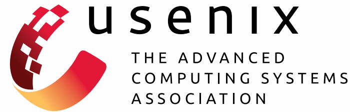

Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems

Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin,
The University of Texas at Austin

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/wang_yang

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems

Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, Mike Dahlin
The University of Texas at Austin

Abstract

This paper presents Exalt, a library that gives back to researchers the ability to test the scalability of today's large storage systems. To that end, we introduce Tardis, a data representation scheme that allows data to be identified and efficiently compressed even at low-level storage layers that are not aware of the semantics and formatting used by higher levels of the system. This compression enables a high degree of node colocation, which makes it possible to run large-scale experiments on as few as a hundred machines. Our experience with HDFS and HBase shows that, by allowing us to run the real system code at an unprecedented scale, Exalt can help identify scalability problems that are not observable at lower scales: in particular, Exalt helped us pinpoint and resolve issues in HDFS that improved its aggregate throughput by an order of magnitude.

1 Introduction

This paper presents Exalt, a library that gives back to researchers the ability to verify the scalability claims of today's large storage systems, which, ironically, have become hard to corroborate precisely because of the scale of these systems.

The advent of Big Data has strained the scalability of traditional storage systems, and several new architectures have been proposed to respond to this challenge [2–4, 7, 12, 13, 17, 22] by supporting up to hundreds of petabytes of storage and tens of thousands of storage nodes. Testing systems at such scale, however, requires access to tens of thousands of machines and at least as many disks, and few researchers have access to resources that plentiful: the rest of us have to design systems that are supposed to operate at a scale much larger than the infrastructure available to test them. Nor are such resource limitations affecting only academia: even industrial researchers who are within reach of clusters of the necessary size may not be able to reserve them for large scale experiments, since these clusters are a primary source of revenue.

These limitations are typically sidestepped in one of two ways. The first is to run experiments on a medium-sized cluster (100-200 machines) and extrapolate the results to larger scales. While this may work reasonably well in some cases, the fundamental assumption on which it rests—that resource consumption increases linearly with the load and the number of machines in the system—does not always hold, as we show in Section 2. To make matters worse, sources of non-linear growth are sometimes hard or impossible to observe in small deployments. For example, the time needed to add a new block to an HDFS file [22] increases with the file's size, but it is only after that size has grown beyond what is likely to be observable in small deployments that the slowdown becomes a limiting factor for the system's performance.

The second common approach for predicting the behavior of large-scale systems is simulation [18, 24, 26]. Unfortunately, the results of a simulation are only as accurate as the model on which the simulation relies; as systems grow in size and complexity, modeling them faithfully becomes prohibitive.

This paper proposes a third way: the Exalt library offers researchers the ability to test the scalability of a large-scale storage system by running its real code, but without requiring access to thousands of machines. The basic insight at the core of Exalt is that, in many large-scale experiments, how data is processed is not affected by the content of the data being written, but only by its size. Exalt leverages this freedom by virtualizing the data, while keeping the metadata intact to ensure that the system continues to function correctly. Specifically, Exalt clients write data in a specific format, *Tardis*, that has two key advantages. First, it allows Exalt to compress the behavior of the system in both space and time. Space compression is a powerful tool for performing large-scale experiments: for example, running 10,000 storage nodes on just 100 machines can bring to light previously unknown scalability bottlenecks in the metadata service. Since compressed

data takes much less time to write, compression in space can in turn result in compression in time: with the system running faster, bugs and performance issues can be discovered more rapidly.

The second key advantage of Tardis is that it addresses a fundamental challenge in virtualizing data: being able to distinguish data from metadata. While the content of the former is not important for the system to function correctly and can therefore be virtualized, the integrity of the latter is essential. This problem is particularly prominent in modern storage systems, which employ a two-layer architecture where the upper layer uses the lower layer as black-box storage: files written to the lower layer contain both data and metadata, which look indistinguishable to the lower layer. The need to ensure the integrity of the metadata is why approaches that virtualize data by altogether disposing of file contents (e.g. [1]) cannot be used in our context.

In summary, this paper makes the following contributions:

- We introduce Tardis, a data representation scheme that allows data to be identified and efficiently compressed even at lower-level storage layers that are not aware of the semantics and formatting used by higher levels of the system. Tardis provides transparent, lossless, computationally efficient compression of data and achieves high compression ratios.

- We present a methodology that utilizes Tardis to test the scalability and robustness of large-scale storage systems: our goal is not to predict every aspect of the performance of such systems (e.g. their power consumption) but, more modestly, to identify scalability problems. Our approach has a "Truman-show" [25] feel: the part of the system whose scalability is being tested processes real data and interacts with the rest of the system as it would in a true large-scale deployment, while the rest of the system uses Tardis to compress data and achieve high degrees of colocation, thereby emulating the behavior of a large number of nodes.

- We present our experience using Exalt, a library that implements Tardis and uses our methodology to identify scalability issues in large-scale storage systems. Using Exalt we found and fixed several such issues in two mature storage systems: HDFS [22] and HBase [2]. All the problems we identified manifest when the scale of the system becomes larger than a typical research cluster. In the case of HDFS, resolving these problems resulted in an order of magnitude improvement of the aggregate system throughput. Our ability to identify these issues was not, for better or worse, due to a prior deep understanding, but rather to the opportunity offered by Exalt to test them at an unprecedented scale.

The rest of the paper is organized as follows. Section 2 discusses the common practices for testing the scalability of large systems. Section 3 introduces the Tardis data representation scheme and Section 4 describes how it can be used to identify scalability problems in large-scale systems. Section 5 reviews the assumptions of Exalt and discusses its applicability in various contexts. Section 6 presents our experience using Exalt to identify performance problems in two mature systems: HDFS and HBase. Section 7 discusses related work and Section 8 concludes the paper.

2 Testing for scalability: common practices

When faced with the challenge of running experiments on a system whose scale vastly exceeds their infrastructure, researchers typically resort to one of two options: they either run the system at the largest scale they can afford and try to extrapolate their results, or they explicitly forgo running certain components of the system, substituting them with stubs that, ideally, maintain the interactions of the original components with the rest of the system, but are simpler and less resource-demanding to run. We discuss both options, and why they are not well-suited for performing scalability tests on large-scale systems.

2.1 Extrapolation

A common approach to estimate the behavior of systems that are too big to test is to run them at a small or medium scale and then to extrapolate, based on those results, how they will behave at a large scale. For example, if the CPU utilization of a bottleneck node is 10% in a 100-node experiment, extrapolation would lead one to estimate that the system will scale to about 1,000 nodes. While attractive for its simplicity, this approach has several drawbacks that make it inaccurate in practice.

First, extrapolation is based on the assumption that resource usage grows linearly with the scale of the system. However, because of design choices and implementation issues, this assumption is frequently violated in practice. For example, HDFS uses an array to maintain a sorted list of files within a directory. Using an array causes insertion to be an $O(N)$ operation, where N is the number of files in the directory. As more files are added to the directory, insertion becomes increasingly expensive: indeed, the cost of adding N files to a directory is $O(N^2)$. Note that a more efficient directory implementation (e.g. a sorted tree map) does not restore linear growth in resource usage, but simply reduces the growth rate to $O(N \cdot \log N)$. In general, once the load on the system is not linear, accurate

extrapolation becomes much harder, especially because, as we have seen, the system's performance may depend on the details of the implementation.

A second, more subtle drawback of extrapolation is that at small scales some important behaviors can easily escape notice. Consider again the above example of a workload of $O(N^2)$ complexity: as long as the value of N is low, the potential scalability bottleneck remains largely inconspicuous. To exacerbate the problem, measuring resource utilization is an inherently noisy process. For example, observing that a Java process uses 100 MB of memory does not, by itself, indicate how much memory is being used by the data structures of that process. Answering that question requires accurate information about the amount of memory used internally by the JVM, the amount of non-garbage-collected memory, etc. The uncertainty added by measurement noise is significantly more prominent at lower scales, where resource utilization is low.

The final drawback, which is closely related to the previous one, is that extrapolation cannot be used to predict behaviors that are only triggered when some resource utilization reaches a certain threshold. For example, HDFS has a blocking disk-scanning procedure that becomes increasingly expensive as the system grows in size. Beyond a certain size, running the procedure causes the corresponding DataNode to start missing heartbeats, which in turn can cause it to be evicted and force all its data to be re-replicated, with serious performance repercussions.

2.2 Using stubs

Another technique for predicting the performance of a system too big to test is to emulate, rather than actually run, some of its components. The emulated components are implemented as stubs, running either locally or remotely. For this approach to be successful, the stubs should be simple to implement and require much more modest resources than the original components they stand in for; at the same time, they should be able to correctly exercise the rest of the system, allowing it to be stress-tested at scale using relatively modest resources.

While attractive in theory, the promises of emulation are often elusive in practice: reproducing accurately the behavior of a non-trivial real system component is hard, and in the process the stub component can end up being almost as complex as the real one, defeating its purpose.

We faced this challenge first-hand when trying to test the scalability of the HDFS NameNode using stub DataNodes. Our goal was to create a large number of stub DataNodes and use them to stress-test the NameNode. Our first attempt did not involve the DataNodes in the protocol at all; to create files and add blocks to them, clients simply invoked `createFile` and `addBlock` at the NameNode. However, the system did not work, since the NameNode expects the DataNodes to confirm the receipt of each block. We therefore modified our clients to notify the stub DataNodes, so they could in turn appropriately notify the NameNode. This did not work, either: the NameNode, we discovered, also expects each DataNode to periodically report the list of blocks it stores on disk. After several frustrating iterations, we eventually came to realize that emulating the correct behavior of DataNodes would have required us to reimplement the full HDFS protocol, including all inter-DataNode communication, local bookkeeping, etc.

3 Compressing data with Tardis

Our approach is based on a simple intuition: for the purposes of testing the scalability of large-scale storage systems, it is typically the size of the data being written that matters, not its actual content. We are then free to *choose* what data clients write during our tests: our work explores the opportunities that this freedom affords.

Specifically, our approach is to design a data format that achieves fast and efficient compression and decompression. As we discuss in Section 3.3, using compressed data lets us colocate multiple nodes on the same machine, which in turn enables running large-scale experiments on a small infrastructure.

Before presenting Tardis, our compression scheme, we set forth the requirements that it must fulfill.

3.1 Compression scheme requirements

The scheme must be lossless. While compression can reduce resource usage and allow node colocation, the ability to recreate the original data is essential. Modern large-scale storage systems typically use a two-layer architecture, where the upper layer uses the lower as a black-box storage [2–4]. What appears like generic data to the lower storage layer may actually be metadata necessary for the correct functioning of the upper layer; it is critical that none of this metadata be lost.

The scheme must achieve a high compression ratio. The motivation for this requirement is straightforward, since the compression ratio directly affects the amount of colocation we can achieve.

The scheme must be computationally efficient. As a counterexample, consider a straw-man scheme in which clients simply write sequences of 0's. This scheme offers obvious opportunities for significant compression; however, if it is possible for the system to interleave client data with metadata, the compression algorithm would need to scan all the input bytes to determine where the sequence of 0's begins and where it ends. The disk and network bottlenecks would have been removed, but at the expense

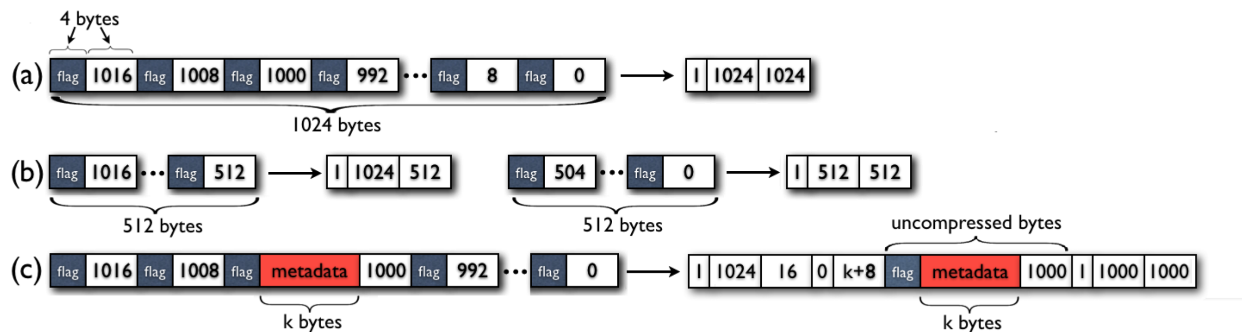

Figure 1: Examples of the Tardis format in compressed and uncompressed form.

of introducing a CPU bottleneck, severely limiting the scalability of this scheme.

Data chunks should be independently compressible. Modern storage systems do not necessarily store data as a single unit, but instead split it into multiple, separately stored chunks, which must be independently compressible. Meeting this requirement is challenging, however, since a client in general has no control over how data is divided into chunks. For example, in HBase the procedure of splitting data into chunks depends on a non-deterministic race between multiple threads.

3.2 Tardis compression

This paper introduces a novel compression scheme, called *Tardis*, that satisfies the above requirements. Tardis consists of a data format and an algorithm for compressing and decompressing the data. Intuitively, Tardis aims to achieve the following two complementary goals. When no metadata is inserted in the middle of the data, the compression algorithm should be able to compress the entire data after scanning only a small fraction of it. Otherwise, the compression algorithm should be able to quickly identify the location of the inserted metadata.

Data format Clients write data as a series of `<flag>` `<marker>` entries, where `<flag>` is a predefined byte sequence that does not appear in the system metadata, and `<marker>` denotes the number of remaining bytes in the data. For example, using a 4-byte flag and 4-byte markers, 1 KB of data would be formatted as:

`<flag>1016<flag>1008...<flag>8<flag>0`

In this example, the first marker denotes that there are 1016 bytes remaining in the sequence, since the (first) flag and the marker itself are 4 bytes each. Of course, the size of flags and markers need not be the same: our prototype uses 8-byte flags and 4-byte markers.

Compressed data format Given a byte sequence in the above format, the compression algorithm would simply need to return its length. However, to enable data chunks to be independently compressible, the algorithm actually returns two numbers: the starting byte of the sequence as well as its length. In the above example (also illustrated in Figure 1a), if the entire 1 KB of data were being compressed, the result would be the pair (1024,1024). If, however, the data were split into two chunks of 512 bytes each (Figure 1b), the first chunk would be compressed as (1024,512) and the second as (512,512).

As we discussed above, in modern storage systems data and metadata are frequently stored together. Figure 1c shows an example where metadata is inserted in the middle of a Tardis sequence. In this case, the metadata splits the original sequence into two subsequences, of length 20 and 1004, respectively. Ideally, we would like to compress each of these sequences separately, leaving the metadata uncompressed. However, since in this case the metadata is inserted in the middle of a flag-marker pair, we simply leave these 8 bytes—the flag and the corresponding marker—uncompressed.[1] This shortens the first subsequence to a length of 16 and the second subsequence to 1000. Note that even if the metadata were not aligned with the flags and markers, the result would be the same: only the flag-marker pair that is split by the metadata is left uncompressed and the rest of the data is compressed as two separate subsequences.

To distinguish between compressed and uncompressed data during decompression, an uncompressed sequence is preceded by a 0 and a 4-byte integer denoting its length, while a compressed sequence is preceded by a 1.

Compression Figure 2 shows the pseudocode for the Tardis compression algorithm. The main function, `TardisCompress`, iteratively calls the `FindSubsequence` function until all input data has been consumed. When `FindSubsequence` returns a new subsequence (line 7), the main function appends the appropriate bytes to the compressed data buffer. We detect the presence of metadata between two subsequences by checking whether the starting position of the new subsequence (*pos*) is after the end of the previous subsequence (*index*). If so, we append a 0 to denote the beginning of an uncompressed sequence,

[1] It is actually possible to include the flag in the compressed sequence, but we omit this optimization for simplicity of presentation.

```
1   #define unit_size = flag_size + marker_size

3   (compressed_data) TardisCompress(data)
4      result = empty buffer
5      index = 0
6      while index < data.len
7         (pos,marker,len)=FindSubsequence(data,index)
8         if pos == -1
9            result.AppendMeta(data,index,data.len-index)
10           return result
11        else
12           if pos > index
13                result.AppendMeta(data,index,pos-index)
14           result.AppendTardis(data,pos,marker,len)
15           index = pos+len
16     return result

18  (pos,marker,length) FindSubsequence(data,startIndex)
19     (pos,marker) = ScanForFlag(data,startIndex)
20     if pos == -1
21        return (-1,-1,-1)
22     lastMarker = data.len-(unit_size+(data.len-
            position)%unit_size)
23     target = min(pos+marker,lastMarker)
24     marker2 = BinarySearch in data from pos to target
25        for the rightmost flag-marker pair such that:
26        (pos2,marker2) = ScanForFlag(data,target) and
27        pos2 != -1 and pos2-pos == marker-marker2
28        return (pos, marker, marker-marker2+unit_size)
29     if no such marker2 is found
30        return (-1,-1,-1)

32  (position, marker) ScanForFlag(data, startIndex)
33     index = linearly search data for (flag,marker)
            starting at startIndex
34     if index >= 0
35        return (index, marker)
36     else
37        return (-1, -1)
```

Figure 2: Pseudocode for Tardis compression.

followed by the length of the metadata, and finally by the metadata itself, uncompressed (*AppendMeta*, line 13).

It is then time to add the new subsequence. To denote that what follows is compressed, we append a 1 before the compressed form of the Tardis subsequence (which, recall, consists of the starting point and length of the subsequence) (*AppendTardis*, line 14).

Function `FindSubsequence` is the core of the algorithm: its task is to identify a Tardis subsequence. Two factors complicate this task: the sequence may have been split into multiple chunks and metadata may have been inserted somewhere in the sequence. Given a starting index in the data, `FindSubsequence` first scans the data to find the first flag, indicating the start of a Tardis sequence, and reads the corresponding marker (line 19). Then, it checks whether some metadata has been added in the middle of this sequence. The check is simple: if no metadata is inserted between two markers with values A and B, then these markers should be placed $B-A$ bytes apart. The purpose of lines 22-23 is to determine which marker should serve as marker B. If the original sequence is not split across chunks, then B is marker 0, which should be m bytes after the first marker, where m is the value of the first marker. Otherwise, B is set to the last marker of the current chunk. If the difference between the values of markers B and A is indeed equal to the byte distance between the markers, the algorithm has found an uninterrupted Tardis subsequence. If that is not the case, the algorithm performs a binary search to find the rightmost flag-marker pair that satisfies the above condition, leveraging the fact that the values of the markers form a sorted sequence (lines 24-30).

In practice, the common case is very simple: as long as there is no metadata inserted in the byte sequence, the compression algorithm needs only to check the first and last number of the sequence. This allows Tardis to compress data much faster than off-the-shelf compression algorithms. For example, when compressing data chunks of 1 MB, Tardis is about 33,000 times faster than Gzip [11] and 2,300 times faster than the straw man compression scheme where client data consists only of 0's and the compression algorithm simply scans the data and compresses sequences of 0's into an integer denoting their length. Of course, the comparison to GZip is not apples-to-apples, since Gzip is a generic compression algorithm; what it does show, however, is that being able to choose the data format drastically reduces the CPU overhead of our approach.

Decompression The decompression algorithm is straightforward. Given a compressed sequence, it iterates through each sequence, whether compressed (preceded by a 1) or uncompressed (preceded by a 0 and the length of the sequence). Uncompressed sequences are copied without modification, while compressed sequences are expanded to their uncompressed form.

Choosing the flag To prevent portions of metadata from being accidentally compressed, the flag sequence should never appear in the metadata. If it did and, by unlucky coincidence, the length value following the fake flag pointed to another flag followed by a 0, all that sequence of bytes would be compressed. Although we could altogether eliminate this danger,[2] it seems unnecessary: Exalt is not intended for production use, and an accidental compression would simply require us to rerun the affected experiment. With a sufficiently large flag, the odds of a false positive can be driven arbitrarily low: our pragmatic approach was to choose as flag an 8-byte random sequence and take our chances. Our experiments have yet to produce a false positive.

3.3 Using compression to enable large-scale tests

Since we are attempting to run a large number of nodes on a much smaller number of machines, we will nec-

[2] It would suffice to escape the flag sequence in the metadata. However, this would require intrusive modifications to the server code, as all metadata insertions would need to be aware of the escaping logic.

essarily have to colocate multiple nodes on the same machine. However, such colocation will cause significant contention on the physical resources of the machine. Specifically, the disk- and memory capacity, and the disk- and network bandwidth available to each machine are typically enough to support only a single node, making straightforward colocation infeasible.

Data compression can help here: storing compressed data on disk decreases the disk capacity and bandwidth requirements of each node, as well as memory capacity and network bandwidth. Of course, data compression is not without cost; in this case, the cost is CPU utilization.

This tradeoff, however, is very attractive for storage systems, where CPU cycles are plentiful and bandwidth and storage capacity are typically the system's bottleneck. It also opens the door to emulating the behavior of storage systems too big to test using HPC computation clusters: indeed, as we will see in Section 6, our analysis of the scalability of HDFS/HBase has been performed by running Exalt on the Stampede high performance cluster at the Texas Advanced Computing Center (TACC) [23].

If data compression is used without colocation, it results in a system that is "compressed" in time, rather than space, since each write will take less time to complete. Running the system at an accelerated pace offers the potential of identifying bugs or performance problems much faster: Section 6.1.4 discusses a case where time compression allowed us to identify a problematic behavior about 100 times faster than a real deployment.

3.4 Implementation

Our implementation of Exalt performs data compression for three key resources: disk, network, and memory. Our goal is to be minimally intrusive. While in-memory compression does require minor modifications to the source code of the storage system being tested, we achieve fully transparent disk and network compression by using byte code instrumentation (BCI) to modify the relevant Java library classes (Socket, SocketInputStream, SocketOutputStream, SocketChannel for network compression; File, FileInputStream, FileOutputStream, RandomAccessFile, and FileChannel for on-disk compression).

File compression is more challenging than network compression because the file interface allows a user to partially update existing data. When that data is already compressed, updating it in place is not straightforward. A naive solution would be to decompress the existing data, update it, and compress it again. However, if the old and the newly compressed data have different sizes, all following data chunks would have to be moved. To address this problem, similarly to the Log-Structured File System (LFS) [20], we transform in-place update operations into append operations. This allows us to efficiently process in-place updates, with only a small bookkeeping overhead to keep track of the latest version of each range of bytes.

Memory compression In-memory data structures do not use a well-defined interface, such as the File or Socket abstraction used by the disk and network. As a result, transparently modifying these data structures to compress and decompress data at the application layer is very hard.[3] Instead, when the in-memory data needs to be compressed, we manually modify the source code of the system. Fortunately, this process is quite simple. One need only identify the data structures that hold the client data. When data is stored in the data structure, it is compressed; when data is retrieved from the data structure, it is decompressed. For example, compressing the in-memory key-value store of HBase required adding 71 lines of code across 4 files.

4 Finding scalability bottlenecks

Data compression gives us the ability to colocate multiple nodes on a single physical machine: in this section, we discuss how we can selectively use this ability to draw meaningful conclusions about the scalability of a large-scale storage system. We will view the system as a collection of *real* and *emulated* nodes. A real node runs the system's actual code and handles unmodified data. An emulated node still runs the system's actual code, but, as needed to support colocation, may (i) store compressed data on its disk, (ii) send compressed data over the network, when it communicates with other emulated nodes, and (iii) store compressed data in memory.

4.1 Exalt methodology

We use this combination of real and emulated nodes as a microscope of sorts that we can focus on a part of the system to identify performance bottlenecks. To ensure that the part of the system "under inspection" behaves exactly as it would in a real large-scale deployment, we leave the corresponding nodes real, while using emulated nodes for the rest of the system. This approach works particularly well at identifying performance issues at centralized components that can become a bottleneck as the scale of the system increases (e.g. HDFS NameNode, HBase Master). Section 6 discusses our experience using this technique to find scalability problems in real systems.

A downside of this methodology is that it may not discover scalability problems that arise at the nodes that are being emulated. To address this issue, after having stress-tested the part of the system under inspection by using the maximum amount of colocation for emulated nodes, we perform a new set of experiments where a small

[3] Transparent compression of in-memory data could be potentially implemented at the kernel level, but it would sacrifice portability.

subset of formerly emulated nodes are also run as real, while the rest is kept emulated. This hybrid configuration makes it possible to identify scalability problems also at nodes that are not under inspection, while maintaining a high degree of colocation, but it is not a panacea: for example, it is still unable to detect performance issues that only manifest when a large number of nodes that are not under inspection perform some collective action (e.g. system-wide recovery).

5 Limitations and applicability

Exalt relies on a number of assumptions to provide high-degrees of node colocation. This section reviews these assumptions and discusses which of them can be weakened to widen the applicability of our approach.

Exalt is primarily designed to evaluate I/O-intensive applications like distributed file systems storing large files [7, 22] or key-value stores with relatively large values [3, 17]. Applications that are not I/O-intensive or store small values can not benefit significantly from Tardis, as they gain little by compressing data. In Section 6.2 we explore in more detail how the size of the value in a key-value store affects the colocation ratio of Exalt.

Our current implementation of Exalt makes two additional assumptions: first, that the target application does not modify the data written by the clients, although it can split the data and insert metadata; and second, that experiments are not sensitive to the contents of the data, so that clients can operate with synthetic data.

While these assumptions hold for the systems we have so far applied Exalt to, they are not fundamentally required for Exalt to be applicable. We consider below some popular techniques that violate these assumptions and discuss how our implementation of Exalt can be modified to work in conjunction with them.

Encryption and erasure coding Both techniques involve encoding data into a different format, violating the assumption that client data is immutable. To handle these cases, Exalt would compress the data using Tardis *before* encoding it, and then add *filler bytes* as necessary to match the length of the (encoded) original data. Filler bytes would use the same format as Tardis (making them highly compressible), but with a different flag sequence (so that they can be distinguished from real data). When reading the data, Exalt would remove the filler bytes before performing decryption and then decompress the Tardis sequence to obtain the client data.

Deduplication Deduplication compares the contents of different data units (files, chunks, etc.) to eliminate duplicates and, by making execution dependent on the actual data, violates our second assumption. Indeed, deduplication schemes that directly compare the units' data are incompatible with Exalt. However, Exalt can still be applied to deduplication approaches that only compare hashes of data units. Exalt would first compute the hash of the client data and then replace the client data with data formatted using an extended version of Tardis, which inserts the hash of the data unit between the flag and the marker. The deduplication module could then use this hash directly to identify duplicate data units.

Compression If the system being tested already uses compression, it is in general not possible to use synthetic data at the clients, since the compression ratio depends on the actual data. If, however, compression is performed only at the client side, Exalt could apply a technique similar to the one used to handle encryption and erasure coding: the client would first compress the real data to determine its compressed size, then create synthetic (Tardis) data, compress it using Tardis' compression and finally append the right amount of filler bytes to match the length of the (compressed) real data.

Data sensitive applications Many applications use SQL-like languages for their queries. The execution of these queries depends on the data, since SQL predicates can be expressed as a function of the data. The rest of the data, which does not affect the processing of the queries, can be synthetic. The efficiency of Exalt in these cases depends on the ratio of sensitive to non-sensitive data.

6 Case studies

Exalt allows us to evaluate storage systems at an unprecedented scale. This section presents our experience applying Exalt to evaluate two real-world storage systems: the Hadoop Distributed File System (HDFS) and the HBase key-value store [2, 22]. We chose these systems for several reasons. First, not only they are popular in their own right, especially among researchers, but their architecture is representative of the majority of existing large-scale storage systems. Second, both systems are open-source, which allows us to perform code modifications where necessary (i.e. for in-memory compression). Finally, both systems have a large development community that has produced a mature and stable codebase. Despite the maturity of the code, we identified several performance issues that arise as the scale of the system increases. Our ability to diagnose these issues was not due to a prior deeper understanding of these systems, but simply to the ability to evaluate them at an unprecedented scale.

In our evaluation, we run HDFS and HBase at a scale about 100 times larger than the size of the infrastructure available to us. For example, one of our experiments uses 96 machines to run an HDFS cluster with 9600 DataNodes. Our experiments identify a number of performance problems that arise at such large scales. Some of these problems pertain to low-level implementation details, while others are due to high-level design choices.

For example, we find that storing many files on an HDFS directory causes file creations to that directory to become increasingly slow; and that keeping less than $\frac{3}{4}$ of the region data in the memory of an HBase region server causes its performance to degrade precipitously. Using Exalt, we were able to identify and fix many of these problems, improving as a result the aggregate HDFS throughput by an order of magnitude.

Unfortunately, we have not yet been able to validate our results by running the actual systems at a large scale: after all, it is the very reason that we do not have access to such plentiful resources that has motivated our work in the first place. The largest validation we have performed involved running HDFS/HBase on 1,500 nodes of the Stampede cluster at the Texas Advanced Computing Center [23]: while our results confirm the prediction of Exalt for that configuration, the scale of the system is still too small to exhibit even the first of the scalability issues identifies by Exalt. Our current confidence in Exalt's effectiveness stems from two sources. First, for each problem that Exalt has identified, we have traced the cause of the problem in the source code, fixed it, and run the modified system to confirm that its performance has been improved. Second, some of our findings have been confirmed by engineers at Facebook, among the few who have access to a large-scale deployment of HDFS [6].

Most of our experiments were performed on the Stampede cluster at TACC, whose machines have 16 cores, 32 GB of memory, but only 80 GB of local disk storage. Since our access to TACC was limited, we ran some of our experiments on three local machines with 16 cores, 64 GB memory and 10 1 TB disks each. These machines were used to test the capacity limitations of individual storage nodes.

6.1 Case study: HDFS

HDFS [22] is an open source implementation of the Google File System (GFS) [7]. Each HDFS cluster contains a single NameNode that stores the file system namespace information and several DataNodes that store the file contents. Each file is split into multiple blocks and each block is stored on three DataNodes. When a client creates a file or adds a block to an existing file, it first contacts the NameNode, which responds with a list of the DataNodes that will store the new block. The client can then directly write the block contents to these DataNodes.

We mainly focus on write workloads since they are more likely to cause scalability problems. Unless otherwise specified, in our experiments each client creates a file in its own directory, writes 192MB of data to it (as suggested by the HDFS developers in their white paper on how to test HDFS' scalability [21]), closes the file, and then starts a new file. This workload achieves the

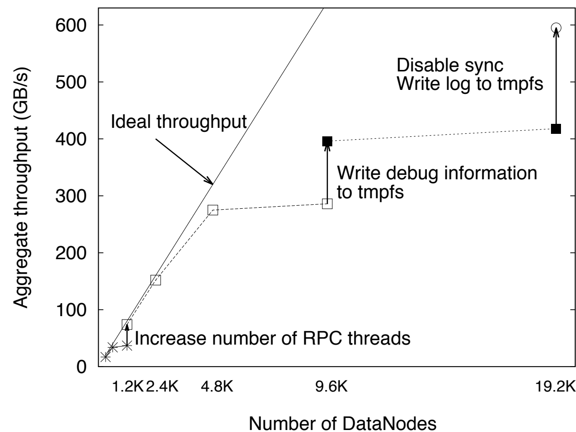

Figure 3: HDFS throughput scalability.

highest scalability among all workloads that we tried; Section 6.1.3 describes the performance problems caused by other workloads. We use a block size of 128 MB and the default 3-way replication (again, as suggested in [21]). Unless otherwise specified, we run DataNodes and clients in emulated mode while the NameNode runs in real mode.

For the above workload, Tardis achieves a compression ratio of over 500, but in practice the degree of colocation is limited by CPU utilization: we colocate 100 DataNodes on one machine and achieve an effective write bandwidth of 10 GB/sec on a disk with 100 MB/sec physical bandwidth. For experiments with modest storage capacity requirements, we can increase the write bandwidth to 20 GB/sec by writing to tmpfs, an in-memory file system. Our largest configuration experiment uses 192 server machines, to emulate an HDFS cluster with 19,200 DataNodes.

6.1.1 HDFS throughput scalability

In some respect, the result of our experiments to test the scalability of HDFS is not surprising: the bottleneck of the system is the centralized NameNode. What is perhaps surprising is that, thanks to Exalt, we were able to increase the system throughput by an order of magnitude without changing the architecture of the system.

Figure 3 reports the results of our experiments. On the x-axis we increase the number of DataNodes and on the y-axis we plot the aggregate throughput of the system, as observed by the clients. The vertical arrows represent the process of fixing an issue that was limiting the system throughput. When an issue is fixed, we rerun the experiment for the same number of DataNodes, to verify that the system indeed achieves a higher throughput. For reference, we also plot a straight line that shows the ideal throughput achievable by a perfectly scalable system that leverages the full bandwidth of all disks (100 MB/s).

Our first experiment shows that the original HDFS sys-

Memory size	1GB	2GB	4GB	8GB
HDFS Capacity	1.15PB	2.35PB	4.76PB	9.49PB

Figure 4: HDFS space scalability as a function of NameNode memory size.

tem quickly saturates at around 37 GB/s. We discovered through profiling that the default number of RPC threads at the NameNode was limiting the achievable throughput; increasing the number of RPC threads from 10 to 256 allows the NameNode to achieve much higher throughput.

After fixing the first issue, the system saturates at around 286 GB/s. Further profiling showed that the I/O accesses at the NameNode were becoming the system bottleneck. More specifically, the NameNode debug information was being stored on the same disk as its log file. This prevented both files from sequentially accessing the disk, thereby introducing a large number of `seek` calls that reduced performance. Our solution was to write the debug information to tmpfs instead, thereby making sure that the NameNode log file was sequentially accessing the disk. Alternatively, one could store the debug information on another disk, if one were available.

Applying the second fix increases the system throughput to 418 GB/s, at which point the system again becomes saturated. This finding is consistent with the scalability assessment of the HDFS developers that "assuming each client has a write throughput of 40 MB/s, the system can support no more than 10,000 clients", which corresponds to an aggregate throughput of 400 GB/s. While this assessment was obtained using extrapolation, we consider it reasonably accurate since it is based on a large deployment of 4,000 nodes.

Since we suspected disk I/O to be the system bottleneck at this point, we performed a final experiment in which disk `sync` is disabled and the NameNode writes all logs to tmpfs. The purpose of this experiment is to project the scalability of the system in the presence of a fast storage medium (e.g. NVRAM,SSD). In this configuration, the system throughput increases by a further 50%, to a maximum throughput of 595 GB/s.

Of course, we do not claim that Exalt's throughput predictions are perfectly accurate; on the contrary, we acknowledge the limitations of running a system whose resources are partially emulated. Nonetheless, the benefits of Exalt are clear: it allowed us to test the system's real code and identify and resolve performance issues at a scale that would have otherwise remained the sole province of a few large companies.

6.1.2 HDFS capacity

The capacity of an HDFS cluster is limited by the amount of memory available to the NameNode. In this

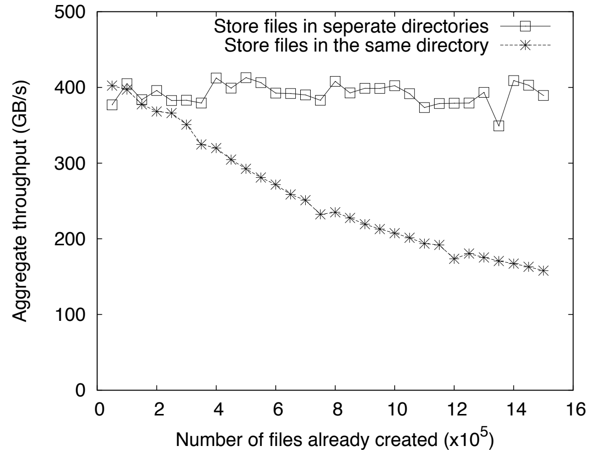

Figure 5: HDFS throughput degradation as the size of directories increases.

experiment, we try to measure how much memory the NameNode needs per 1 PB of HDFS storage space. Figure 4 shows that the capacity of HDFS grows linearly with the amount of memory at the NameNode. In particular, 1 GB of NameNode memory can support about 1.2 PB of raw HDFS space (400 TB of data, since blocks are 3-way replicated). This result is close to the estimation of HDFS developers: "1 GB of metadata $\approx$ 1 PB of physical storage" [21].

Using Exalt allows us to perform this experiment using only 16 TB of disk storage, while a real deployment would require a total of 10 PB of disk storage.

6.1.3 Performance degradation in HDFS

The above experiments use a workload that provides high scalability. Other workloads are not as accommodating. We evaluate two such workloads that can drastically degrade the HDFS performance.

In the first workload, all clients create files in the same directory. As shown in Figure 5, the aggregate system throughput steadily decreases as more files are created. Further profiling allowed us to identify the cause of this behavior in the source code: the NameNode uses an ArrayList data structure to maintain an alphabetically sorted list of the files inside a directory. Adding an element to a sorted array is an O(N) operation, since it requires a suffix of the sorted array to be shifted by one position. Therefore, the bigger the directory, the longer it takes to add a file to it. As a double check, we verified that, if we limit the number of files written to each directory, creating more files does not cause a performance degradation.

In the second workload, one client creates a file and keeps appending data to it. As shown in Figure 6, once the file grows sufficiently large, the aggregate system throughput decreases steadily. Note that in this experiment there are only a few clients and the system is not fully sat-

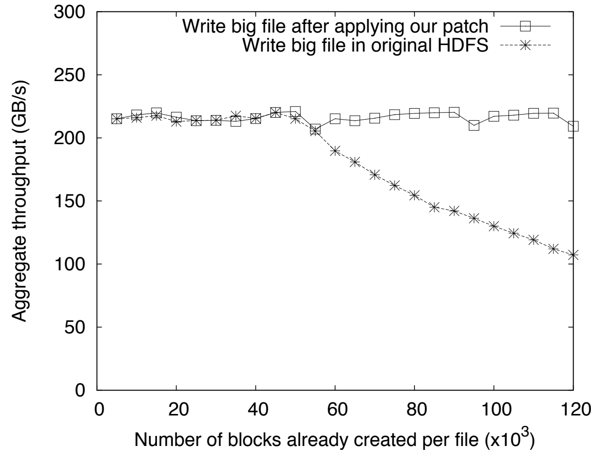

Figure 6: HDFS throughput degradation as the size of files increases.

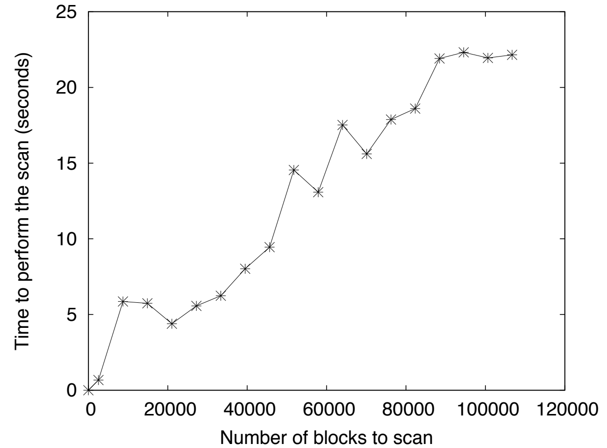

Figure 7: Time of the block-scan procedure on a DataNode, as the number of blocks increases.

urated, which accounts for the fact that the aggregate system throughput is lower than in the previous experiment. Profiling led us to the cause of the problem: before the NameNode creates a new block for a file, it needs to calculate the file's length. It does this by scanning all existing blocks and computing the sum of the lengths of all blocks. This, too, is an O(N) operation. We fixed this problem by adding a length field to each file and updating the field when a block is added or updated. As Figure 6 shows, after applying our fix the system throughput no longer decreases as the files grow in size.

As before, Exalt allows us to identify these performance issues without requiring access to a large amount of disk storage. Running this experiment in a real deployment would require 900 TB of disk storage; with Exalt, we only need 1.5 TB.

6.1.4 DataNode scalability

As disk capacities increase every year, and most HDFS deployments use multiple disks per DataNode, it is important for the DataNode's performance to not decrease as more storage capacity is added to it. While running HDFS in hybrid mode—keeping some DataNodes real— we observed uncommonly high latencies for some requests. Our profiling indicated that the source of the problem was a disk scan that the DataNode periodically performs on all its blocks. Figure 7 shows that the time a real node takes to perform this scan increases linearly with the number of blocks stored on the disk. Unfortunately, this scan is a blocking operation, preventing write requests and heartbeats from being sent or received. As the duration of this scan becomes longer, it can have serious performance consequences, including timeouts at the clients or even missed heartbeats, which would cause unnecessary re-replication of the DataNode's data. This issue is confirmed by Facebook engineers; to address it, they modified HDFS to allow the block scan to be performed in parallel with heartbeats and write requests [6].

While reproducing this problem is easy, triggering it in a real deployment would require 8 TB of disk storage on a single DataNode; using Exalt, we triggered this problem using an 80 GB disk. After identifying the problem, we reproduced it on a real DataNode with 8 TB of disk storage (Figure 7).

Note that although it could be triggered with only a few machines, this problem would be hard to identify and tedious to reproduce during debugging, since it would take at least a few hours for the latency increase to be observable. Exalt's time compression helps in this case. If emulated nodes have exclusive access to a machine's resources, the system works at an accelerated speed: in this example, the problem would manifest itself in a matter of minutes.

6.2 HBase

HBase [2] is a distributed key-value store built upon HDFS. The basic data unit of HBase is a region, which corresponds to a continuous key range in a table. An HBase cluster includes a Master, responsible for assigning regions to different region servers. Client requests to a specific region are directed to the corresponding region server. The region server processes write requests by logging them to HDFS while also keeping them in a memory buffer called *memcache*. When the size of the memcache exceeds a threshold, the region server writes the whole memcache into a checkpoint file on HDFS, so that it can garbage collect the previous log files. A checkpoint is also taken if the total memory usage across all regions exceeds some limit; in this case, a region server checkpoints the region with the largest memcache. When necessary to

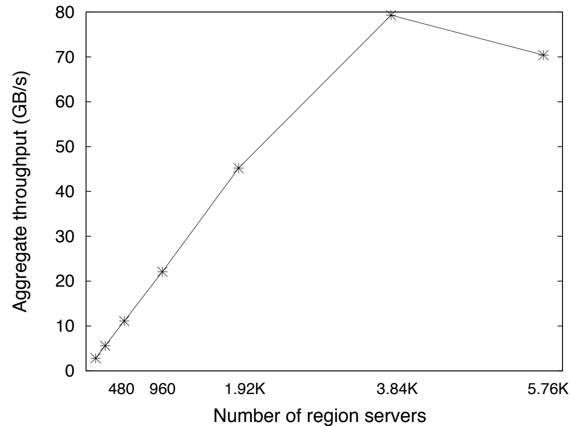

Figure 8: HBase throughput scalability.

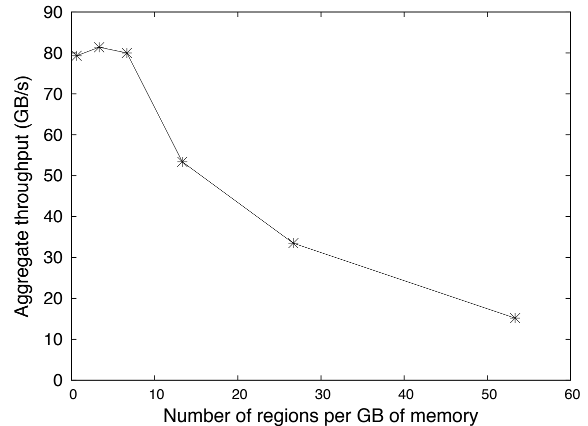

Figure 9: HBase aggregate throughput as the number of regions per GB of memory changes.

free up space, the region server performs *compaction* to merge several checkpoints. In essence, a region server transforms the random access patterns of a key-value store into the append-only interface of HDFS. When a region grows large, HBase splits that region into two for load balancing; conversely, if two adjacent regions are too small, they are merged into one. Apart from the Master and the region servers, an HBase cluster incorporates a ZooKeeper ensemble that performs lease management.

We evaluate HBase using a simple workload that can achieve a high throughput: we create enough regions so that each region server stores about 10 regions. We start multiple clients that randomly write key-value pairs to those regions. The key size is 4 bytes and the value size is 1 MB. To measure the maximum achievable throughput, we disable split, merge, and compaction operations—to ensure that split and merge operations do not occur, we limit the number of key-value pairs written to a region. We plan to study the effects of split, merge, and compaction in the future.

In our experiments, we keep the HBase Master, HDFS NameNode and ZooKeeper cluster real, while all DataNodes and region servers are emulated. In each experiment we assign 500 MB of physical memory to region servers. However, we perform in-memory compression, which effectively increases each region server's memory to 16 GB.

Figure 8 demonstrates the throughput scalability of HBase as the number of available region servers increases. Note that the raw throughput of HBase is much lower than that of HDFS (see Figure 3). This is due to two reasons: first, HBase needs to write data twice to HDFS—once for logging and once for checkpointing. Second, region servers are relatively more CPU-intensive than DataNodes and therefore can not benefit as much from colocating multiple nodes on the same machine.

HBase can achieve a maximum write throughput of about 80 GB/s. Considering that HBase writes data twice, this translates to a 160 GB/s throughput at the HDFS layer, which is about 40% of the maximum throughput achievable by HDFS. Our profiling shows that the `sync` calls to disk at the HDFS NameNode are still the bottleneck of the system. The reason for this 60% performance loss is that the region servers perform many additional directory operations, other than simply creating and closing files. For example, when a log file is garbage-collected, the region server first moves it to an "old log" directory as a backup and only deletes it after some time has elapsed.

In Figure 8 each region server has 16 GB of memory and holds 10 regions: since the default maximum size of a region is 200MB, all data can be cached in memory. Our next experiment evaluates how the performance of HBase is affected when we decrease the memory size per region. As shown in Figure 9, HBase throughput drops significantly when the number of regions per GB of memory exceeds 7, which translates to about 150 MB of memory per region. In other words, in order for HBase to work efficiently in a large-scale deployment, each region server must be equipped with a considerably large amount of memory: enough to hold at least $\frac{3}{4}$ of its on-disk data. The reason for this performance drop is that region servers flush their regions to HDFS files when their memory usage exceeds a certain threshold. If the number of regions per GB of memory is high, this will create a large number of small files on HDFS, which stresses the HDFS NameNode. Resolving this problem requires a significant redesign of HBase, which is beyond the scope of this paper. Note that this performance drop is only observed at large scales, since small deployments can not generate enough load to saturate the HDFS NameNode.

Our last experiment explores the effect of writing small values on the colocation ratio achievable in Exalt (Figure 10). Not surprisingly, Exalt achieves high colocation ratios when the value sizes are large (around 500 KB),

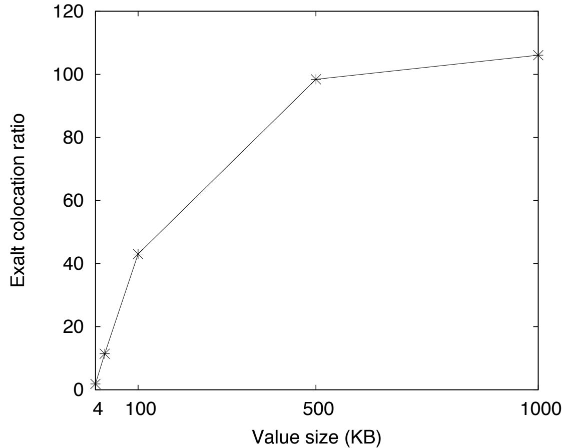

Figure 10: Colocation ratio of Exalt.

but does not fare equally well for small values. It is worth noting that the achievable colocation ratio for a given workload is not infinite; eventually CPU utilization becomes the bottleneck of the system. For HBase, this happens at a colocation ratio of about 110.

7 Related work

As we mentioned earlier, two common approaches to evaluating the scalability of large storage systems are using extrapolation and stub components. For example, extrapolation is used, among others, in RAMCloud [19], Spanner [12], and Salus [27], while the stub approach is used in HDFS [21, 22]. Section 2 discusses these approaches in detail, so we do not discuss them further here.

Several tools have been proposed to address the gap between the size of the experiments that researchers would like to run and the resources available to them.

In DieCast [9] this experimental gap is addressed using time dilation [10]. DieCast runs multiple processes inside virtual machines on a single host and slows down each process by a constant factor. It compensates for this slow-down by multiplying the measured throughput by the same factor. DieCast can achieve some degree of colocation when CPU utilization is the bottleneck, but does nothing to reduce the large amount of disk space necessary to evaluate large-scale storage systems.

The system that comes closer to addressing the experimental gap for storage systems is David [1]. David leverages the observation that to evaluate a local file system it is not necessary to store the actual data. Thus, David only stores the file system's metadata: the data is simply discarded. This technique allows David to evaluate local file systems of much larger size than that of the local disk on which they are run. Unfortunately, this approach cannot be easily applied to distributed storage services. For example, when users write a key-value pair to HBase, the region server adds a timestamp and a region identifier to the write request and stores this metadata, together with the users' data, on the local file system of an HDFS DataNode. Since data and metadata look indistinguishable to the HDFS layer, David would discard metadata critical for the correct operation of the system.

Memulator [8] emulates nonexistent storage components by storing data in memory and accurately predicting how long each operation takes. Its purpose is to test the behavior of the system on devices that the researchers do not have access to. Unlike Exalt, it does not save any resource usage, which makes it not applicable to our goal.

Finally, simulation is a technique used by several systems to evaluate the performance of large-scale deployments. The approaches vary from disk simulation [24], network simulation [15, 18], to simulation of large-scale P2P systems [26]. A well-known drawback of simulation is that its results are only as good as its model of how the system works. Unfortunately, as systems grow in complexity, coming up with a model that accurately captures all their features becomes prohibitively hard.

There exist several compression algorithms [5, 14, 16, 28, 29] one may consider using in our context. However, all these algorithms are designed to be general-purpose and as such they need to scan all the input bytes. Tardis, on the other hand, owes its efficiency largely to the fact that it does not have to scan most of the input bytes.

8 Conclusion

Exalt is a library that gives back to researchers the ability to evaluate the scalability of large storage systems. Exalt is based on the Tardis compression scheme, which leverages a specific data format to achieve efficient compression and high degrees of colocation, which in turn allows researchers to perform large-scale experiments on as few as one hundred machines. We have used Exalt to identify several performance problems in HDFS and HBase. Fixing these problems allowed the system to significantly increase its maximum achievable throughput. We plan to use Exalt to evaluate the performance of more large-scale systems (e.g. Cassandra [13]).

Finally, we plan to further explore the relationship between space and time compression to quickly diagnose problems that might otherwise require several days or weeks of testing.

Acknowledgements

We thank Mark Silberstein for his insight during early discussions of this work, our shepherd Miguel Castro and the anonymous reviewers for their helpful comments, and TACC for providing access to the Stampede cluster. This work was supported in part by NSF grant CiC-FRCC-1048269 and by a Google Graduate Fellowship.

References

[1] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *FAST*, 2011.

[2] Apache HBASE. `http://hbase.apache.org/`.

[3] B. Calder et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[5] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 1984.

[6] Private communication with Facebook engineers Siying Dong and Liyin Tang.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43. ACM Press, 2003.

[8] J. L. Griffin, J. S. andSteven W. Schlosser, J. S. Bucy, and G. R. GangerNitin. Timing-accurate Storage Emulation. In *FAST*, 2002.

[9] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI*, 2008.

[10] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *NSDI*, 2006.

[11] Gzip. http://www.gzip.org/.

[12] J. Corbett et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[13] A. Lakshman and P. Malik. Cassandra – A decentralized structured storage system. In *LADIS*, 2009.

[14] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, Nov. 1990.

[15] Network Emulation with the NS Simulator. http://www.isi.edu/nsnam/ns/ns-emulation.html.

[16] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, July 1997.

[17] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *OSDI*, 2012.

[18] NS-2. `http://www.isi.edu/nsnam/ns/`.

[19] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[20] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, Feb. 1992.

[21] K. Shvachko. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[23] Texas Advanced Computing Cener (TACC). https://www.tacc.utexas.edu/.

[24] The DiskSim Simulation Environment. http://www.pdl.cmu.edu/DiskSim/.

[25] The Truman Show. http://www.imdb.com/title/tt0120382/.

[26] K. Wang. Exploring the Design Tradeoffs for Exascale System Services through Simulation. http://datasys.cs.iit.edu/ kewang/documents/presentation_1.pptx.

[27] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *NSDI*, 2013.

[28] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[29] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sept. 1978.