Tupleware: “Big” Data, Big Analytics, Small Clusters

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, Stan Zdonik
Department of Computer Science, Brown University
{crottyan, agg, kayhan, kraskat, ugur, sbz}@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—processing petabytes of data distributed across large cloud deployments consisting of thousands of cheap commodity machines. Yet, the vast majority of users analyze relatively small datasets of up to several terabytes in size, perform primarily compute-intensive operations, and operate clusters ranging from only a few to a few dozen nodes. 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 complex analytics 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 that compiles workflows of user-defined functions into distributed programs. Our preliminary results show performance improvements of up to three orders of magnitude over alternative systems.

1. INTRODUCTION

Current analytics frameworks (e.g., Hadoop [1], Spark [43]) are designed to process massive datasets distributed across huge clusters. These assumptions represent the problems faced by giant Internet companies but neglect the needs of the typical user. Instead, we argue that typical users analyze datasets that are not that big, perform analytics tasks that are becoming increasingly complex, and utilize clusters that are much smaller than the infrastructures targeted by existing systems.

(1) “Big” Data: The ubiquity of big data is greatly exaggerated. For the vast majority of users, physical data size is rarely the issue. The typical Cloudera customer, for instance, seldom analyzes datasets that exceed a few terabytes in size [16], and even companies as large as Facebook, Microsoft, and Yahoo! frequently run analytics jobs on datasets smaller than 100GB [38]. Additionally, sampling techniques that further reduce the size of the input dataset are extremely common, since they help to improve performance and avoid model overfitting.

This article is published under a Creative Commons Attribution License(http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well as allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2015.
7th Biennial Conference on Innovative Data Systems Research (CIDR ‘15) January 4-7, 2015, Asilomar, California, USA.

(2) Big Analytics: Complex tasks, ranging from machine learning (ML) to advanced statistics, have come to define the modern analytics landscape. As users constantly strive to extract more value than ever from their data, algorithms are becoming increasingly compute-intensive, and the actual processing becomes the true problem. These tasks are most naturally expressed as workflows of *user-defined functions* (UDFs), which can contain arbitrary logic. Furthermore, any effort that requires users to be bound to a single programming environment is unlikely to achieve widespread adoption. We therefore need a language-agnostic solution that will allow users to stay within their preferred computing environments where they are most productive while also optimizing specifically around the features of UDFs.

(3) Small Clusters: Most frameworks are built around the major bottlenecks of large cloud deployments, in which optimizing for data movement (e.g., disk I/O, network transfers) is the primary focus. Yet, the vast majority of users typically operate smaller, more powerful clusters ranging in size from a few to a few dozen nodes [30] that are easier to maintain with readily available vendor support [24]. These clusters generally have abundant amounts of RAM, making it feasible to keep even large datasets entirely in memory with just a few machines, and individual nodes often contain high-end multi-core processors with advanced features (e.g., SSE/AVX, DDIO). Furthermore, recent advances in networking technologies have started to change the game for distributed data processing by providing bandwidth on the same order of magnitude as the memory bus. For instance, InfiniBand FDR $4\times$ bandwidth is roughly equivalent to DDR3-1066 (see Figure 1), suggesting that a single memory channel can be fully saturated by data transferred over the network.

One possible solution to these three observations is to adapt recent work on high-speed DBMSs for traditional OLAP workloads [45, 26, 20]. While these systems excel at simple calculations, they have two significant weaknesses when dealing with the next generation of analytics workloads. Foremost, from a usability perspective, SQL is exceptionally cumbersome for expressing many classes of problems, such as ML algorithms. Second, many of the well-known optimizations for relational queries no longer suffice for more complex analytics workflows.

Another approach involves building upon the previously mentioned, more flexible MapReduce-style analytics frameworks that specifically treat UDFs as first-class citizens. However, these systems still regard UDFs as black boxes, and thus they have difficulty leveraging the characteristics of UDFs to optimize workflows at a low level.

Since neither of these alternatives seems adequate, we believe that the most promising avenue for improving the performance of complex analytics tasks involves extending existing code generation

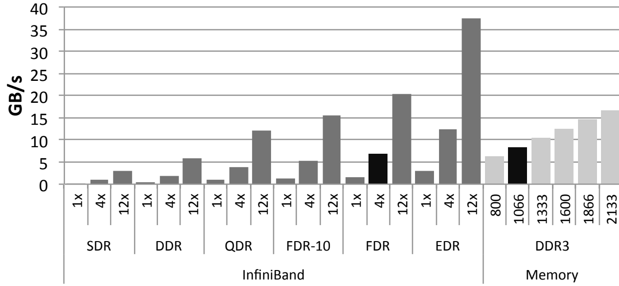

Figure 1: InfiniBand vs Memory Bandwidth

techniques for traditional SQL workloads [28] to this new domain. In order to explore this idea, we are developing TUPLEWARE, a high-performance distributed analytics system that compiles workflows of UDFs directly to compact, self-contained distributed programs. Our approach marries ideas from the DBMS and compiler communities to take full advantage of the underlying hardware.

We have designed TUPLEWARE to integrate with the LLVM [29] compiler framework, which provides a number of unique advantages over traditional systems. From a usability perspective, LLVM makes the frontend completely language-agnostic, so users can write programs in their favorite programming language. From an architecture perspective, LLVM allows the system to introspect UDFs and gather statistics useful for modeling expected execution behavior. Therefore, UDFs are no longer black boxes, opening the door for a completely new class of optimizations that consider UDF characteristics in order to generate different code on a case-by-case basis. In summary, we make the following contributions:

- We present TUPLEWARE, a high-performance distributed system designed specifically for complex, compute-intensive analytics tasks on small clusters.
- We propose a novel process for program synthesis that opens the door for a new breed of code generation optimizations by considering properties about the data, computations, and underlying hardware.
- We benchmarked TUPLEWARE and achieved speedups of up to three orders of magnitude over other systems for several common analytics tasks.

2. TUPLEWARE

The goal of TUPLEWARE is to specifically target complex analytics on more reasonable dataset sizes while taking full advantage of the unique characteristics of smaller, more powerful clusters. Optimizing for small clusters fundamentally changes the way we should design analytics tools, since workloads are usually limited by the computation rather than the data transfer. With the bottlenecks of large cloud deployments gone, the last remaining bottleneck is the CPU, and current systems cannot be easily adjusted to meet these new challenges. For instance, recent attempts to adapt Hadoop for InfiniBand produced rather disappointing results [31, 22].

We are therefore building TUPLEWARE from the ground up to address the computation bottleneck. As previously mentioned, TUPLEWARE leverages LLVM to optimize workflows without borders between UDFs, seamlessly integrating the computations with the overarching control flow in order to synthesize the most efficient code possible. As shown in Figure 2, TUPLEWARE consists of three main parts: (1) Frontend, (2) Program Synthesis, and (3) Execution.

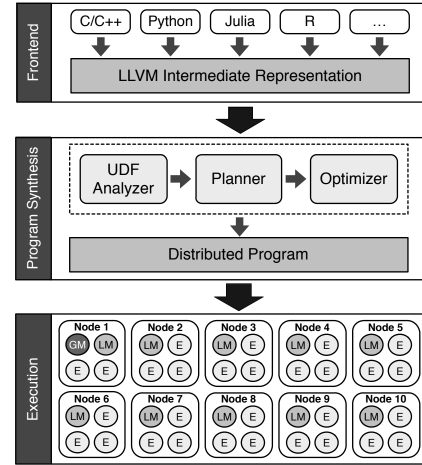

Figure 2: TUPLEWARE's Architecture

2.1 Frontend

Users 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. Similar to many recent frameworks (e.g., Spark, Stratosphere [23], DryadLINQ [42]), TUPLEWARE allows users to define complex workflows directly inside a host language by passing UDFs to API operators. However, by basing our platform on LLVM, TUPLEWARE is language-agnostic and can support any language with an LLVM compiler.

TUPLEWARE's algebra is based on the strong foundation of functional programming with monads. 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. This abstraction allows for efficient parallel data processing and attempts to: (1) strike a middle ground between the expressiveness of MapReduce and optimizability of SQL; (2) natively handle iterative workflows in order to optimize transparently across iterations; and (3) provide flexible shared state primitives with configurable synchronization patterns that can be directly accessed from within the workflow.

Expressive & Optimizable: While popular, many have criticized MapReduce [18] for rejecting the advantages (e.g., optimizability, usability) of high-level languages like SQL [3]. However, SQL is unwieldy for expressing complex workflows, resulting in convoluted queries that are difficult to understand and maintain. Some recent frameworks (e.g., Spark, Stratosphere, DryadLINQ) have started to bridge the gap between expressiveness and optimizability by supporting arbitrary UDF workflows, but they do not consider how to optimize execution at a low level.

Iterations: Many algorithms are most naturally expressed iteratively, but neither MapReduce nor SQL effectively supports iteration [30, 17]. Handling iterations via an external driver program on the client is straightforward, but cross-iteration optimization becomes difficult. Both Spark and DryadLINQ take this approach, submitting a completely independent job for each iteration. In contrast, a number of iterative extensions to MapReduce incorporate native iteration (e.g., Stratosphere, HaLoop [11], Twister [19]), but they either lack low-level optimization potential or do not scale well.

Shared State: Shared state is a key ingredient of many algorithms, but several attempts to support distributed shared state within a MapReduce-style framework severely restrict how and when programs can interact with global variables. For instance, the Iterative Map-Reduce-Update [10] model supplies map and reduce functions

with read-only copies of state values that are recalculated during the update phase after each iteration. However, this paradigm is designed for iterative refinement algorithms and cannot model convex optimization problems (e.g., neural networks, maximum likelihood Gaussian mixtures). Furthermore, Iterative Map-Reduce-Update precludes algorithms with different synchronization patterns (e.g., Hogwild! [36]). Spark also 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. Additionally, Spark's broadcast variables allow remote machines to cache large read-only values between tasks, but they cannot be used to represent shared state that changes frequently (e.g., ML models) because they can never be updated.

2.2 Program Synthesis

After a user has composed and submitted an analytics task, Tupleware automatically compiles the workflow into a distributed program during the Program Synthesis stage. *Program synthesis* is the process of automatically compiling a workflow of UDFs into a distributed program, which involves (1) analyzing UDFs in order to perform low-level optimizations, (2) determining how to parallelize a task in a distributed setting, and (3) generating code to actually execute the workflow.

UDF Analyzer: Systems that treat UDFs as black boxes have difficulty making informed decisions about how best to execute a given workflow. Although recent frameworks (e.g., Stratosphere) have proposed methods for introspecting UDFs, they primarily focus on detecting special patterns (e.g., if a map filters tuples) in order to apply high-level workflow rewrites. In contrast, our process leverages the LLVM framework to look deeper inside UDFs and gather statistics for optimizing workflows at a low level. The UDF Analyzer examines the LLVM intermediate representation of each UDF to determine different features (e.g., vectorizability, computation cycle estimates, memory bandwidth predictions) useful for modeling execution behavior. This is a crucial step in our program synthesis process and allows us to generate more efficient code by considering low-level UDF characteristics.

Planner: Data parallel processing allows for the utilization of all available computing resources to more efficiently evaluate a given workflow. Our programming model supplies the Planner with information about the parallelization semantics of different operations in a workflow. The Planner can then determine how best to deploy a job given the available resources and physical data layout, taking into account the optimum amount of parallelization for each individual operation. For example, workflows involving smaller datasets may not benefit from massive parallelization due to the associated deployment overhead. The Planner annotates a logical execution plan with this information before passing it to the Optimizer for translation to a distributed program.

Optimizer: *Code generation* is the process by which compilers translate a high-level language into an optimized low-level form. As other work has shown [28], 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 Optimizer translates the logical plan produced by the Planner into a self-contained distributed program and uses UDF statistics gathered by the UDF Analyzer to apply low-level optimizations tailored to the underlying hardware. As part of the translation process, the Optimizer produces all of the data structure, control flow, synchronization, and communication code necessary to form a complete distributed program. We describe the different types of optimizations and provide an example in Section 3.

2.3 Execution

The result of the Program Synthesis process is a fully compiled, self-contained distributed program that includes all necessary communication and synchronization code, thereby minimizing the overhead associated with interpreted execution models. This program is then automatically deployed on a cluster of machines, visualized in Figure 2 as 10 nodes (boxes) each with 4 hyper-threads (circles inside the boxes). We take a multi-tiered approach to distributed execution and assign special roles to individual hyper-threads, as described below.

Global Manager: The Global Manager (GM) is at the top of the hierarchy and supervises the current stage of the execution (e.g., maintaining the distributed catalog, monitoring cluster health, coarse-grained partitioning of data, keeping track of shared state values). We reserve a single hyper-thread in the cluster for the GM (located in Node 1), allowing other hyper-threads to execute data processing tasks uninterrupted.

Local Manager: We dedicate one hyper-thread per node as a Local Manager (LM), which is responsible for the fine-grained management of the local shared memory. 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. We also try to avoid unnecessary object creation and data copying when possible. For instance, TUPLEWARE can perform 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.

Executor Thread: The LM is also responsible for actually deploying compiled programs and does so by spawning new executor threads (E), which actually execute the workflow. Assigning each thread to a dedicated core allows it to run without interruption and avoids costly context switching and CPU-cache pollution. 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. Since TUPLEWARE's data request model is multi-tiered and pull-based, we achieve automatic load balancing with minimal overhead. Each thread requests data in small blocks that fit in the CPU cache 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.

3. OPTIMIZATIONS

By integrating high-level query optimization techniques with low-level compiler techniques, we believe TUPLEWARE is able to push the envelope of query optimizations to new grounds. We can leverage our unique architecture to apply optimizations that were previously impossible.

3.1 Optimization Categories

Since our system can introspect UDFs and generates code for each workflow, our optimizer can apply a broad range of optimizations that occur on both a logical and physical level, which we divide into three categories.

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

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

Combined: Some systems incorporate DBMS and compiler optimizations separately, first performing algebraic transformations and then independently generating code based upon a fixed strategy. On the other hand, our program synthesis process combines an optimizable high-level algebra and statistics gathered by the UDF Analyzer with the ability to dynamically generate code, enabling optimizations that would be impossible for either a DBMS or compiler alone. Optimizations in this category 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. We provide an example of one such optimization for selection operations in the next section.

3.2 Example

We are exploring compilation techniques for selections within the context of UDF-centric workflows to generate different code based upon characteristics about the data (e.g., selectivity) and the computations (e.g., complexity). For example, consider the following workflow defined on a TupleSet ts with selection predicate p and map function f:

```
ts.select(x => p(x)).map(x => f(x))
```

The most straightforward way to execute this workflow is the *branch* strategy. As shown in the code snippet below, for each input tuple in $data$, a conditional statement checks to see whether that tuple satisfies the predicate p. If the predicate is satisfied, then the map function f is applied to the tuple, which is then added to an output buffer $result$; otherwise, the loop skips the tuple and proceeds to the next iteration.

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

This strategy performs well for both very low and very high selectivities, when the CPU can perform effective branch prediction. For intermediate selectivities, though, branch misprediction penalties have a major negative impact on performance.

An alternative to the branch approach is the *no-branch* strategy, which eliminates branch mispredictions by replacing the control dependency with a data dependency. The code snippet below shows how this approach maintains a pointer pos to the current location in the output buffer that is incremented every time an input tuple satisfies the predicate. If a tuple does not satisfy the predicate, then the pointer is not incremented and the previous value is overwritten.

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

Although the tradeoffs between these two approaches have been studied by other work [37], we extend these techniques by examining how UDFs impact the decision. For instance, a very complex map function f might benefit from the branching in the first strategy, whereas a very simple f would not be prohibitive for the no-branch strategy. We are also investigating how to optimize at a low level by generating different code that considers: (1) how best to handle arbitrary UDFs that come after a selection in a workflow; (2) how the optimal strategy changes when considering multiple predicates of varying complexities; and (3) how to allocate space for the result of a selection.

Subsequent UDFs: Unlike standard DBMSs, TUPLEWARE considers selection operations in the context of more complex UDF workflows. As described, a traditional approach would attempt to pipeline selections with all subsequent operations in the workflow, but considering the characteristics of these subsequent UDFs might change the optimal way to generate the code. For example, it may sometimes be better to defer execution of the remainder of the workflow until filling an intermediate buffer with tuples that satisfy the predicate, and the benefits would be twofold: (1) far better instruction locality because the processing occurs in a tight loop free of conditional branches; and (2) allowing for vectorization that would otherwise be prevented by the conditional statement.

Multiple Predicates: Increasing the number of predicates shifts the bottleneck from branch mispredictions to the actual predicate evaluation. The no-branch strategy always performs a constant amount of work, which increases linearly with each additional predicate. Conversely, short-circuit evaluation in the branch strategy becomes increasingly valuable, thereby making this approach an attractive option. The fixed nature of predicate evaluation in the no-branch strategy, however, lends itself to SIMD vectorization, and we notice a significant opportunity for the vectorized computation of compound predicates as demonstrated by promising initial results.

Result Allocation: Result allocation is particularly difficult for selections, since the output size is not known a priori. The most obvious way to completely avoid over-allocation is to allocate space for only one result tuple every time an input tuple passes the predicate. However, the overhead of tuple-at-a-time allocation quickly becomes prohibitive for even relatively small data sizes. The other extreme would assume a worst case scenario and allocate all possible necessary space, thereby paying a larger allocation penalty once at the beginning and avoiding result bounds checking. This approach may work for very high selectivities but ultimately wastes a lot of space. Therefore, an intermediate allocation strategy that incrementally allocates blocks of tuples seems most promising, and we are looking into the tradeoffs associated with these various strategies.

4. PRELIMINARY RESULTS

We evaluated our TUPLEWARE prototype against two popular analytics frameworks (Hadoop 2.4.0 and Spark 1.1.0) on a small, powerful cluster consisting of 10×`c3.8xlarge` EC2 instances with Intel E5-2680v2 processors (10 cores, 25MB Cache), 60GB RAM, 2 × 320GB SSDs, and 10 Gigabit*4 Ethernet. Our preliminary results show that our techniques can outperform these systems by up to three orders of magnitude for the tested workloads.

4.1 Workloads and Data

Our benchmarks included five common ML tasks that operate on datasets of 1, 10, and 100GB in size. We implemented a consistent version of each algorithm across all systems using synthetic datasets in order to test across a range of data characteristics (e.g., size, dimensionality, skew). For iterative algorithms, we report the total time taken to complete 20 iterations. We record the total runtime of each algorithm after the input data has been loaded into memory and parsed, except in the case of Hadoop, which must always read from and write to HDFS. We now describe each ML task.

K-means: 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 fit-

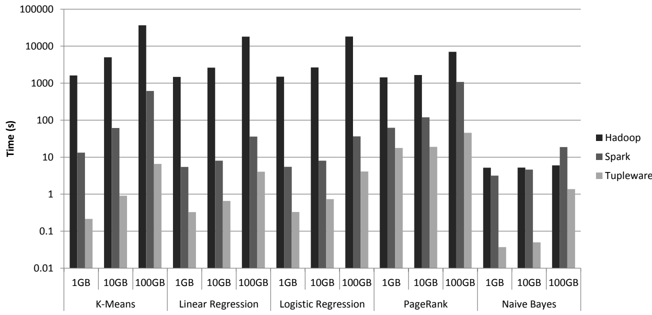

Figure 3: Preliminary Benchmarking Results

ting a linear equation to a set of observed data points. Our implementation builds a model on data with 1024 features using a parallelized batch gradient descent algorithm.

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 data with 1024 features.

Naive Bayes: A naive Bayes classifier is a conditional model that uses feature independence assumptions to assign class labels. We trained a naive Bayes classifier on a dataset with 1024 features and 10 possible labels.

PageRank: PageRank is an iterative link analysis algorithm that assigns a weighted rank to each page in a web graph to measure its relative significance. Our synthetic web graph had one million distinct pages.

4.2 Discussion

As shown in Figure 3, TUPLEWARE outperforms Hadoop by up to three orders of magnitude and Spark by up to two orders of magnitude for the tested ML tasks.

TUPLEWARE is able to significantly outperform Hadoop because of the substantial I/O overhead required for materializing intermediate results to disk between iterations. On the other hand, TUPLEWARE is able to cache intermediate results in memory and perform hardware-level optimizations to greatly 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 keep 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. However, for CPU-intensive ML tasks such as k-means, TUPLEWARE outperforms Spark by synthesizing distributed programs and employing low-level code generation optimizations.

In order to explain these speedups, we conducted a preliminary performance breakdown to quantify the impact of some of TUPLEWARE's distinguishing features. First, we found that although C++ offers significantly lower level control than Java, the choice of C++ over Java has relatively little overall impact on runtime (around $2.5\times$ faster for most workloads). We also found that by compiling workflows directly into distributed programs, TUPLEWARE can both inline UDF calls and avoid the overhead associated with Volcano-style iterators, improving performance by $2.5\times$ and $4\times$, respectively. The remainder of the speedup can be attributed to a combination of optimization techniques and intangible design differences (e.g., workflow compilation, deployment architecture). We are still investigating other factors and plan to have a more complete performance breakdown in the future.

5. FUTURE DIRECTIONS

Code generation for UDF-centric workflows on modern hardware offers a wide range of unique optimization opportunities. We now outline three promising future research directions.

Automatic Compression: TUPLEWARE's UDF Analyzer can detect UDFs inside a workflow that are likely to be memory-bound. An interesting line of work would be to automatically inject code to compress and decompress data at different points in the workflow. The challenge is not only to inject the code but also to develop an accurate cost model that can decide based on the UDF and data statistics whether compressing and decompressing data on-the-fly is worthwhile. Similar to work examining how relational operations can operate directly on compressed data [39], it might even be possible to do the same for some UDFs.

On-Demand Fault Tolerance: As our experiments demonstrate, TUPLEWARE can process gigabytes of data with sub-second response times, suggesting that completely restarting the job entirely would be simpler and more efficient in the rare event of a failure. Extremely long-running jobs on the order of hours or days, though, might benefit from intermediate result recoverability. In these cases, TUPLEWARE could perform 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 fault tolerance mechanisms on a case-by-case basis. If our previously described program synthesis process estimates that a particular job will have a long total runtime, we could combine that estimation with the probability of a failure (given our intimate knowledge of the underlying hardware) to decide whether to include checkpointing code.

High-Performance Networks: High-performance networks offer tremendous opportunities for data analytics, since they can provide bandwidth in the same ballpark as the memory bus. This property changes a number of long-standing assumptions about distributed data processing, including the primacy of data locality and the common practice of always pushing the work to the data. Furthermore, recent advances in Remote Direct Memory Access (RDMA) improve data transfer latency and could allow the CPU to concentrate on other tasks [32], and advanced CPU features, such as Intel's Data Direct I/O (DDIO) technology [6], might enable the next generation of systems to read data directly into the local CPU cache from another machine. Current frameworks, though, are based on the TCP/IP stack and would require significant rewriting in order to fully take advantage of these new technologies [31]. As the network bottleneck starts to disappear, we therefore need a critical rethinking of data partitioning and load balancing schemes.

6. RELATED WORK

While other work has looked at individual components, we plan to use TUPLEWARE to collectively explore how to (1) easily and concisely express UDF-centric workflows, (2) synthesize self-contained distributed programs optimized at the hardware level, and (3) deploy tasks efficiently on a small cluster.

6.1 Programming Model

Numerous extensions have been proposed to support iteration

and shared state within MapReduce [11, 19, 7], and some projects (e.g., SystemML [21]) 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 [14], Ciel [33], and Piccolo [34] lack the low-level optimization potential that TUPLEWARE's frontend provides.

DryadLINQ [42] 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 cross-iteration optimizations.

Scope [13] 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 UDF-centric workflows.

TUPLEWARE also has commonalities with the programming models proposed by Spark [43] and Stratosphere [23]. 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.

6.2 Code Generation

Code generation for query evaluation was proposed as early as System R [9], but this technique has recently gained popularity as a means to improve query performance for in-memory DBMSs [35, 28]. Both HyPer [26] and VectorWise [46] propose different optimization strategies for query compilation, but these systems focus on SQL and do not optimize for UDFs. LegoBase [27] includes a query engine written in Scala that generates specialized C code and allows for continuous optimization, but LegoBase also concentrates on SQL and does not consider UDFs.

DryadLINQ compiles user-defined workflows using the .NET framework but applies only traditional high-level optimizations. Similarly, Tenzing [15] and Impala [2] are SQL compilation engines that also focus on simple queries over large datasets.

OptiML [40] 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.

6.3 Single-Node Frameworks

BID Data Suite [12] and Phoenix [41] 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 [5] and Matlab [4] have these same limitations. More specialized systems (e.g., Hogwild! [36], DimmWitted [44]) provide highly optimized implementations for specific algorithms on a single machine, whereas TUPLEWARE is intended for general analytics in a distributed environment.

7. CONCLUSION

Complex analytics tasks have become commonplace for a wide range of users. However, instead of targeting the use cases and computing resources of the typical user, existing frameworks are designed primarily for working with huge datasets on large cloud deployments with thousands of commodity machines. This paper described our vision for TUPLEWARE, a new 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.

8. ACKNOWLEDGMENTS

This research was funded by the Intel Science and Technology Center for Big Data and gifts from Amazon, Mellanox, and NEC.

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

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

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

[6] Intel Data Direct I/O Technology. http://www.intel.com/content/www/us/en/io/direct-data-i-o.html, 2014.

[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] I. T. Association. InfiniBand Roadmap. http://www.infinibandta.org/, 2013.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Cruanes, B. Dageville, and B. Ghosh. Parallel sql execution in oracle 10g. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 850–854, New York, NY, USA, 2004. ACM.

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

[19] 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.

[20] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[21] A. Ghoting et al. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[22] J. Huang, X. Ouyang, J. Jose, M. W. ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-performance design of hbase with rdma over infiniband. In *Proceedings of the IEEE International Conference on Parallel Distributed Processing Symposium*, IPDPS, pages 774–785, 2012.

[23] 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.

[24] Intel. Intel Xeon Processor E7 Family: Reliability, Availability, and Serviceability . *Whitepaper*, 2010.

[25] JEDEC. DDR3 SDRAM Standard. `http://www.jedec.org/standards-documents/docs/jesd-79-3d`, 2013.

[26] 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.

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

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

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

[30] 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.

[31] X. Lu, N. S. Islam, M. W. ur Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *Proceedings of the International Conference on Parallel Processing*, ICPP, pages 641–650, 2013.

[32] P. MacArthur and R. D. Russell. A performance study to guide rdma programming decisions. In *Proceedings of the International Conference on High Performance Computing and Communication & International Conference on Embedded Software and Systems*, HPCC-ICESS, pages 778–785, 2012.

[33] 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 *in Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI). USENIX*.

[34] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables.

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

[36] 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.

[37] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 109–120, New York, NY, USA, 2002. ACM.

[38] 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.

[39] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[40] 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.

[41] 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.

[42] 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.

[43] 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. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[44] C. Zhang and C. Re. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.

[45] M. Zukowski, P. A. Boncz, N. Nes, and S. HÃl'man. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

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