Optimization Techniques for "Scaling Down" Hadoop on Multi-Core, Shared-Memory Systems

K. Ashwin Kumar Jonathan Gluck Amol Deshpande Jimmy Lin
University of Maryland, College Park
{ashwin, jdg, amol}@cs.umd.edu, jimmylin@umd.edu

ABSTRACT

The underlying assumption behind Hadoop and, more generally, the need for distributed processing is that the data to be analyzed cannot be held in memory on a single machine. Today, this assumption needs to be re-evaluated. Although petabyte-scale datastores are increasingly common, it is unclear whether "typical" analytics tasks require more than a single high-end server. Additionally, we are seeing increased sophistication in analytics, e.g., machine learning, where we process smaller and more refined datasets. To address these trends, we propose "scaling down" Hadoop to run on multi-core, shared-memory machines. This paper presents a prototype runtime called Hone ("Hadoop One") that is API compatible with Hadoop. With Hone, we can take an existing Hadoop application and run it efficiently on a single server. This allows us to take existing MapReduce algorithms and find the most suitable runtime environment for execution on datasets of varying sizes. For dataset sizes that fit into memory on a single machine, our experiments show that Hone is substantially faster than Hadoop running in pseudo-distributed mode. In some cases, Hone running on a single machine outperforms a 16-node Hadoop cluster.

1. INTRODUCTION

The Hadoop implementation of MapReduce [6] has become the tool of choice for "big data" processing (whether directly or indirectly via higher-level tools such as Pig or Hive). Among its advantages are the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance. There has been much activity in applying Hadoop to problems in data management as well as data mining and machine learning. Over the past several years, the community has accumulated a significant amount of expertise and experience on how to recast traditional algorithms in terms of the restrictive primitives *map* and *reduce*.

Computing environments have evolved substantially since the introduction of Hadoop. For example, in 2008, a typical Hadoop node might have two dual-core processors with a total of 4 GB of RAM. Today, a high-end commodity server might have two eight-core processors and 256 GB of RAM—such a server can be purchased for roughly $10,000 USD. This means that a single server today has more cores and more memory than did a small Hadoop cluster from a few years ago. The assumption behind Hadoop and the need for distributed processing is that the data to be analyzed cannot be held in memory on a single machine. Today, this assumption needs to be re-evaluated.

Although it is true that petabyte-scale datastores are becoming increasingly common, it is unclear whether datasets used in "typical" analytics tasks today are really too large to fit in memory on a single server. Of course, organizations such as Yahoo, Facebook, and Twitter routinely run Pig or Hive jobs that scan terabytes of log data, but these organizations should be considered outliers—they are not representative of data analytics in most enterprise or academic settings. Even still, according to the analysis of Rowstron et al. [20], at least two analytics production clusters (at Microsoft and Yahoo) have median job input sizes under 14GB and 90% of jobs on a Facebook cluster have input sizes under 100GB. Holding all data in memory does not seem too far-fetched.

There is one additional issue to consider: over the past several years, the sophistication of data analytics has grown substantially. Whereas yesterday the community was focused on relatively simple tasks such as natural joins and aggregations, there is an increasing trend toward data mining and machine learning. These algorithms usually operate on more refined, and hence, smaller datasets—typically in the range of tens of gigabytes.

These factors suggest that it is worthwhile to consider in-memory data analytics on modern servers—but it still leaves open the question of how we orchestrate computations on multi-core, shared-memory machines. Should we go back to multi-threaded programming? That seems like a step backwards because we embraced the simplicity of MapReduce for good reason—the complexity of concurrent programming with threads is well known. Our proposed solution is to "scale down" Hadoop to run on shared-memory machines [10]. In this paper, we present a prototype runtime called Hone ("Hadoop One") that is API compatible with standard (distributed) Hadoop. That is, we can take an existing Hadoop algorithm and efficiently run it, without modification, on a multi-core, shared-memory machine using Hone. This allows us to take an existing application and find the most suitable runtime environment for execution on datasets of varying sizes—if the data fit in memory, we can avoid network latency and significantly increase performance in a shared-memory environment.

Hadoop API compatibility is the central tenet in our design. Although there are previous MapReduce implementations for shared-memory environments (see Section 2), taking advantage of them would require porting Hadoop code over to another API. In contrast, Hone is able to leverage existing implementations. In this paper, we present experiments on a number of "standard" MapReduce

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

10.5441/002/edbt.2014.03

algorithms (word count, PageRank, etc.) as well as a Hadoop implementation of Latent Dirichlet Allocation (LDA). This implementation represents a major research effort [26] and demonstrates API compatibility on a non-trivial application.

Contributions. Our contributions can be summarized as follows:

- Hone is a scalable MapReduce implementation for multi-core, shared-memory machines. To our knowledge it is the first MapReduce implementation that is both Hadoop API compatible and optimized for scale-up architectures.
- We propose and evaluate different approaches to implementing the data shuffling stage in MapReduce, which is critical to high performance.
- We discuss key challenges in implementing Hone on the JVM, how we addressed them, and lessons we learned along the way.
- We evaluate Hone on a number of real-world applications, comparing it to Hadoop pseudo-distributed mode, a 16-node Hadoop cluster, and a few other systems.
- We share a synthetic workload generator for evaluating Hone that may be of independent interest for evaluating other systems.

2. RELATED WORK

There has been much work on MapReduce and related distributed programming frameworks over the past several years. The literature is too vast to survey, so here we specifically focus on MapReduce implementations for shared-memory environments. Perhaps the best known is a series of systems from the Phoenix project: the first system, Phoenix [19], evaluated the suitability of MapReduce as a programming framework for shared-memory systems. Phoenix2 [24] improved upon Phoenix by identifying inefficiencies in handling large datasets—it utilizes user-tunable hash-based data structures to store intermediate data. Phoenix++ [23] made further improvements by observing that optimal intermediate data structures cannot be determined *a priori*, as they depend on the nature of the application. Thus, the system provides container objects to store map output as an abstraction to the developer. We see several shortcomings of the Phoenix systems that limit broad applicability. First, they are implemented in C/C++ and are not compatible with Hadoop. Therefore, scaling down a Hadoop application using Phoenix involves essentially a full reimplementation. Second, hash-based containers are not a feasible solution for a Java implementation, especially for applications that generate a large amount of intermediate data. Java objects tend to be heavyweight, and standard Java collections are inefficient for storing large datasets in memory. We discuss this in more detail in Section 4.2.

In addition to the Phoenix systems, there have been other MapReduce implementations for multi-core and share-memory environments. Mao et al. [16] described Metis, which proposed using containers based on hashing, where each hash bucket points to a B-tree to store intermediate output. Chen et al. [5] proposed a "tiled" MapReduce approach to iteratively process small chunks of data with efficient use of resources. Jiang et al. [8] built upon Phoenix and provided an alternate API for MapReduce. All of these systems were implemented in C, and in some cases modify the MapReduce model, and therefore they are not compatible with Hadoop. There has also been a previous attempt to develop MapReduce implementations on multi-GPU systems by Stuart et al. [22].

Shinnar et al. [21] presented Main Memory MapReduce (M3R), which is an implementation of the Hadoop API targeted at online analytics on high mean-time-to-failure clusters. Although close in spirit to our system, they mainly focus on scale-out architectures, whereas we focus explicitly on single-machine optimizations. Furthermore, their experiments do not provide insights on scalability and workloads behaviors in a scale-up setting. Spark [25] provides primitives for cluster computing built on a data abstraction called resilient distributed datasets (RDDs), which can be cached in memory for repeated querying and efficient iterative algorithms. Spark is implemented in Scala, and so like Hone it runs on the JVM. However, Spark provides a far richer programming model than MapReduce. Aside from not being directly Hadoop compatible, our experiments with Spark show that it performs poorly on multi-core, shared-memory machines (Section 6.4). This is no surprise since Spark has not been optimized for such architectures.

Recently, Appuswamy et al. [1] also observed that most MapReduce jobs are small enough to be executed on a single high-end machine. However, they advocated tuning Hadoop pseudo-distributed mode instead of building a separate runtime. In the next section, we discuss Hadoop pseudo-distributed mode in detail, and it serves as a point of comparison in our experiments. We show that a well-engineered, multi-threaded MapReduce implementation optimized for execution on a single JVM can yield substantial performance improvements over Hadoop pseudo-distributed mode.

	Write 8GB		**Read 8GB**	
	Cold Cache	Warm Cache	Cold Cache	Warm Cache
HDFS	178.0s	32.7s	81.4s	28.9s
Disk	194.0s	25.3s	27.1s	1.7s
	Write 64MB		**Read 64MB**	
	Cold Cache	Warm Cache	Cold Cache	Warm Cache
HDFS	5.10s	1.72s	5.64s	2.12s
Disk	0.47s	0.11s	3.27s	0.20s

Table 1: Performance comparisons between HDFS and direct disk access.

3. HADOOP ON SINGLE MACHINE

We begin by discussing why Hadoop does not perform well on a single machine. To take advantage of multi-core architectures, Hadoop provides pseudo-distributed mode (PDM henceforth), in which all daemon processes run on a single machine (on multiple cores). This serves as a natural point of comparison, and below we identify several disadvantages of running Hadoop PDM.

Multi-process overhead: In PDM, mapper and reducer tasks occupy separate JVM processes. In general, multi-process applications suffer from inter-process communication (IPC) overhead and are typically less efficient than an equivalent multi-threaded implementation that runs in a single process space.

I/O Overhead: Another disadvantage of Hadoop PDM is the overhead associated with reading from and writing to HDFS. To quantify this, we measured the time it takes to read and write a big file (8GB) and a single split of the file (64MB) using HDFS as well as directly using Java file I/O (on the server described in Section 5.1). These results are shown in Table 1.

We find that direct reads from disk are much faster than reads from HDFS for both the 8GB and 64MB conditions. Performance improvements are observed under both cold and warm cache conditions, and the magnitude of improvement is higher under a warm cache. Interestingly, we find that writing 8GB is faster using HDFS, but in all other conditions HDFS is slower. In the small data case (64MB), writes are over a magnitude faster under both cold and warm cache conditions. These results confirm that disk I/O operations using HDFS can be extremely expensive [7, 15] when compared to direct disk access. In Hadoop PDM, mappers read from HDFS and reducers write to HDFS, even though the system is running on a single machine. Thus, Hadoop PDM suffers from these HDFS performance issues.

Framework overhead: Hadoop is designed for high-throughput processing of massive amounts of data on potentially very large clusters. In this context, startup costs are amortized over long-running jobs and thus do not have a large impact on overall performance. Hadoop PDM inherits this design, and in the context of a single machine running on modest input data sizes, job startup costs become a substantial portion of overall execution time.

Hadoop PDM on a RAM disk provides negligible benefit: One obvious idea is to run Hadoop PDM using a RAM disk to store intermediate data. RAM disks tend to help most with random reads and writes, but since most Hadoop I/O consists of sequential operations, it is not entirely clear how much a RAM disk would help. Our initial experiments with Hadoop PDM did explore replacing rotational disk with RAM disk. We ran evaluations on the applications in Section 5.2, but results showed no benefits when using a RAM disk. Moreover, previous studies have shown that a RAM disk is at least four times slower than raw memory access [9, 17]. We expected greater benefits by moving completely to managing memory directly, so we did not pursue study of Hadoop PDM on RAM disks any further.

4. HONE ARCHITECTURE

The overall architecture of Hone is shown in Figure 1. Below, we provide an overview of each processing stage.

Mapper Stage: As in Hadoop, this stage applies the user-specified mapper to the input dataset to emit intermediate (key, value) pairs. Each mapper is handled by a separate thread, which consumes the supplied *input split* (i.e., portion of the input data) and processes input records according to the user-specified InputFormat. Like Hadoop, the total number of mappers is determined by the number of input splits. This stage uses a standard thread-pooling technique to control the number of mapper tasks that execute in parallel. Mappers in Hone accept input either from disk or from a *namespace* residing in memory (see Section 4.2 for more details).

Data Shuffling Stage: In MapReduce, intermediate (key, value) pairs need to be shuffled from the mappers where they are created to the reducers where they are consumed. In Hadoop, data shuffling is interwoven with sorting, but in Hone these are two separate stages. The next section describes three different approaches to data shuffling. Overall, we believe that efficient implementations of this process is the key to a high-performance MapReduce implementation on multi-core, shared-memory systems.

Sort Stage: As with Hadoop, Hone sorts intermediate (key, value) pairs emitted by the mappers. Sorting is handled by a separate thread pool with a built-in load balancer, on streams that have already been assigned to the appropriate reducer (as part of the data shuffling stage). If the sort streams grow too large, then an automatic splitter divides the streams on the fly and performs parallel sorting on the split streams. The split information is passed to the reducer stage for proper stream assignment. The default stream split size can be set as part of the configuration.

Reducer stage: In this stage, Hone applies the user-specified reducer on values associated with each intermediate key, per the standard MapReduce programming model. A reducer either writes output (key, value) pairs to disk or to memory via the namespace abstraction for further processing.

Combiners: In a distributed setting, combiners mimic the functionality of reducers locally on every node, serving as an optimization to reduce network traffic. Proper combiner design is critical

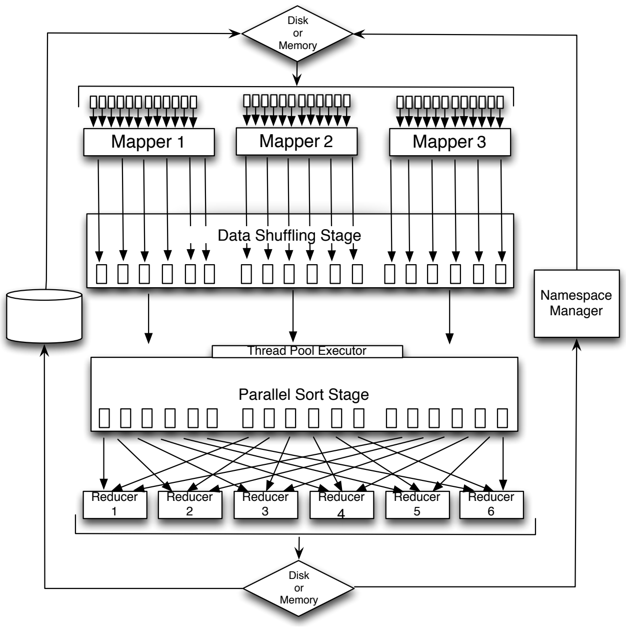


Figure 1: Hone system architecture.

to the performance of a distributed MapReduce algorithm, but it is unclear whether combiners are useful when the entire MapReduce application is running in memory on a single machine. For this reason, Hone currently does not support combiners: since they are optional optimizations, we can ignore them without affecting algorithm correctness.

Namespace Manager: This module manages memory assignment to enable data reading and writing for MapReduce jobs. It converts filesystem paths that are specified in the Hadoop API into an abstraction we call a *namespaces*: output is directed to an appropriate namespace that resides in-memory, and, similarly, input records are directly consumed from memory as appropriate.

4.1 In-Memory Data Shuffling

We propose three different approaches to implement data shuffling between mappers and reducers: (1) the pull-based approach, (2) the push-based approach, and (3) the hybrid approach. These are described in detail below.

Pull-based Approach: In the pull-based approach, each mapper emits keys to r streams, where r is the number of reducers. Each mapper applies the partitioner to assign each intermediate (key, value) pair to one of the r streams based on the key (per the standard MapReduce contract). If m is the number of mappers, then there will be a total $m \times r$ intermediate streams. In the sort stage, these $m \times r$ intermediate streams are sorted in parallel. In the reducer stage, each reducer thread *pulls* from m of the $m \times r$ streams (one from each mapper). Figure 2 shows an example with three mappers and six reducers, with eighteen intermediate streams.

With this approach we encounter an interesting issue regarding garbage collection. In Java, a thread is its own garbage collection (GC) root. So any time a thread is created, irrespective of creation context, it will not be ready for garbage collection until its run method completes. This is true even if the local method which created the thread completes. In Hone, we maintain a pool of mapper threads containing tp_m threads (usually for large jobs, $tp_m \ll m$, where m is the number of mappers, determined by the split size).

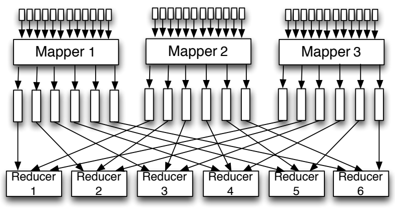


Figure 2: Pull-based approach

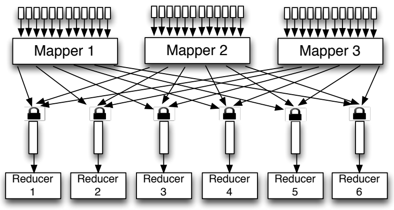


Figure 3: Push-based approach

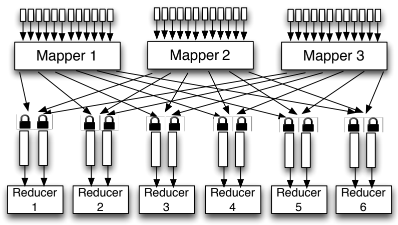


Figure 4: Hybrid approach

Thus, tp_m mappers are running concurrently, and the objects that each creates cannot be garbage collected until the mapper finishes. Increasing this thread pool size allows us to take advantage of more cores, but at the same time this increases the amount of garbage that cannot be collected at any given time. Hone needs to contend with the characteristics of the JVM, and garbage collection is one re-occurring issue we have faced throughout this project.

Push-based Approach: In this approach, Hone creates only r intermediate streams, one for each reducer. This is shown in Figure 3, where we have six streams. Each mapper emits intermediate (key, value) pairs directly into one of those r streams based on the partitioner. In this way, the mappers *push* intermediate data over to the reducers. Because r streams are being updated by the mappers in parallel, these streams must be synchronized and guarded by locks. Due to this synchronization overhead, contention is unavoidable, but this cost varies based on the distribution of the intermediate keys to reducers. There are two ways of dealing with contention cost: the first is to employ scalable and efficient locking mechanisms (more discussion below), and second is to increase the number of reducers so that key distribution to reducers is spread out, which in turn will reduce synchronization overhead. However, if we have too many reducers, context-switching overhead of reducer threads will negatively impact performance.

The push-based approach creates fewer intermediate data structures for the same amount of intermediate (key, value) pairs, and thus in this manner is more efficient. In the pull-based approach, since *each* mapper output is divided among r streams, the object overhead in maintaining those streams is much higher relative to the actual data held in those streams (compared to the push approach). In order to take advantage of greater parallelism in the reducer stage (for the pull-based approach), we may wish to increase the number of reducers, which further exacerbates the problem.

Another advantage of the push-based approach is that reducers are only consuming from a single stream, so we would expect better reference locality (and the benefits of processor optimizations that may come from more regular memory access patterns) compared to the pull-based approach. The downside, however, is synchronization overhead since all the mapper threads are contending for write access to the reducer streams.

Hybrid Approach: As a middle ground between the pull and push approaches, we introduce a hybrid approach that devotes a small number of streams to each reducer. In Figure 4, each reducer reads from two streams, which is the default. There are two ways to distribute incoming (key, value) pairs to multiple streams for each reducer: the first is to distribute evenly, and the second is to distribute according to the current lock condition of a stream. The second approach is perhaps smarter, but Hone currently implements the first method, which we empirically discovered to work well. Having multiple streams reduces, but does not completely eliminate lock contention, but at the same time, the hybrid approach does not suffer from a proliferation of stream objects. The number of streams per reducer can be specified in a configuration, which provides users a "knob" to find a sweet spot between the two extremes.

In the push and hybrid data-shuffling approaches, lock efficiency plays an important role in overall performance. We have implemented and experimented with various lock implementations, including Java synchronization, test-and-set (tas) spin lock, test and test and set (ttas) spin lock, and reentrant lock. Each lock implementation has its own advantages and disadvantages, but overall we find that Java synchronization in JDK7 performs the best.

Tradeoffs: We experimentally compare the three different data-shuffling approaches, but we conclude this section with a discussion of the factors that may impact performance.

Obviously, input data size is an important factor. Larger inputs translate into more splits, more mappers, and thus more active streams that are held in memory (for the pull-based approach). In contrast, there are only r streams in the push-based approach, where r is the number of reducers. Note that the number of reducers is a user-specified parameter, unlike the number of mappers, which is determined by the input data. As previously discussed, the cost of fewer data streams (less object overhead) is synchronization costs and contention when writing to those streams. The hybrid approach tries to balance these two considerations.

Another factor is the amount of intermediate data that is produced. Some MapReduce jobs are primarily "reductive" in that they generate less intermediate data than input data, but other types of applications generate more intermediate data than input data; some text mining applications, for example, emit the cross product of their inputs [12]. This characteristic may have a significant impact on the performance of the three data-shuffling approaches.

Finally, the distribution of the intermediate keys will play an important role in performance—this particularly impacts synchronization overhead in the push-based approach. For example, with that approach, if the distribution is Zipfian (as with word count and certain types of graph algorithms), then increasing the number of reducers may not substantially lower contention, since the "head" of the distribution will always be assigned to a single reducer [13]. On the other hand, if the intermediate key distribution is more uniform, we would expect less lock contention since mapper output would be more evenly distributed over the reducer streams, reducing the chance that multiple mappers are contending for a lock.

4.2 Challenges and Solutions

This section discusses key challenges in developing Hone for the Java Virtual Machine on multi-core, shared-memory architectures and how we addressed them.

Memory consumption: To retain compatibility with Hadoop, we made the decision to implement Hone completely in Java, which meant contending with the limitations of the JVM. In a multi-core, shared-memory environment, the mapper, sort, and reducer threads compete for shared resources, and thus we must be careful about the choice of data structures, the number of object creations, proper de-referencing of objects for better garbage collection, etc. We discovered early that many Java practices scale poorly to large datasets.

With a naïve implementation based on standard Java collections, on a server with 128GB RAM, an initial implementation of MapReduce word count on an input size 10% of the total memory generated out-of-memory errors because standard Java collections are heavyweight [18]. For example, an implementation using a Java `TreeMap<String, Integer>` to hold intermediate data can have up to 95% overhead, i.e., only 5% of the memory consumed is used for actual data.

To address this issue, we extensively use primitive data structures such as byte arrays to minimize JVM-related overhead. In the mapper stage, (key, value) pairs are serialized to raw bytes and in the reducer stage, new object allocations are reduced by reading pairs from byte arrays using bit operations and reusing container objects when possible. We avoid using standard Java collections in favor of more efficient custom implementations.

Sorting is expensive: Intermediate (key, value) pairs emitted by the mappers need to be sorted by key. For large intermediate data (on the order of GBs), we found sorting to be a major bottleneck. This is in part because operations such as swapping objects can be expensive, but the choice of data structures has a major impact on performance also. In Hadoop, sorting is accomplished by a combination of in-memory and on-disk operations. In Hone, however, everything must be performed in memory.

We experimented with two approaches to sorting. In the first, each thread from the mapper thread pool handles both mapper execution as well as sorting. In the second approach, mapper execution and sorting are handled by separate thread pools. We ultimately adopted the second design. Note that sorting is orthogonal to the pull, push, hybrid data-shuffling approaches.

We see a number of advantages to our decoupled approach. First, the optimal thread pool size depends on factors such as the number of cores available, the size of the intermediate data, and skew in the intermediate key distribution. The decoupled approach lets us configure the sort thread pool size based on these considerations, independent of the mapper thread pool size. Second, the decoupled approach allows the garbage collector to clean up memory used by the mapper stage before moving to the sort stage. Finally, combining mapping and sorting creates a mix of different memory access patterns, which can negatively impact performance.

Hone implements a custom quick sort that works directly on byte arrays; these are the underlying implementations of the streams that the mappers write to when emitting intermediate data. The main idea is to store intermediate (key, value) pairs in a data byte array in serialized form, and to create an offset array that records offset information corresponding to the serialized objects in the data byte array. Offsets are also stored in byte arrays. Once mapper output is stored in these data and offset byte arrays, quick sort is applied. Offsets are read from the offset array and data are read using bit operations depending on the data type (to avoid object materialization whenever possible). Values are compared with each other, but only offset bytes are swapped. Usually, the size of the offset byte array is much less than the size of the data byte array, and therefore it is more efficient to perform swapping on the offset byte array. Moreover, most of the bytes in the offset byte array contain zeros (i.e., the high order bytes of an offset): only the non-zero bytes and the bytes that are not equal need to be swapped. This eliminates a large amount of the total cost of swapping elements during sorting.

Interactions between data shuffling and sorting: In the pull-based approach to data shuffling described in Section 4.1, the sort stage takes maximum advantage of parallelism since the intermediate data are divided among $m \times r$ streams (usually a large number). However, in the push and the hybrid approaches, intermediate data are held in a much smaller number of streams: r in the case of the push approach and a small factor of r for the hybrid approach. In both cases, this reduces the amount of parallelism available, since each sorting thread must handle a much larger amount of data. For large datasets, this becomes a performance bottleneck. For the push and hybrid approaches, we remedy this by splitting intermediate streams into several smaller streams on the fly. The sizes of these streams is a customizable parameter, but we have heuristically settled on a value that works reasonably well across different applications (10000 bytes).

Disk-based readers and writers: One of the challenges in developing a Hadoop-compatible MapReduce implementation is that Hadoop application code makes extensive use of disk-based readers and writers, mainly implemented using the RecordReader and RecordWriter classes. The simplest way to avoid disk overhead is to provide an API to access memory directly and then change the application code to take advantage of these hooks. Since we wanted to make Hone compatible with the existing Hadoop API, we needed deeper integration.

We introduce the notion of a *namespace*, which is a dedicated region in memory where data are stored. Application code can access namespaces through the standard Hadoop `Job` object. To maintain compatibility with the Hadoop API, we provide efficient in-memory alternatives for Hadoop FileReader and FileWriter classes. Disk paths in application code are automatically converted to appropriate namespaces and output is redirected to these namespaces.

Iteration support: Iterative MapReduce algorithms, where a sequence of MapReduce jobs are chained together such that the output of the previous reducer stage serves as the input to the next mapper stage, are a well-known weakness of Hadoop [4]. Since many interesting algorithms are iterative in nature (e.g., PageRank, LDA), this is an important problem to solve. The primary issue with Hadoop-based implementations of iterative algorithms is that reducer output at each iteration *must* be serialized to disk, even though it should be considered temporary since the data are immediately read by mappers at the next iteration. Of course, serializing serves the role of checkpointing and provides fault tolerance, but since Hadoop algorithms are forced to do this at *every* iteration, there is no way to trade off fault tolerance for speed.

In Hone, all of these issues go away, since intermediate data reside in memory at the end of each iteration. The choice to serialize data to disk can be made independently by the developer. Thus, Hone provides natural support for iterative MapReduce algorithms. In a bit more detail: typically, in an iterative algorithm, there is a “driver program” that sets up the MapReduce job for each iteration, checks for convergence, and decides if another iteration is necessary. Convergence checks are usually performed by reading reducer output (i.e., files on HDFS). In Hone, this is transparently handled by our notion of namespaces.

Garbage collection and off-heap memory allocation: All objects allocated in the JVM heap are scanned periodically by the garbage collector, with frequency determined in part by the size of the heap and in part by the rate at which new objects are created. This significantly impacts the overall performance of Hone, with a major culprit being the mappers, which generate a large number of objects to store the intermediate data. The techniques that we have discussed so far (e.g., using byte arrays to serialize objects, reducing the number of data streams, etc.) help in reducing the garbage collection overhead. As an additional optimization, we tried taking advantage of *off-heap memory*, via the `ByteBuffer` class in the Java NIO package. This allows us to manage the memory directly without interference, since the JVM garbage collector does not touch memory allocated in this fashion.

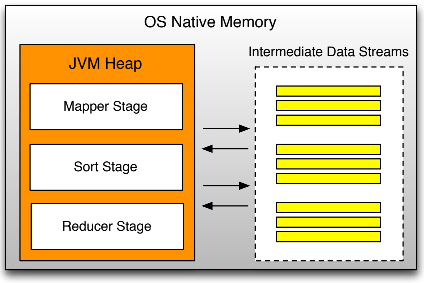


Figure 5: Offloading intermediate data to *off-heap* direct native memory.

Parameter	Description
s	Split size. Default value is 64MB.
r	Number of reducers.
tp_m	Mapper stage thread pool size. Default value is 15.
tp_r	Reducer stage thread pool size. Default value is 15.
tp_s	Sort stage thread pool size. Default value is 15.
arch	Approach to data shuffling: {*pull*, *push*, *hybrid*}. Default is *pull*.
lockType	Type of lock for *push* and *hybrid* data shuffling: Java synchronization, test-and-set (tas) spin lock, test and test and set (ttas) spin lock, and reentrant lock. Default is Java synchronization.
hs	Number of streams for each reducer for *hybrid* data shuffling.

Table 2: Description of Hone parameters.

Figure 5 shows the memory heap managed by JVM encapsulated under OS native memory, where Hone operates. Memory needed for various MapReduce stages in Hone is still allocated through the JVM heap, but intermediate data created by the mappers are offloaded into data streams that are created off-heap. These off-heap data streams are then accessed by the sort stage, which performs in-place sorting, and then finally handed over to the reducers. Since this optimization uses non-standard APIs, we evaluated its impact in a separate set of experiments (see Section 7.2), but we do not use off-heap memory for most of our experiments.

NUMA support: To perform NUMA-specific optimizations, a system must support the ability to pin threads to specific cores. Unfortunately, Java does not provide explicit support for CPU affinity, as the assignment of threads to cores is handled opaquely by the JVM. However, thread-to-core affinity constructs can be supported via a library in C/C++ and accessed in Java via JNI. Hone currently does not take advantage of such optimizations.

Cache locality: High-performance algorithms usually require very careful tuning to take advantage of cache locality and processor prefetching—these effects can be substantial [3]. Hone, however, does not currently implement any optimizations along these lines, primarily because the intermediate JVM abstraction often obscures the machine instructions that are being executed, compared to a low-level language such as C where the programmer retains greater control over the system. Nevertheless, there may be opportunities to gain greater efficiencies via more cache-conscious designs. For example, combiners might help an application optimize for cache locality, e.g., by performing aggregation while intermediate data still reside in cache. However, this must be balanced against the overhead of context switching (from mapper to combiner execution). In the future, it would be interesting to explore whether such cache optimizations can be reliably implemented on the JVM.

5. EXPERIMENTAL SETUP

5.1 Comparison Systems

Hone is open-source and can be downloaded at `hadoop1.org`. As previously discussed, it is implemented in Java for Hadoop API compatibility. There are a number of tunable parameters in Hone, which are summarized in Table 2. Unless otherwise specified, all experiments used default settings. We compared the performance of Hone against Hadoop PDM, a 16-node Hadoop cluster, a reimplementation of Phoenix in Java (described below), Phoenix system variants, and Spark. Details are provided below.

All single-machine experiments were performed on a server with dual Intel Xeon quad-core processors (E5620 2.4 GHz) and 128GB RAM. This architecture has a 64KB L1 cache per core, a 256KB L2 cache per core, and a 12MB L3 cache shared by all cores of a single processor. With hyperthreading, this machine can support up to 16 threads concurrently. The machine has six 2TB 7200 RPM disks, each with 64MB cache, arranged in a RAID6 configuration. For Hadoop PDM, we ran Hadoop YARN 2.0.3; the configuration parameters were set for the maximum allowable in-memory buffer sizes, but note that Hadoop buffer sizes are limited to 32-bit integers. For comparisons with Phoenix, we ran Phoenix2 and Phoenix++; for Spark, we used version 0.8.0. Our Hadoop cluster ran CDH4 (YARN) and comprises 16 compute nodes, each of which has two quad-core Xeon E5520 processors, 24GB RAM, and three 2TB disks. Note that with YARN, however, one node serves as the Application Master (AM), leaving only 15 actual worker nodes. The Hadoop cluster ran JDK6, whereas we used JDK7 everywhere else. Note that both the individual server and the Hadoop cluster were purchased around the same time; since they represent hardware from the same generation, our experiments fairly capture the capabilities of a high-end server and a modest Hadoop cluster.

Java implementation of Phoenix. The Phoenix project [19, 24, 23] shares similar goals as Hone in terms of exploring MapReduce implementations for shared-memory systems. The biggest difference, however, is that Phoenix is implemented in C/C++ and thus abandons Hadoop API compatibility—this means that Hadoop applications need to be rewritten to take advantage of Phoenix.

Another substantial difference between Phoenix and Hone is how intermediate data are shuffled from the mappers to the reducers. Whereas Hone uses the pull, push, and hybrid approaches discussed in Section 4.1, Phoenix uses an array of hash tables: the array contains one hash table per mapper, and the entire data structure can be visualized as a 2D grid. Each hash table entry has an array of keys that hash to that location, and each key points to an array of associated values. Conceptually, we can think of each mapper as writing to a “column” in the 2D data grid. The reducers scan entries of the hash tables belonging to all the mappers to grab the appropriate intermediate (key, value) pairs; conceptually, we can think of this as reading the “rows” in the 2D data grid.

It did not appear to us that the Phoenix data-shuffling approach can be efficiently implemented in Java, but we nevertheless attempted an adaptation to help us better understand the differences between the languages and the unique challenges that Java imposes. Due to the overhead of materializing key and value objects in Java, a straightforward implementation was utterly infeasible. Instead, we opted to minimize object overhead by replacing Phoenix’s emission values arrays with byte arrays containing serialized data. The main data structure is a Java `HashMap` array, with one map per mapper, where the map keys are the intermediate keys and the map values are the intermediate values (associated with the key), both in serialized form. In order to support this data structure we also needed to create a second `HashMap` array, one map per mapper, so that reducers would know with which keys to query the main data structure. In each map, the map key is the reducer id and the map

Application	Dataset	Small	Medium	Large
Word Count (WC)	Wiki articles	128MB, 256MB, 512MB	1GB, 2GB	4GB, 8GB, 16GB
K-means (KM)	3D points	12M, 25M	51M, 102M	204M, 398M
Inverted Indexing (II)	Wiki articles	128MB, 256MB, 512MB	1GB, 2GB	4GB, 8GB, 16GB
PageRank (PR)	Wiki graph	0.4M, 0.8M	1.8M, 3.5M	7.2M
LDA	TREC docs	125MB, 256MB	512MB	1GB

Table 3: Data conditions for each application. For *k*-means and PageRank, we show the number of points and number of graph nodes, respectively.

value is the reducer's keylist—once again, held in serialized form. Whenever a mapper emits an intermediate (key, value) pair, the system determines which reducer has ownership of that key and writes the key to that reducer's keylist. We feel that we have accurately captured the data-shuffling approach of Phoenix, and that our implementation represents a reasonable attempt to study how the "data grid" design would fare in Java.

5.2 Applications and Datasets

Our experiments explored a range of MapReduce applications, described below (see Table 3):

- Word Count (WC): This application counts the frequencies of all words in a collection of text documents.
- *K*-means (KM): This application implements *k*-means clustering using Lloyd's Algorithm. Since the algorithm is iterative, reducer output is passed to the input of the next mapper stage through the Hone namespace manager. These iterations proceed until convergence.
- Inverted Indexing (II): This application builds a simple inverted index, which comprises a mapping from terms to postings which hold information about documents that contain those terms. An inverted index is the core data structure used in keyword search.
- PageRank (PR): This application computes PageRank, the stationary distribution of a random walk over a graph. Like *k*-means clustering, this algorithm is iterative.
- Latent Dirichlet Allocation (LDA): This application builds topic models over text documents using Latent Dirichlet Allocation [2] via variational inference. The implementation represents a substantial research effort [26] and demonstrates Hone on a non-trivial application. This algorithm is also iterative.

In terms of datasets, for word count and inverted indexing, we used articles from English Wikipedia totaling 16GB. For *k*-means clustering, we randomly generated 398 million 3D coordinates; in all our experiments we ran clustering with $k = 10$. For PageRank, we used a Wikipedia article link graph that contains 7.2M nodes. For LDA, we used a document collection from the Text Retrieval Conference (TREC) that totals 1GB [26].

One important variable in our experiments is the amount of data processed. To examine these effects, we generated subsets of varying sizes from the above datasets. We also divided the data conditions somewhat arbitrarily into small, medium, and large categories, summarized in Table 3.

In addition to the above real applications and datasets, we built a synthetic workload generator to better understand Hone performance under different workloads. Details of these experiments are discussed in Section 7.

6. APPLICATION RESULTS

In this section, we present results of experiments that compared Hone against a variety of other systems on the applications described in Section 5.2. To thoroughly characterize performance, we varied both the amount of compute resources available as well as the amount of data processed.

6.1 Strong Scalability Analysis

In a strong scalability analysis, the problem size stays fixed but the number of processing elements varies. A program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used (N). Maintaining strong scalability is challenging because coordination overhead increases with the number of processing elements.

If the amount of time to complete a work unit with one processing element is t_1 and the amount of time to complete the same unit of work with N processing elements is t_N, the strong scaling efficiency (SSE), as a percentage of linear, is given as follows:

$$\text{SSE} = \left(\frac{t_1}{N \cdot t_N}\right) \times 100\% \qquad (1)$$

Table 4 shows the strong scalability results for Hone and Hadoop PDM on different applications. In all cases, the experiments ran on the server described in Section 5.1 and Hone used the pull-based data-shuffling approach. We increased the number of threads from 1 to 16 (by varying the thread pool sizes) while keeping the dataset size constant; the table shows running time in seconds and the strong scaling efficiency based on Equation (1). For iterative algorithms the reported values capture the running time of the first iteration. Speedup of Hone over Hadoop PDM is summarized in Table 5. These experiments used the large dataset condition for each application: for word count and inverted indexing, 8GB; for *k*-means, 398M 3D points; for PageRank, the Wikipedia article link graph with 7.2M nodes; for LDA, 1GB. For both Hone and Hadoop PDM, the number of mappers was determined automatically based on the split size, which is 64MB in our case. We used binary search to determine the optimal number of reducers, and figures for both Hone and Hadoop PDM are reported with optimal settings.

From these results we see that Hone is substantially faster than Hadoop PDM in terms of absolute running time. Keep in mind that our machine has only 8 physical cores, so the runs with 16 threads are taking advantage of hyperthreading. Overall, Hone exhibits much better strong scaling efficiency, outperforming Hadoop PDM in nearly all conditions. It is interesting to see that efficiency varies by application—for example, Hone achieves over 90% efficiency up to 8 threads on LDA, but for inverted indexing and PageRank, performance deteriorates substantially as we increase the thread count. From Table 5, we find no discernible trend on the relative performance of Hone compared to Hadoop PDM as the number of threads increases for the five applications.

6.2 Weak Scalability Analysis

In a weak scalability analysis, the problem size (i.e., workload) assigned to each processing element stays constant and additional processing elements are used to solve a larger overall problem. In this case, linear scaling is achieved if the running time stays constant while the workload is increased in direct proportion to the number of processing elements.

If the amount of time to complete a work unit with one processing element is t_1, and the amount of time to complete N of the same work units with N processing elements is t_N, the weak scaling effi-

threads	Hone										PDM									
	WC		KM		II		PR		LDA		WC		KM		II		PR		LDA	
2	946	100%	480	80%	1124	92%	14.0	73%	6296	97%	2309	54%	2096	64%	3200	63%	92	82%	23673	74%
4	490	97%	281	68%	671	77%	10.8	56%	3160	97%	1498	54%	1676	40%	2102	47%	74	50%	12421	70%
8	292	81%	185	52%	480	54%	9.5	27%	1675	91%	1089	28%	1100	31%	1501	33%	69	27%	10024	44%
16	253	47%	166	29%	405	32%	7.8	17%	957	78%	837	18%	849	19%	1383	18%	66	14%	8002	27%

Table 4: Strong scalability experiments with Hone and Hadoop PDM: cells show running time in seconds and the strong scaling efficiency.

threads	WC	KM	II	PR	LDA
2	2.5×	4.5×	3×	6.5×	4×
4	3×	6×	3×	7×	4×
8	3.5×	6×	3×	7×	6×
16	3×	5×	3.5×	8.5×	8×

Table 5: Speedup of Hone over Hadoop PDM based on the strong scalability results in Table 4.

ciency (WSE), as a percentage of linear, is given as follows:

$$\text{WSE} = \left(\frac{t_N}{t_1}\right) \times 100\% \qquad (2)$$

Figure 6 shows running times for Hone with different numbers of threads and Table 6 provides the weak scalability efficiency computed from those results. In all cases, the experiments ran on the server described in Section 5.1. For these experiments, Hone used the pull-based data-shuffling approach and the number of reducers was tuned using binary search, as with the strong scalability analysis. For each application we increased the dataset size with the number of threads. For word count and inverted indexing, the dataset size varied from 128MB to 2GB; for *k*-means, from 12M to 204M points; for PageRank, from 0.4M nodes to 7.2M nodes; for LDA, from 125MB to 1GB. Note that for the LDA application we did not have sufficient data to run 16 threads, so that data point was omitted from this experiment. For iterative algorithms the reported values capture the running time of the first iteration.

As with the strong scalability experiments, we see that efficiency varies by application. Again, since our machine has only 8 physical cores, it is no surprise that efficiency falls off dramatically with 16 threads, since they may be contending for physical resources. Inverted indexing and PageRank perform poorly, while the three remaining applications appear to scale better. In particular, both word count and LDA scale almost linearly up to the physical constraints of the hardware, and *k*-means scales well up to 4 threads. Note that this experiment used the pull-based approach to data shuffling for all data conditions, even though experiments below show that for larger datasets, a hybrid approach works better. This means that we can achieve even higher weak scaling efficiency if we dynamically adjust the data-shuffling approach.

6.3 Comparison of Hadoop Implementations

In the next set of experiments, we compared Hone with Hadoop PDM, our fully-distributed 16-node Hadoop cluster, and our Java implementation of Phoenix. Running times for the five sample applications on varying amounts of data are shown in Figure 7. In the legend, "H-Pull", "H-Push" and "H-Hybrid" refer to the pull, push, and hybrid data-shuffling approaches in Hone (for the hybrid approach, each reducer works on two streams). "Hadoop PDM" refers to Hadoop pseudo-distributed mode. "Hadoop Cluster" refers to Hadoop running on the 16-node cluster. "Phoenix" refers to our implementation of Phoenix in Java. Note that Hone, Hadoop PDM, and our Java Phoenix implementation ran on the same machine; in all cases we fully utilized the machine with 16 threads. For iterative algorithms the reported values capture the

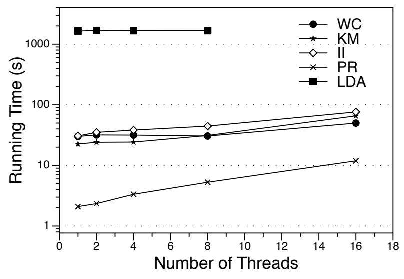


Figure 6: Results of weak scalability experiments with Hone.

threads	WC	KM	II	PR	LDA
2	95%	94%	88%	90%	98%
4	95%	94%	76%	63%	99%
8	99%	61%	66%	40%	99%
16	61%	30%	41%	18%	-

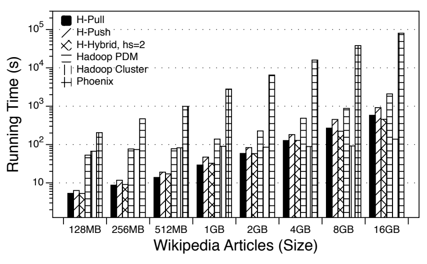
Table 6: Weak scaling efficiency based on Figure 6.

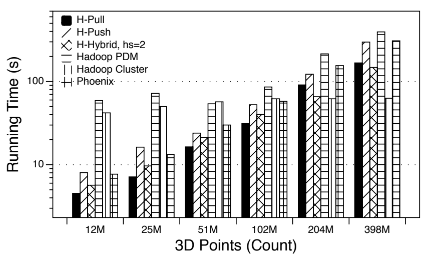
running time of the first iteration. As before, we report results with the optimal reducer setting discovered via binary search.

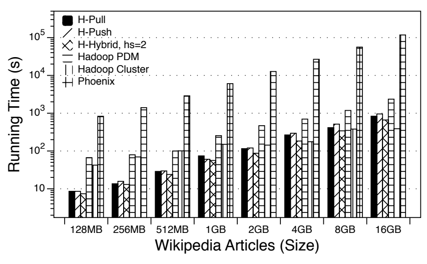
We summarized the speedup comparing Hone to Hadoop PDM and the Hadoop cluster as follows: For each data size category (small, medium, and large), we considered only the largest data condition for each application, as shown in Table 3. For example, with word count we considered the 512MB, 2GB, and 16GB conditions. For each data condition, we selected the fastest from the {pull, push, hybrid} approaches and divided that running time by the running time of either Hadoop PDM or the Hadoop cluster. If the value is greater than one, indicating that Hone is slower, we take the negative; otherwise, we take the inverse, indicating that Hone is faster. These values are reported in Table 7.

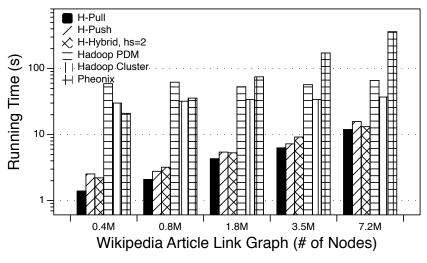
For word count, as we can see from Figure 7(a), the pull-based approach to data shuffling is the fastest for small to medium datasets. With large datasets, however, the hybrid approach appears to be the fastest. In all cases, the push-based approach is the slowest of the Hone configurations. Hone performs substantially faster than Hadoop PDM across all data conditions and is faster than the 16-node Hadoop cluster on small and medium datasets. The latter finding is not surprising since Hadoop jobs on a distributed cluster have substantial startup costs relative to the amount of data processed. We find that our Java Phoenix implementation is very slow, sometimes by up to two orders of magnitude compared to Hone.

Results for *k*-means clustering and inverted indexing follow the same general trends as word count: pull-based data shuffling is faster on smaller datasets, but the hybrid approach is faster on bigger datasets. Hone is faster than Hadoop PDM across all dataset sizes, and it is faster than the 16-node Hadoop cluster on small to medium datasets. However, the Hadoop cluster is faster on the large datasets. For inverted indexing, the Java implementation of Phoenix is terrible, just like in word count. However, for *k*-means, the performance gap between Java Phoenix and Hone is substantially smaller—in some cases, the performance of Java Phoenix


(a) Word Count


(b) *K*-means


(c) Inverted Indexing


(d) PageRank

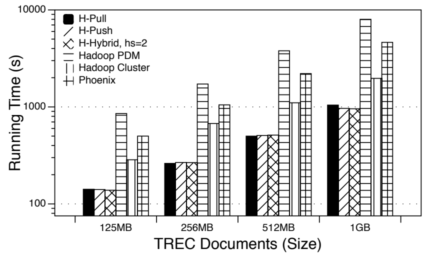

(e) LDA

Figure 7: Comparing Hone, Hadoop PDM, the 16-node Hadoop cluster, and our Java Phoenix implementation on five different applications with varying amounts of data. Note that the *y*-axis is plotted on a log scale.

Hone vs. Hadoop PDM

	WC	KM	II	PR	LDA
small	6×	14×	4×	30×	7×
medium	4×	3×	5×	9×	8×
large	5×	3×	4×	6×	8×

Hone vs. Hadoop cluster

	WC	KM	II	PR	LDA
small	6×	7×	4×	15×	3×
medium	2×	2×	2×	5×	2×
large	−2×	−2×	−2×	3×	2×

Table 7: Relative performance of Hone compared to Hadoop PDM (top) and the 16-node Hadoop cluster (bottom) for different data conditions. Negative values indicate that Hone is slower than the comparison system.

approaches Hone configurations. We believe that this is because *k*-means generates far less intermediate data than inverted indexing or word count, which confirms our thinking all along—that the optimization of intermediate data structures for data shuffling is the key to achieving high performance.

We notice a different pattern for PageRank and LDA: Hone is faster than both Hadoop PDM and the Hadoop cluster across all data conditions. This is perhaps due to the relatively small size of the datasets—the graph is relatively small, and the document collection for LDA is smaller than the Wikipedia articles used in word count and inverted indexing. Thus, these results appear to be consistent with the above findings. The pull-based approach outperforms all others for PageRank, but all three data-shuffling approaches are roughly equal for LDA.

Summarizing these results, our experiments suggest a few key takeaways. For small to medium datasets, the pull-based approach to data shuffling seems to be the fastest, but for large datasets, the hybrid approach can be faster—however, there are application-specific differences, such as with LDA. In our applications, we did not find a case where the push-based approach was convincingly better, which suggests that contention on the reducer streams and synchronization overhead significantly impacts performance.

In all these experiments, Hone is faster than Hadoop PDM, and in some cases, faster than the 16-node Hadoop cluster as well. For the cluster results, one can criticize that we have not used sufficiently large datasets to make distributed Hadoop worthwhile—but that's exactly the point we are trying to make. As individual servers grow in memory capacity and core count, the sizes of datasets that can be handled in memory grows as well, and in these cases, a scale-up solution is perhaps preferable to a scale-out solution.

Finally, comparisons to our Java Phoenix implementation show that intermediate data structures for data shuffling need to be specifically designed for the execution environment. We adapted an approach that works well for C/C++, but translates into an inefficient design in Java. This shows, not surprisingly, that optimizations need to be targeted to the specific execution environment.

6.4 Comparison with Other Systems

Although Phoenix and Spark have very different designs compared to Hone, we believe that a comparison is still instructive. Here, we describe experiments on the word count and *k*-means clustering applications with varying amounts of data.

Phoenix2 [24] and Phoenix++ [23] are implemented in C/C++. Thus, a comparison against Hone gives us a sense of how much the choice of language matters. We downloaded both systems and ran an "out of the box" evaluation with default settings on word count and *k*-means clustering (both were existing implementations). We used the same server as all our other single-machine experiments. Relative performance is shown in Table 8 for word count (top) and *k*-means (bottom): positive values indicate that Hone is faster and negative values indicate that the comparison system is faster.

The earlier system, Phoenix2, is actually slower than Hone on word count, and has scalability limitations—throwing segmentation faults beyond 2GB. Phoenix++, on the other hand, is 2–3× faster than Hone on word count up to 8GB, but has roughly comparable performance at 16GB. For *k*-means clustering, both Phoenix2 and Phoenix++ are substantially faster than Hone, by a factor of up to 7× on larger data. The symbol $\sim$ indicates that speed is roughly the same as Hone. Note that Phoenix++ performs "in-mapper combining" [14] and requires the developer to specify a data structure to store intermediate data (based on the application type), which gives it a performance advantage over Phoenix2 and Hone. How-

Word Count

	128MB	256MB	512MB	1GB	2GB	4GB	8GB	16GB
Phoenix2	2×	2×	3×	2.4×	2.1×	-	-	-
Phoenix++	−2×	−3×	−2×	−2×	−2×	−3×	−2×	∼
Spark	3×	3×	3×	2×	2×	2×	∼	∼

K-means

	12M	26M	51M	102M	204M	398M
Phoenix 2	−4.5×	−3.5×	−4×	−4.5×	−6.5×	−6.4×
Phoenix++	−5×	−4×	−5×	−6×	−7.5×	−7×
Spark	4×	3×	3×	2×	2×	3×

Table 8: Relative performance of Hone compared to Phoenix2, Phoenix++, and Spark for word count (top) and k-means (bottom). Negative values indicate that Hone is slower than the comparisons system. Note that Phoenix2 does not scale beyond 2GB and terminates with a segmentation fault. The symbol ∼ indicates that speed is roughly the same as Hone.

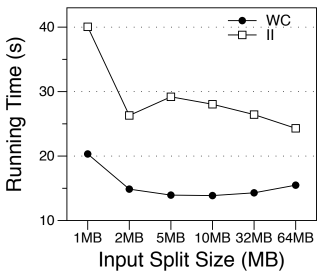


Figure 8: Running time of word count and inverted indexing on the 512MB dataset with different split sizes (running 16 threads).

ever, this substantially alters the MapReduce model. These results show that there can be significant performance advantages to adopting C/C++, but at the cost of abandoning Hadoop compatibility.

In our evaluation of Spark, we loaded all data into memory (via the RDD abstraction) and allowed the system to fully utilize hardware resources (16 threads). These evaluations were performed on the same server as all the other single-machine experiments, and we used existing word count and k-means implementations. Results are shown in Table 8 for word count (top) and k-means (bottom). For word count, Hone is faster than Spark on datasets up to 4GB and roughly comparable in terms of speed for larger datasets. For k-means, Hone is consistently faster than Spark for all dataset sizes. We are quick to emphasize that this is inherently an apples-to-oranges comparison because Hone and Spark are very different. Whereas Spark provides a general data processing model, Hone is limited to MapReduce. Spark was designed for distributed execution on a cluster, whereas Hone was specifically optimized for running on a single machine. However, since Spark and Hone both run on the JVM, the performance differences gives us a sense of the gains that might be attributed to careful engineering against the characteristics of the platform.

6.5 Effects of Input Split Size

In standard distributed Hadoop, input splits for the mappers are aligned with HDFS blocks so that tasks can be efficiently placed on machines where the blocks are held locally. Because Hone runs on a single machine, this constraint is not applicable and thus we have greater flexibility in tuning the split size. There are two considerations to balance when setting a value: Smaller splits lead to more mapper tasks and thus generate more opportunities to extract parallelism. On the other hand, if the split size is too small, the mapper threads don't have sufficient work to perform, and thus we waste time context switching.

We performed an experiment to empirically determine how these two factors play out. Figure 8 plots the execution time of word count and inverted indexing for different split sizes on the 512 MB dataset (using 16 threads). As we can see, for word count the optimal split size is 5MB, whereas for inverted indexing, the we achieve the fastest running time with 64MB.

Parameter	Description
d_m	Mapper emit distribution; possible values are {*one-to-one*, *many-to-one*, *one-to-many*}.
λ	For d_m = *one-to-many*, the number of intermediate (key, value) pairs to emit per token is decided by drawing from a Poisson distribution with mean λ. Default value is 10.
ψ	For d_m = *many-to-one*, the probability to emit an intermediate (key, value) pair per token. Default value is 0.1.
d_i	Intermediate key distribution; possible values are {*uniform*, *Zipfian*, *biased*}.
$zipf_{skew}$	For d_i = *Zipfian*, the Zipfian skew parameter. Default value is 10.
α	For d_i = *biased*, probability an intermediate (key, value) pair will be sent to a single "special" reducer. Default value is 0.7.
ps	Payload size, the length of the randomly-generated string that serves as the intermediate value. Default value is 3.
b_{cpu}	Parameter to control CPU-intensiveness of the workload (in the mapper). For each token, value of π is calculated to b_{cpu} digits before generating intermediate data. Default value is 2.

Table 9: Description of workload generator parameters.

7. SYNTHETIC WORKLOAD RESULTS

To better understand the effects of different job characteristics on Hone, we built a synthetic workload generator that allows us to create different types of MapReduce jobs. Whereas our sample applications encode a specific set of fixed characteristics, the workload generator lets us vary those characteristics independently. We also used this tool to evaluate the impact of the off-heap memory allocation optimization.

7.1 Workload Generator

The basic structure of the synthetic job is similar to the word count application, but with a number of adjustable parameters, shown in Table 9. The job takes as input a collection of text documents to simulate actual data processing; the mapper tokenizes each document and processes each token in turn. What happens next depends on the workload parameters (details below), but intermediate (key, value) pairs are generated with an integer between 1 and 10,000 as the key and a random string as the value. The reducers simply count the number of values that are associated with each key and output the final counts.

Mapper emit distribution parameter (d_m) determines the number of intermediate (key, value) pairs emitted *per token* in the mapper. This parameter attempts to model the fact that some MapReduce algorithms generate more intermediate data than input data, while others generate less. The possible settings are {*one-to-one*, *many-to-one*, *one-to-many*}.

Implementing the *one-to-one* setting is straightforward: for each input token the job emits an intermediate (key, value) pair, based on constraints specified below. For the *one-to-many* setting, we need a mechanism to stochastically determine the number of (key, value) pairs to emit for each token. For this we draw from a Poisson distribution with a mean of λ (set to 10 by default). In the *many-to-one* case, for every token, we generate an intermediate (key, value) pair with probability ψ. The default value of ψ is 0.1, which means that one intermediate pair is emitted every ten tokens on average.

Intermediate key distribution parameter (d_i) determines how intermediate (key, value) pairs are assigned to reducers. For example,

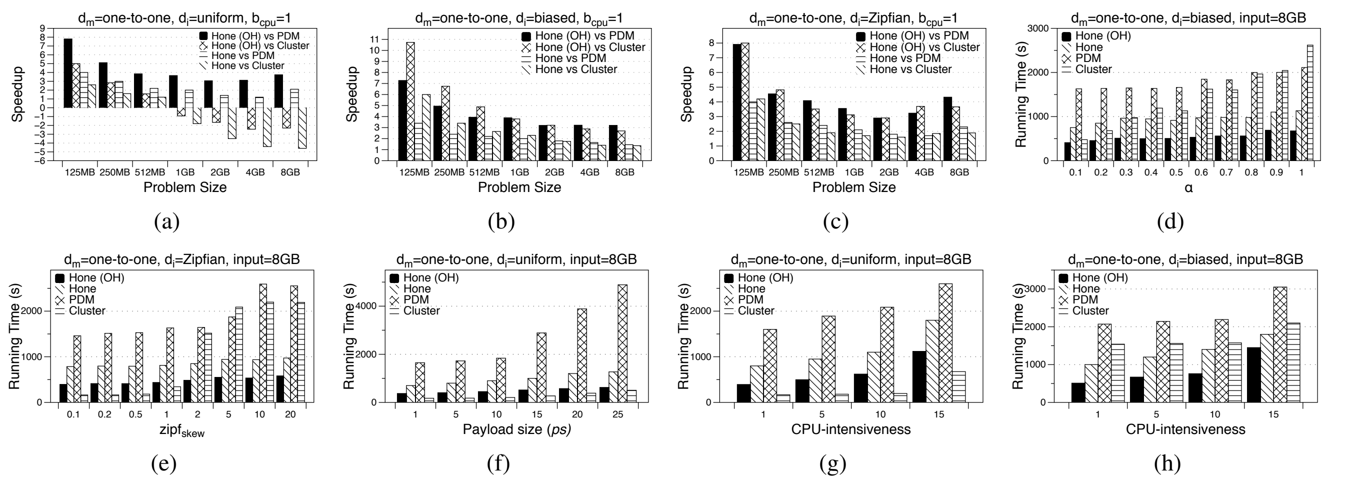


Figure 9: Experimental results using our synthetic workload generator.

word count exhibits a Zipfian intermediate key distribution since term occurrences are (roughly) Zipfian (i.e., lots of occurrences of common words and a long tail). Graph algorithms often behave similarly due to the presence of "supernodes", or nodes with many incoming edges. On the other hand, intermediate data in *k*-means clustering is uniformly distributed.

In order to capture these different characteristics, we provide a parameter d_i that can be set to $\{$*uniform*, *Zipfian*, *biased*$\}$. In all cases the intermediate key is an integer between 1 and 10,000, but the selection of the key is based on this setting:

- In the case of *uniform*, the integer is selected based on a uniform distribution.
- In the case of *Zipfian*, the key is drawn from a Zipfian distribution with skew parameter $zipf_{skew}$, with a default value of 10.
- In the case of *biased*, the intermediate key is selected such that each emitted (key, value) pair is assigned to a "special" reducer with probability α, or is otherwise assigned to one of the other reducers with uniform probability. This means that if we have r reducers, one reducer will receive α fraction of all intermediate data, while the remaining data will be distributed evenly among the $r-1$ other reducers. The default value of α is 0.7.

Payload Size (***ps***) determines the size of the intermediate value. In word count the payload is always an integer, but other MapReduce applications may emit bigger values that have complex internal structure. In the workload simulator, the value in the intermediate (key, value) pair is a randomly-generated string of length *ps*, with three as the default value.

CPU-intensiveness parameter (b_{cpu}) controls the amount of processing that is performed in the mapper in the simulated workload. To simulate workloads that are CPU-intensive to varying degrees, we compute π to b_{cpu} digits for each token before proceeding to generate intermediate output. The default value is two.

7.2 Summary of Findings

We have been exploring the performance of Hone and other systems under different workloads using the synthetic workload generator. This is the subject of ongoing explorations, but in Figure 9 we share a few of our initial findings. In particular, we have been using this approach to examine the performance impact of the off-heap memory optimization discussed in Section 4.2; the is abbreviated "OH" in the figure legends. The basic setup is the same as in all the experiments above, comparing Hone (using the pull-based approach to data shuffling) with Hadoop PDM and the 16-node Hadoop cluster. As with before, all in cases we used binary search to find the optimal number of reducers, and report results based on those settings.

Hone gracefully handles skew: Figure 9(a) shows that if the intermediate key distribution is uniform, then Hone is faster than Hadoop PDM for all data conditions examined and Hone is faster than the full Hadoop cluster for smaller datasets. This finding is consistent with the results from Section 6. However, for non-uniform intermediate key distributions, Hone appears to be faster than Hadoop PDM and the full Hadoop cluster for the data conditions we examined; this is shown in Figures 9(b), 9(c), 9(d), and 9(e). Skew creates stragglers (tasks that take substantially more time than the others), which is a well-known issue for Hadoop in a distributed environment [13, 11], and the effects appear to carry over to Hadoop PDM as well. On the other hand, the design of Hone makes it more resistant to skew effects.

Hone is less sensitive to payload size: Increasing the payload size increases disk activity and increases pressure on the communication channels for Hadoop PDM and the Hadoop cluster. On the other hand, Hone appears to be relatively insensitive to the payload size since everything is held in memory (provided, of course, that we have sufficient memory). This result is shown in Figure 9(f).

Hone effectively utilizes CPU resources: As the workload becomes more CPU-intensive, Hone is able to effectively utilize available cores; see Figures 9(g) and 9(h). Our single server has only 8 physical cores, so at some point we begin to saturate all available compute capacity—the full Hadoop cluster obviously has an advantage here because it has more cores. In Figure 9(h), we make an interesting observation: for Hadoop PDM and the Hadoop cluster, increasing the CPU-intensiveness parameter (at least up to 10) does not have much of an impact on the overall running time, which suggests that there are bottlenecks elsewhere (e.g., I/O and skew issues). In this sense, Hone achieves a better balanced design.

Hone off-heap can be up to 2× faster than Hone on-heap: With Hone, offloading intermediate data to *off-heap* native direct memory consistently improves performance, compared to the default setting where all intermediate data are stored on the JVM heap.

8. CONCLUSIONS

As others have suggested, we need to re-think scale-out vs. scale-up architectures as the amount of cores and memory on high-end commodity servers continues to increase. There is no doubt that the *total* amount of data is also growing rapidly, but it is unclear if the datasets used in *typical* analytical tasks today are increasing as fast. The crux of the scale-out vs. scale-up debate hinges on these relative rates of growth: server capacities are (roughly) growing with Moore's Law, which should continue for at least another decade. If dataset sizes are growing at a slower rate, then scale-up architectures will become increasingly attractive.

Ultimately, the datacenter is likely to consist of a mix of scale-out and scale-up systems—we will continue to run large, primarily disk-based jobs to scan petabytes of raw log data to extract interesting features, but this paper explores the interesting possibility of switching over to a multi-core, shared-memory system for efficient execution on more refined datasets. With Hone, this can all be accomplished without leaving the comforts of MapReduce: we simply select the most appropriate execution environment based on dataset size and other characteristics of the workload. This brings us to the biggest limitation of our current work and the subject of ongoing research—how to *a priori* determine the best Hone configuration in terms of the data-shuffling approach, split size, thread pool sizes, etc. In the future, we can imagine an optimizer that is able to examine a Hadoop workload and *automatically* decide what job to run where and the optimal parameter settings.

9. ACKNOWLEDGMENTS

This work has been supported by NSF under awards IIS-0916043, CCF-1018625, IIS-1144034, and IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsor. The last author is grateful to Esther for her loving support and dedicates this work to Joshua and Jacob.

10. REFERENCES

[1] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? *SoCC*, 2013.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 2003.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *VLDB*, 2010.

[5] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. *PACT*, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[7] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li. A novel approach to improving the efficiency of storing and accessing small files on Hadoop: a case study by PowerPoint files. *SCC*, 2010.

[8] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce system with an alternate API for multi-core environments. *CCGRID*, 2010.

[9] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. *SIGMOD*, 2012.

[10] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin. Hone: "scaling down" hadoop on shared-memory systems. *VLDB Demo*, 2013.

[11] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in MapReduce applications. *SIGMOD*, 2012.

[12] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. *SIGIR*, 2009.

[13] J. Lin. The curse of Zipf and limits to parallelization: A look at the stragglers problem in MapReduce. *LSDS-IR Workshop*, 2009.

[14] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[15] X. Liu, J. Han, Y. Zhong, C. Han, and X. He. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. *CLUSTER*, 2009.

[16] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report CSAIL-TR-2010-020, MIT, 2010.

[17] McObject LLC. In-memory database systems: Myths and facts, 2010.

[18] N. Mitchell and G. Sevitsky. Building memory-efficient Java applications: Practices and challenges. *PLDI Tutorial*, 2009.

[19] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. *HPCA*, 2007.

[20] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. *HotCDP*, 2012.

[21] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *VLDB*, 2012.

[22] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. *IPDPS*, 2011.

[23] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. *MAPREDUCE*, 2011.

[24] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. *IISWC*, 2009.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.

[26] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja. Mr. LDA: A flexible large scale topic modeling package using variational inference in MapReduce. *WWW*, 2012.