of storing the model as a Context variable is most noticeable in our naive Bayes experiment due to the fact that each machine can directly update a local copy of the model instead of performing an expensive shuffle operation.

7 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 a cluster.

7.1 Programming Model

Numerous extensions have been proposed to support iteration and shared state within MapReduce [10, 18, 7], and some projects (e.g., SystemML [19]) 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. Other programming models, such as FlumeJava [13], Ciel [27], and Piccolo [29] lack the low-level optimization potential that Tupleware's algebra provides.

DryadLINQ [35] 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 [12] 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 [36] and Stratosphere [21]. These systems have taken steps in the right direction by providing richer APIs that can supply an 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 low-level optimizations that target the underlying hardware, as well as to efficiently support distributed shared state.

7.2 Code Generation

Code generation for query evaluation was proposed as early as System R [8], but this technique has recently gained popularity as a means to improve query performance for in-memory DBMSs [30, 24]. Both HyPer [22] and VectorWise [37] propose different optimization strategies for query compilation, but these systems focus on SQL and do not optimize for UDFs. LegoBase [23] 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 ML or UDFs.

DryadLINQ compiles user-defined workflows into executables using the .NET framework but applies only traditional high-level optimizations. Similarly, Tenzing [14] and Impala [2] are SQL compilation engines that also focus on simple queries over 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.

7.3 Single-Node Frameworks

BID Data Suite [11] 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 [6] and Matlab [5] have these same limitations. More specialized systems (e.g., Hogwild! [31]) provide highly optimized implementations for specific algorithms on a single machine, whereas Tupleware is intended for general computations in a distributed environment.

8 Conclusion

Advanced analytics workloads, in particular distributed ML, have become commonplace for a wide range of users. However, instead of targeting the hardware to which most of these users have access, existing frameworks are designed primarily for large cloud deployments with thousands of commodity machines. This paper described our vision for Tupleware, a new analytics system geared towards compute-intensive, in-memory analytics on small clusters. Tupleware combines ideas from the database and compiler communities to create a user-friendly yet highly efficient end-to-end data analysis solution. Our preliminary experiments demonstrated that our approach can achieve speedups of up to several orders of magnitude for common ML tasks.

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] Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `http://www.agner.org/optimize/instruction_tables.pdf`.

[4] Mapreduce: A major step backwards. `http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html`.

[5] Matlab. `http://www.mathworks.com/products/matlab/`.

[6] R project. `http://www.r-project.org/`.

[7] S. Alsubaiee, Y. Altowim, H. Altwaijry, 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.

[8] 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.

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

[10] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[11] J. Canny and H. Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 95–103, New York, NY, USA, 2013. ACM.

[12] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[13] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 363–375. ACM, 2010.

[14] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. In *Proceedings of VLDB*, pages 1318–1327, 2011.

[15] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[16] T. Cruanes, B. Dageville, and B. Ghosh. Parallel sql execution in oracle 10g. SIGMOD '04, pages 850–854, New York, NY, USA, 2004. ACM.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[19] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[20] B. Graham and M. R. Rangaswami. Do you hadoop? a survey of big data practitioners. Sandy Hill Group, 2013.

[21] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J.-C. Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *ICDE*, pages 1292–1295. IEEE Computer Society, 2013.

[22] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid oltp & olap main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

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

[24] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[25] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[26] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Record*, 40(4):11–20, 2011.

[27] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[28] A. Nadkarni and L. DuBois. Trends in enterprise hadoop deployments. IDC, 2013.

[29] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[30] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using jvm. In *ICDE*, page 23, 2006.

[31] 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.

[32] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. 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.

[33] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[34] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[37] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.