Compaction management in distributed key-value datastores

Muhammad Yousuf Ahmad
McGill University
muhammad.ahmad2@mail.mcgill.ca

Bettina Kemme
McGill University
kemme@cs.mcgill.ca

ABSTRACT

Compactions are a vital maintenance mechanism used by datastores based on the log-structured merge-tree to counter the continuous buildup of data files under update-intensive workloads. While compactions help keep read latencies in check over the long run, this comes at the cost of significantly degraded read performance over the course of the compaction itself. In this paper, we offer an in-depth analysis of compaction-related performance overheads and propose techniques for their mitigation. We offload large, expensive compactions to a dedicated *compaction server* to allow the datastore server to better utilize its resources towards serving the actual workload. Moreover, since the newly compacted data is already cached in the compaction server's main memory, we fetch this data over the network directly into the datastore server's local cache, thereby avoiding the performance penalty of reading it back from the filesystem. In fact, pre-fetching the compacted data from the remote cache *prior* to switching the workload over to it can eliminate local cache misses altogether. Therefore, we implement a smarter warmup algorithm that ensures that all incoming read requests are served from the datastore server's local cache even as it is warming up. We have integrated our solution into HBase, and using the YCSB and TPC-C benchmarks, we show that our approach significantly mitigates compaction-related performance problems. We also demonstrate the scalability of our solution by distributing compactions across multiple compaction servers.

1. INTRODUCTION

A number of prominent distributed key-value datastores, including Bigtable [3], Cassandra [12], HBase[1], and Riak[2], can trace their roots back to the log-structured merge-tree (LSMT) [13] – a data structure that supports high update throughputs along with low-latency random reads. Thus, these datastores are well-suited for online transaction processing (OLTP) applications that have demanding workloads. In order to handle a high rate of incoming updates, the datastore does not perform updates in place but creates new values for the updated keys and initially buffers them in main memory, from where they are regularly flushed, in sorted batches, to read-only files on stable storage. As a result, reading even a single key could potentially require traversing multiple files to find the correct value of a key. Hence, a continuous build-up of these immutable files can cause a gradual degradation in read performance that gets increasingly worse over time. In order to curb this behavior, the datastore runs special maintenance operations – commonly referred to as *compactions* – on a regular basis. A compaction merge-sorts multiple files together, consolidating their contents into a single file. In the process, individual values of the same key, potentially spread across multiple files, are merged together, and any expired or deleted values are discarded. Thus, over the long run, compactions help maintain read latency at an acceptable level by containing the gradual build-up of immutable files in the system. However, this comes at the cost of significant latency peaks during the execution of compactions, as they compete with the actual workload for CPU, memory, and I/O resources.

Since compactions are an essential part of any LSMT-based datastore, we would like to be able to exercise a greater degree of control over their execution in order to mitigate any undesirable impacts on the performance of the regular workload. Datastore administrators, based on their experience and understanding of application workloads, manage these performance overheads by carefully tuning the size and schedule of compaction events [1]. For example, a straightforward mitigation strategy could be to postpone major compactions to off-peak hours. Recent proposals and prototypes of smarter compaction algorithms in Cassandra and HBase (e.g., *leveled*, *striped*) attempt to make the compaction process itself more efficient, generally by avoiding repetitive re-compactions of older data as much as possible. However, there is currently a dearth of literature pertaining to our understanding of how and when exactly compactions impact the performance of the regular workload.

To this end, as our first contribution, this paper presents an in-depth experimental analysis of these overheads. We hope that this helps data platform designers and application developers to better understand and evaluate these overheads with respect to resource provisioning and framing performance-based service-level agreements.

[1]http://hbase.apache.org/
[2]http://basho.com/riak/

Proceedings of the VLDB Endowment, Vol. 8, No. 8
Copyright 2015 VLDB Endowment 2150-8097/15/04.

Since our work relates to OLTP applications, a primary concern is transaction response time. Our observations show that large compaction events have an especially negative impact on the response time of reads due to two issues. First, during the compaction, the compaction process itself competes for resources with the actual workload, degrading its performance. A second major problem are the cache misses that are induced upon the compaction's completion. Distributed key-value datastores generally rely heavily on main memory caching to achieve low latencies for reads[3]. In particular, if the entire working dataset is unable to fit within the provisioned main memory, the read cache may experience a high degree of churn, resulting in very unstable read performance. Since distributed datastores are designed to be elastically scalable, it is generally assumed that a sufficient amount of servers can be conveniently provisioned to keep the application's growing dataset in main memory. Even so, under update-intensive workloads, frequent compactions can become another problematic source of cache churn. Since a compaction consolidates the contents of multiple input files into a new output file, all references to the input files are obsoleted in the process, which necessitates the datastore to invalidate the corresponding entries in its read cache. Our analysis shows that the cache misses caused by these large-scale evictions result in an extended degradation in read latency, since the datastore server then has to read the newly compacted data from the filesystem into its cache all over again.

With this in mind, our second major contribution is to propose a novel approach that attempts to keep the impact of compactions on the performance of the actual workload as small as possible – both during a compaction's execution, *and* after it completes.

As a first step, we offload the compactions themselves to dedicated nodes called *compaction servers*. Taking advantage of the data replication inherent in distributed datastores, we enable a compaction server to transparently read and write replicas of the data by acting as a specialized peer of the datastore servers.

In the second step, aiming to reduce the overhead of cache misses after the compaction, we use the compaction server as a remote cache for the datastore server. That is, instead of reading the newly compacted files from the filesystem, the datastore server reads them directly from the compaction server's cache, thereby significantly reducing both load time and read latency. Although this alleviates the performance penalty incurred by local cache misses to a great degree, it does not completely eliminate it. In order to address the remaining overhead of these cache misses, one approach could be to eagerly warm the datastore server's cache with the compacted data immediately upon the compaction's completion. But such an approach is only feasible when we have enough main memory provisioned, such that the datastore server can simultaneously fit both the current data as well as the compacted data in its cache, thus allowing for a seamless switch between the two. Instead, we propose a smart warmup algorithm that fetches the compacted data from the remote cache in sequential chunks, where each chunk replaces the corresponding range of current data in the local cache. During this *incremental* warmup phase, we guarantee that each read request is served completely either by the old data files or by the freshly compacted data. This ensures that all incoming read requests can be served from the datastore server's local cache even as it is warming up, thereby completely eliminating the performance penalty associated with switching over to the newly compacted data.

[3]http://www.slideshare.net/xefyr/hbasecon2014-low-latency

In short, the main contributions of this paper are:

1. An experimental analysis of the performance impacts associated with compactions in HBase and Cassandra.
2. A scalable solution for offloading compactions to one or more dedicated compaction servers.
3. A solution for efficiently streaming the compacted data from the compaction server's cache into the datastore server's local cache over the network.
4. A smart algorithm for incrementally warming up the datastore server's cache with the compacted data.
5. An implementation of the above and its evaluation based on HBase.

Our paper does not follow the typical structure found in research papers, which first present the solution in full followed by the experiments. Instead, we use a step-wise approach, where we first describe a part of our solution, immediately accompanied by an experimental evaluation of this part to better understand its implications. In this spirit, Section 2 provides an overview of the log-structured merge tree, HBase, and Cassandra. Section 3 describes the overall architecture of our approach and a high-level description of the integration of our new components into HBase. Section 4 then shortly describes the experimental setup, before Section 5 digs into the details of our solution and their evaluation. Section 6 discusses scalability, along with some further experimental results, and Section 7 discusses the fault-tolerance aspects of our solution. Section 8 presents a summary of the related work. We conclude in Section 9.

2. BACKGROUND

This section provides an overview of the background relevant to our understanding of compactions in LSMT-based datastores, as well as a short overview of how compactions are performed in HBase and Cassandra.

2.1 LSMT

The log-structured merge-tree (LSMT) [13] is a key-value data structure that aims to provide a data storage and retrieval solution for high-throughput applications. It is a hybrid data structure, with a main memory layer (C0) placed on top of one or more filesystem layers (C1, and so on). Updates are collected in C0 and flushed down to C1 in batches, such that each batch becomes an immutable file, with the key-value pairs written in sorted order. This approach has several important implications.

Firstly, for the client, updates are extremely fast, since they are applied in-memory. Secondly, flushing updates down in batches is more efficient since it significantly reduces disk I/O. Moreover, appending a batch of updates to a single file is much faster than executing multiple random writes on a rotational storage medium (e.g., magnetic disk). This enables the data structure to support high update throughputs. Thirdly, multiple updates on a given key may end up spread across C0 and any number of files in C1 (or below). In other words, we can have multiple values per key. Therefore, a random read on a given key must first search through C0 (a quick, in-memory lookup), then C1 (traversing all the files in that layer), and so on, until it

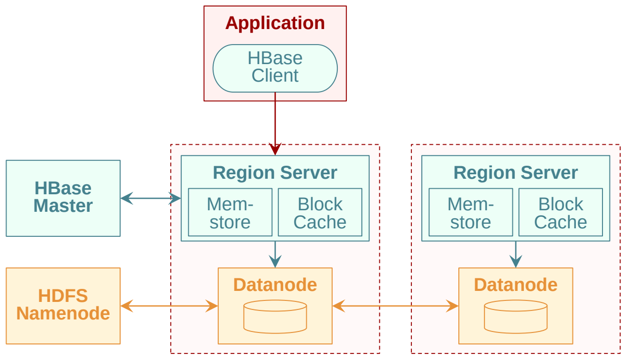


Figure 1: HBase Architecture

finds the most recent value for that key. Since the contents of a file are already sorted by key, an in-file index can be used to speed up random reads within a file.

These read-only files inevitably start building up in the filesystem layer(s), resulting in reads becoming increasingly slow over time. This is remedied by periodically selecting two or more files in a layer and merge-sorting them together into a single file. The merge process overwrites older values with the latest ones and discards deleted values, thereby clearing up any stale data.

2.2 HBase

HBase is a modern distributed key-value datastore inspired by Bigtable [3]. HBase offers the abstraction of a table, where each row represents a key-value pair. The key part is the unique identifier of the row, and the value part comprises an arbitrary number of column values. Columns are grouped into column families to partition the table vertically. Each table can also be partitioned horizontally into many *regions*. A region is a contiguous set of rows sorted by their keys. When a region grows beyond a certain size, it is automatically split into half, forming two new regions.

Every region is assigned by a Master server to one of multiple *region servers* in the HBase cluster (see Figure 1). Through well-balanced region placement, the application workload can be evenly distributed across the cluster. When a region server becomes overloaded, some of its regions can be reassigned to other underloaded region servers. When the cluster reaches its peak load capacity, new region servers can be provisioned and added to the online cluster, thus allowing for elastic scalability. HBase relies on Zookeeper[4], a lightweight quorum-based replication system, to reliably manage the meta-information for these tasks.

HBase uses HDFS[5] as its underlying filesystem for the application data, where each column family of each region is physically stored as one or more immutable files called *storefiles* (corresponding to LSMT layer C1). HDFS is a highly scalable and reliable distributed filesystem based on GFS [9]. It automatically replicates file blocks across multiple *datanodes* for reliability and availability. Normally, there is a datanode co-located with each region server to promote data locality. HDFS has a Namenode, similar in spirit to the HBase Master, for meta-management.

[4]http://zookeeper.apache.org/
[5]http://hadoop.apache.org/

Applications interact with HBase through a client library that provides an interface for reading and writing key-value pairs, either individually or in batches, and performing sequential scans that support predicate-based filtering. Each read (i.e., *get* or *scan*) or write (i.e., *put* or *delete*) request is sent to the region server that serves the region to which the requested key-value pair(s) belongs. A write is served simply by applying the received update to an in-memory data structure called a *memstore* (corresponding to LSMT layer C0). This allows for having multiple values for each column of a row. Each region maintains one memstore per column family. When the size of a memstore reaches a certain threshold, its content are flushed to HDFS, thereby creating a new storefile. A read is served by scanning for the requested data through the memstore and through all region's storefiles that might contain the requested data. Each region server maintains a *block cache* that caches recently accessed storefile blocks to improve read performance.

Periodically, or when the number of a region's storefiles crosses a certain configurable limit, the parent region server will perform a compaction to consolidate the contents of several storefiles into one. When a compaction is thus triggered, a special algorithm decides which of the region's storefiles to compact. If it selects all of them in one go, it is called a *major* compaction, and a *minor* compaction otherwise. Unlike a minor compaction, a major compaction additionally also removes values that have been flagged for deletion via their latest updates. Therefore, major compactions are more expensive and usually take much longer to complete.

2.2.1 Exploring Compactions

The default compaction algorithm in HBase uses a heuristic that attempts to choose the optimal combination of storefiles to compact based on certain constraints specified by the datastore administrator. The aim is to give the administrator a greater degree of control over the size of compactions and thus, indirectly, their frequency as well. For example, it is possible to specify minimum and maximum limits on the number of storefiles that can be processed per compaction. Similarly, the algorithm also allows us to enforce a limit on the total file size of the group, so that minor compactions do not become too large. Finally, a ratio parameter can be specified that ensures that the size of each storefile included in the compaction is within a certain factor of the average file size of the group. The algorithm explores all possible permutations that meet all these requirements and picks the best one (or none), optimizing for the ratio parameter. We can configure HBase to use different ratio parameters for peak and off-peak hours.

2.3 Cassandra

Cassandra is another popular distributed key-value datastore. Its design incorporates elements from both Bigtable and Dynamo [7]. As a result, it has a lot in common with HBase, yet also differs from it in several important respects.

Unlike HBase, Cassandra has a decentralized architecture, so a client can send a request to any node in the cluster, which then acts as a proxy between the client and the nodes that actually serve the client's request. Cassandra also allows applications to choose from a range of consistency settings per request. The lowest setting allows for inconsistencies such as stale reads and dirty writes (though, eventually, the datastore does reach a consistent state), but offers supe-

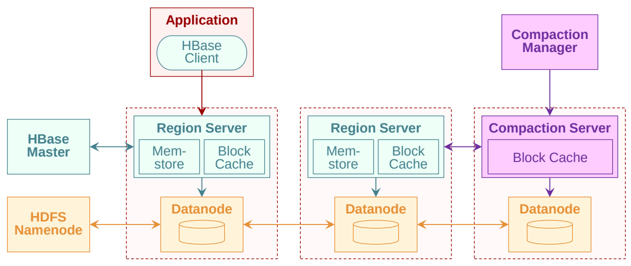


Figure 2: Offloading Compactions

rior performance. The strictest setting matches the consistency level of HBase, but sacrifices on performance. While HBase maintains its own block cache, Cassandra relies instead on the OS cache for faster access to hot file blocks. At a finer granularity, it also offers the option of using a row-level cache. Finally, Cassandra uses a slightly different compaction algorithm (see tiered compactions in Section 8). Unlike HBase, minor compactions in Cassandra clean up deleted values as well. Cassandra also throttles compactions to limit their overhead.

Despite these differences, Cassandra has two important similarities with HBase: Cassandra also partitions its data and runs compactions on a per-partition basis. Moreover, Cassandra also flushes its in-memory updates in sorted batches into read-only files. These similarities make us believe that many aspects of our approach, although implemented in HBase, are generally applicable to Cassandra and other datastores in the LSMT family as well.

3. ARCHITECTURE

Our solution adds two new components to the datastore architecture: a centralized *compaction manager* and a set of *compaction servers*. The integration of these components into the HBase architecture is depicted in Figure 2.

A compaction server performs compactions on behalf of region servers. Therefore, it also hosts a datanode in order to gain access to the HDFS layer. Whenever a region server flushes region data, it writes a new storefile to HDFS, which can then be read by the compaction server. Similarly, upon compacting a region, the compaction server writes the compacted storefile back to HDFS as well.

Compaction servers can be added or removed, allowing for scalability. Each compaction server is assigned some subset of the data. The compaction manager manages these assignments, mapping regions to compaction servers akin to how the HBase Master maps regions to region servers.

While our implementation makes substantial additions and changes to HBase, we have attempted to perform them in a modular manner. We used the HBase Master and region server code as a base for implementing the compaction manager and the compaction server, respectively. For example, the compaction server reuses the code for scanning storefiles from HDFS and performing compactions on them. That is, we take the compaction algorithm as a black box, without modifying it. However, we modified specific subcomponents of the region server code so that it could offload compactions to a compaction server and also receive the compacted data back over the network for more efficient warmup.

4. EXPERIMENTAL SETUP

Since the next section combines the presentation of our proposed solutions along with a detailed performance analysis of each of the steps, we provide a summary of the general experimental setup before proceeding.

4.1 Environment

We ran our experiments on a homogeneous cluster of 20 Linux machines. Each node has a 2.66 GHz dual-core Intel Core 2 processor, 8 GB of RAM, and a 7,200 RPM SATA HDD with 160 GB. The nodes are connected over a Gigabit Ethernet switch. The OS is 64-bit Ubuntu Linux and the Java environment is 64-bit Oracle JDK 7. We used the following software versions: HBase 0.96, HDFS 2.3, Cassandra 2.0, and YCSB 0.1.4.

4.2 Datastores

4.2.1 HBase/HDFS

The HBase Master, the HDFS Namenode, and ZooKeeper services all share one dedicated node[6]. We modified a few key configuration parameters in HBase in order to better study the overheads of compactions. The compaction file selection ratio was changed from 1.2 to 3.0. Region servers were allocated 7 GB of main memory, of which 6 GB went to the block cache. We used Snappy[7] for compression. In all our experiments, each region server and compaction server hosts their respective datanode, with a minimum of three datanodes in the cluster.

4.2.2 Cassandra

Since Cassandra prefers to use the OS cache, we allocated only 4 GB of main memory to its process and kept the row cache disabled. We used the ByteOrderedPartitioner, which allows us to efficiently perform sequential scans by primary key (the default, random partitioner is unsuitable for this purpose). Since the standard YCSB binding for Cassandra is outdated, we implemented a custom binding for Cassandra 2.0 using the latest CQL 3 API.

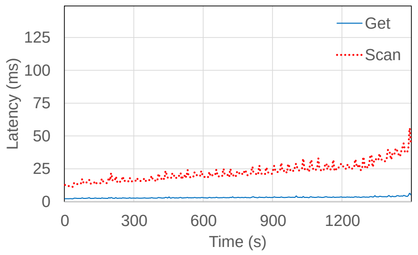
4.3 Benchmarks

We are interested in running OLTP workloads on a cloud datastore. A typical OLTP workload generates a high volume of concurrently executing read-write transactions. Most transactions execute a series of short reads and updates, but a few might also execute larger read operations such as partial or full table scans. In our experiments, we try to emulate these workload characteristics with two benchmarks.

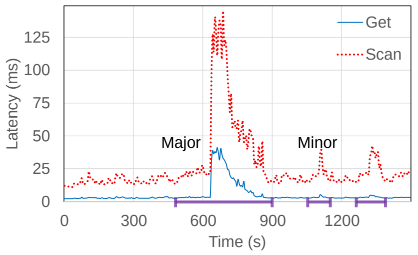
YCSB is a popular microbenchmark for distributed datastores. We used it to stress both HBase and Cassandra with an update-intensive workload. We launch separate client processes for reads and writes. Our write workload consists of 100% updates, while our read workload comprises 90% gets and 10% scans. We used the Zipfian distribution to reflect an OLTP workload more closely.

TPC-C is a well-known OLTP benchmark that is generally used for benchmarking traditional relational database setups. We used an implementation of TPC-C called

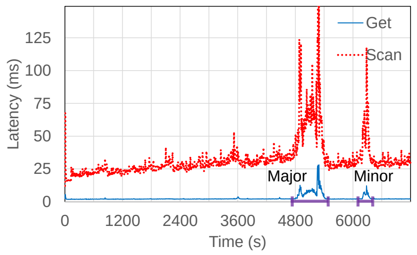
[6]Reliability was not a focus of the evaluation, so we provisioned one ZooKeeper server only, with sufficient capacity.
[7]https://code.google.com/p/snappy/



(a) No Compactions (HBase)



(b) Compactions (HBase)



(c) Compactions (Cassandra)

Figure 3: Motivation: (a) Under an update-intensive workload, read latency in HBase gets increasingly worse over time if the storefiles that build up are not regularly compacted (the figure is scaled for scans, but gets are affected just the same). (b) Although regular compactions help maintain read performance within reasonable limits over the long run, read latency still spikes significantly during the compaction events themselves. (c) Cassandra suffers from the same problem; we can see that the larger of the two compactions has a significant negative impact on read performance over a period of around ten minutes; note the same two distinct phases.

PyTPCC[8], which works with various cloud datastores, including HBase. Since there is no support for transactions in HBase, the benchmark simply executes its transactions without ACID guarantees. For convenience, it does not simulate the *think time* between transactions, thus allowing us to stress the datastore with less clients. The workload comprises five transaction types: New-Order (45%), Payment (43%), Order-Status (4%), Delivery (4%), and Stock-Level (4%). We populated 50 warehouses, corresponding to around 14 GB of actual data.

5. OFFLOADING COMPACTIONS

A key performance goal of OLTP applications is maintaining low response times under high throughput. In this section, we first show that read performance can suffer significantly during and immediately after a large compaction, in both HBase and Cassandra. We then propose and evaluate a number of strategies for alleviating this problem.

5.1 Motivation

To understand the implications of HBase compactions on read performance, we ran a YCSB workload with 10 read threads against one region server (no compaction server). Our test table held three million rows in a single region, equivalent to around 4 GB of actual, uncompressed data. This ensured that the working dataset fit comfortably within the region server's 6 GB block cache. We recorded the response times of gets and scans over the course of the experiment, at five-second intervals.

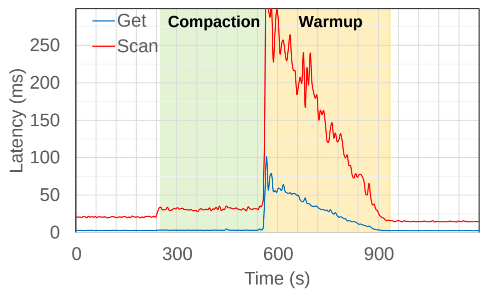
The graphs in Figure 3 show the response time of gets and scans over the duration of each experiment. Figure 3(a) shows the observed degradation in read performance over time when compactions are disabled altogether. Figure 3(b) shows that while compactions help maintain read performance within reasonable limits over the long run, each compaction event causes a significant spike in response time. We can also see that a major compaction causes a much larger and longer degradation in read performance relative to minor compactions. Note that both gets and scans are severely affected by the major compaction. A similar experiment on Cassandra shows that it also exhibits severe compaction-related performance degradation (see Figure 3(c)).

[8]http://github.com/apavlo/py-tpcc

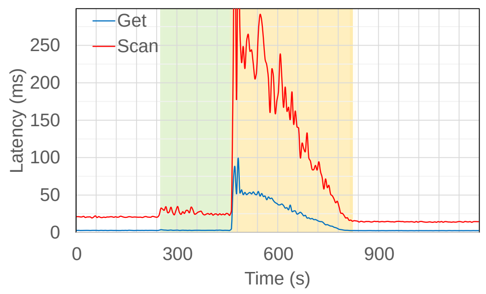
Figure 4(a) zooms into the compaction phase. We can see that a major compaction can add a noticeable performance overhead on the region server that executes it, and can typically take on the order of a few minutes to complete. The response times of read operations executing on this region server degrade noticeably during this time. The figure shows two distinct phases of degradation: *compaction* and *warmup*. The compaction phase is characterized by higher response times over the duration of the compaction. We observed that this is mainly due to the CPU overhead associated with compacting the storefile data. Both gets and scans are affected. The warmup phase starts when the compaction completes. At this time the server switches from the current data to the newly compacted data. The switch triggers the eviction of the obsoleted file blocks en masse, followed by a flurry of caches misses as the compacted data blocks are then read and cached. This leads to a severe degradation in read response times for an extended period. Figure 3(c) shows that Cassandra similarly exhibits these two phases as well.

5.2 Compaction Phase

We first attempt to deal with the overhead of the compaction phase. Our observations show that the performance degradation in this phase can be exacerbated by the datastore server experiencing high loads. In other words, overloading an already saturated processor can cause response times to spike and the compaction itself to take much longer to complete. One approach to manage this overhead is for the datastore to limit the amount of resources that a compaction consumes. By throttling compactions in this way, the datastore can amortize their cost over a longer duration. In fact, this is the approach taken by Cassandra; it throttles compaction throughput to a configurable limit (16 MB/s by default). However, we believe that this approach does not sufficiently address the problem, for three reasons mainly. Firstly, Figure 3(c) shows that despite the throttling, response times still spiked with the compaction, just as was observed with HBase with no throttling. We could, of course, throttle more aggressively, thereby amortizing the overhead over a much longer period, but this leads to our second concern. The longer a compaction takes, the more obsoleted data (deleted and expired values) the datastore server must maintain over that duration, thus continuing to



(a) Compaction Phases (SS)



(b) Compaction Offloading (CO)

Figure 4: Compaction Phase

hurt read performance. Thirdly, even when throttling compactions helps to alleviate their overhead to some extent, it offers no further benefits for managing the overhead of the subsequent warmup phase.

Therefore, our approach offloads these expensive compactions to a dedicated compaction server, thus allowing the region server to fully dedicate its resources towards serving the actual application workload. There are two obvious benefits to this approach. First, it eliminates the CPU overhead that the region server would otherwise incur over the duration of the compaction. Second, the compaction can generally be executed faster, since it is running on a dedicated server. Although the compaction server needs to read the compaction's input storefiles from the filesystem rather than main memory (the region server can read the data from its block cache), we could not observe any negative impact in our experiment as a result of this. We evaluated the benefit of offloading the compaction using YCSB. Figure 4(b) plots the response times of gets and scans under our approach, where we simply added the compaction manager and one compaction server to the previous experiment. Comparing Figure 4(b) against the standard setup in Figure 4(a), we can see that with a dedicated compaction server, the compaction phase is shorter, with a noticeable improvement in read latency as well. On the other hand, we see no improvement in the long-running warmup phase after the compaction has completed. Therefore, next, we discuss the advantages of having the compacted data in the compaction server's main memory for improving the warmup phase.

5.3 Warmup Phase

As previously discussed, we observe that once the compaction completes, the region server must read the output storefile from disk back into its block cache in order to serve reads from the newly compacted data. At this stage, read performance can suffer significantly due to the high rate of cache misses as the block cache gradually warms up again. The experimental results presented so far clearly show that the warmup phase has a significant negative impact on the performance of our workload. In fact, we tend to see an extended phase of up to a few minutes of severely degraded response times for both individual gets as well as scans. Therefore, in the remainder of this section, we analyze this particular performance issue and attempt to mitigate it.

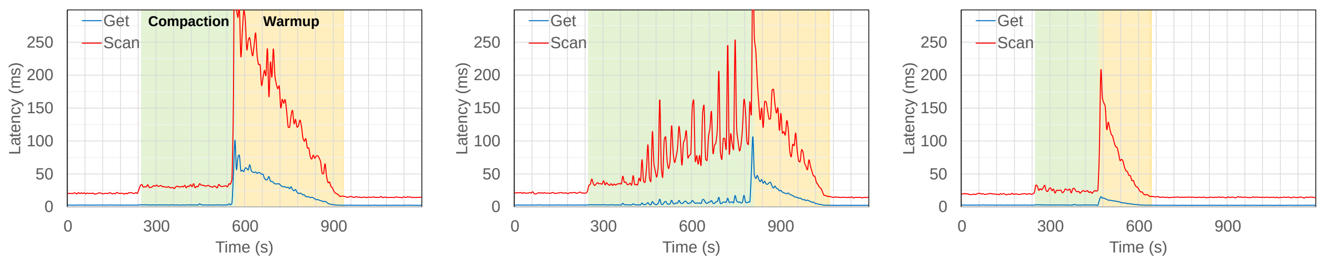
5.3.1 Write-Through Caching

First, we analyze the warmup phase in the standard setup (i.e., the region server does not offload the compaction). We consider whether caching a compaction's output in a *write-through* manner – i.e., each block written to HDFS is simultaneously cached in the block cache as well – could present any benefit under the standard setup. Ideally, this would eliminate the need for a warmup phase altogether. However, our observations show that this approach does not in fact yield promising results. In order to test this idea, we modified HBase to allow us to cache compacted blocks in a write-though manner. In Figure 5(b), we can compare the performance of this approach against the standard setup (Figure 5(a)). We see that while the warmup phase improves to an extent, the performance penalty is passed back to the compaction phase instead. Upon further investigation, we witnessed large-scale evictions of hot blocks from the block cache during the compaction, resulting in heavy cache churn, which severely degraded read performance. In other words, we see that during the course of the compaction, the newly compacted data competes for the limited capacity of the region server's block cache even as the current data is still being read, since the switch to the new data is made only once the compaction completes. Therefore, this approach only shifts the problem back to the compaction phase.

Clearly, the larger the main memory of each region server is, compared to the size of the regions it maintains, the more of the current data *and* compacted data will fit together into main memory, and the less cache churn we will observe. However, that would lead to a significant over-provisioning of memory per region server, since the extra memory would only be used during compactions. For this reason, we believe that having a few compaction servers acting as remote caches that are shared by many region servers, can solve this problem with less overall resources.

5.3.2 Remote Caching

Our approach of offloading compactions presents us with an interesting opportunity to take advantage of write-through caching on the compaction server instead, thereby combining both approaches. As a dedicated node, it can be asked to play the role of a remote cache during the warmup phase since it already has the compaction output cached in its main memory. With this approach, instead of reading the newly compacted blocks from its local disk, the region server requests them from the compaction server's memory instead. There is an obvious trade-off here between disk and network I/O. Since our main aim is achieving better response times, we deem this trade-off to be worthwhile for setups where network I/O is faster than disk I/O.



(a) Standard Setup (SS) (b) Write-Through Caching (WTC) (c) Remote Caching (RC)

Figure 5: Warmup Phase

We have implemented a remote procedure call that allows the region server to fetch the cached blocks from the compaction server instead of reading them from the local HDFS datanode. To reduce the network transfer overhead, we compress the blocks at the source using Snappy, and subsequently uncompress them upon receiving them at the region server. This comes at the cost of a slight processing overhead, but the savings in the total transfer time and network I/O make this an acceptable trade-off. We evaluate the effectiveness of this approach in Figure 5(c), which shows a significant improvement in response times in the warmup phase as compared to not having a remote cache available. While the eviction of the obsoleted data blocks still causes cache misses, note that the warmup phase completes quicker due to the much faster access of blocks from the compaction server's memory over the network rather than from disk. Of course, the benefit of compaction offloading on the compaction phase is retained as well.

Nevertheless, we still observe a distinct performance boundary between the compaction and warmup phases where the cache misses occur. Hence, while the remote cache offers a significant improvement over reading from the local disk, the performance penalty due to these cache misses remains to be addressed.

5.4 Smart Warmup

To obtain further improvements, we essentially need to avoid cache misses by preemptively fetching and caching the compacted data. We discuss two options for doing this.

5.4.1 *Pre-Switch Warmup*

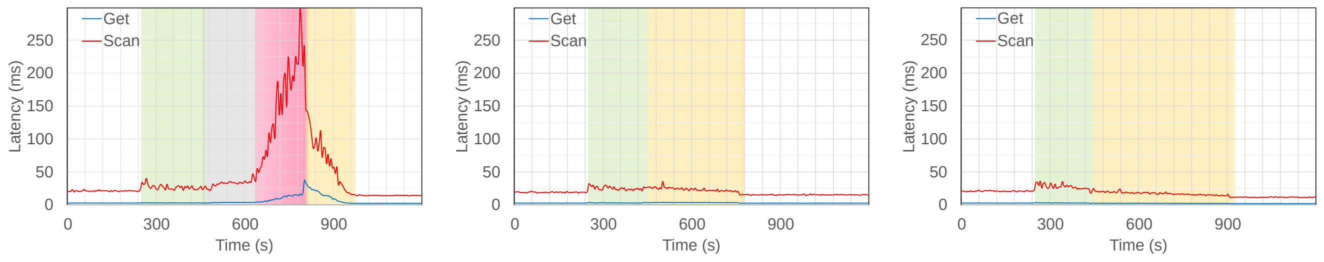
In the first option, we warm the local cache up (transferring data from the compaction server to the region server) *prior* to making the switch to the compacted data. This is similar in principle to the write-through caching approach previously discussed. That is, its effectiveness depends on the availability of additional main memory, such that the region server can simultaneously fit both the current data as well as the compacted data in its cache, thus allowing for a seamless switch. When compared with write-through caching, in which the warmup happens *during* the compaction itself, here we perform the warmup *after* the compaction completes. Therefore, since the compaction is performed remotely and the compacted data fetched over the network, the region server's performance does not suffer during the compaction, and, once the switch is made, the remainder of the warmup is more efficient as well.

Figure 6(a) shows the performance of this approach. The pre-switch warmup comprises two sub-phases, depicted in the figure using gray and pink, respectively. Recall that 6 GB of main memory is available for the block cache. Since the current data takes up around 4 GB, the pre-switch warmup can fill up the remaining 2 GB without severely affecting the performance of the workload (gray). However, as the warmup continues beyond this point (pink), the compacted data competes with the current data in the cache, resulting in severely detrimental cache churn. This also affects post-switch performance (orange), since we must then re-fetch the compacted data that was overwritten by the current data. Therefore, the longer the pre-switch warmup phase takes, the less effective this approach becomes. Nevertheless, its overall performance is still better than the write-through caching approach without the compaction server (Figure 5(b)), since, in the latter case, the old and new data already start to compete during the compaction phase once the block cache fills up; whereas, with the remote cache, the detrimental cache churn occurs only for a much shorter part of the pre-switch warmup phase.

Since OLTP workloads typically generate regions of hot data, we also tried a version of this approach where we warm the cache up with only as much hot data as can fit side-by-side with the current data (gray) so that we do not cause any cache churn (pink). However, this strategy appeared to offer no additional benefit when tested. We realized that this is because the hot data comprises less than 1% of the blocks, which can easily be fetched almost immediately in either case (before or after the switch), meaning that 99% of cache misses are actually associated with cold data.

5.4.2 *Incremental Warmup*

Our experimental analysis above shows that the less additional memory is provisioned on the region server, the worse the pre-switch warmup will perform. Therefore, we now present an *incremental* warmup strategy that solves this problem without requiring the provisioning of additional memory. It works on two fronts. The first aspect is that we fetch the compacted data from the remote cache in sequential chunks, where each chunk replaces the corresponding range of current data in the local cache. For this, we exploit the fact that the storefiles written by LSMT datastores are pre-sorted by key. Hence, we can move sequentially along the compacted partition's key range. That is, we first transfer the compacted data blocks with the smallest keys in the storefiles. At the same time, we evict the current data blocks that cover the same key range that we just transferred. That is, the newly compacted data blocks replace the data blocks with the same key range. At any given time, we keep track of the *incremental warmup threshold* T which represents the



(a) Pre-switch Warmup (PSW) (b) Incremental Warmup (IW) (c) Throttled Incremental (TIW)

Figure 6: Smart Warmup

row key with the following property: all newly compacted blocks holding row keys smaller or equal to T have been fetched and cached, and, correspondingly, all current blocks holding rows keys up to T have been evicted from the local cache. This means that all current blocks with row keys larger than T have not been evicted yet and are still in the region server's cache.

Read operations are now executed in the following way on this mixed data. Given a get request for a row with key R, or a scan request that starts at key R, we direct it to read the newly compacted storefile if $R \leq T$, and the current storefiles (can be one or more, with overlapping key ranges) otherwise. In this way, we ensure that all incoming requests can be served immediately from the region server's block cache even as it is warming up, thus removing the overhead associated with cache misses. As Figure 6(b) shows, the improvement offered by this approach is significant.

While a get only reads a single row, a scan spans multiple rows and thus could potentially span multiple blocks of a storefile. Therefore, a scan may fall under one of the three following cases. If the scan starts and ends below the incremental threshold, T, it will read only compacted data that is already cached. If the scan starts below but ends beyond T, it will still read the compacted data, although all of this data might not yet be cached when the scan starts. But as the scan progresses, so will T, in parallel, as the compacted data is streamed into the region server, and thus, this scan will most likely be fully covered by the cache as well. Only in the case that the scan overtakes T, accessing keys with a value higher than the current T, it will slow down due to cache misses. If the scan starts and ends beyond T, it will read the current data instead and will also, most likely, be fully covered by the cache. In the case that T overtakes it midway, evicting the blocks it was about to read, it will encounter cache misses. However, since scanning rows from locally cached blocks is faster than fetching blocks from the remote cache, we do not expect or observe this to happen often. In fact, we saw relatively very few cache misses overall in our experiment. Note that in all cases, any given read request is served either entirely from the compacted data or entirely from the current data.

Moreover, note that a region may comprise multiple column families, and each family has its own storefile(s). The algorithm iterates over the region's column families, warming them up one at a time. Therefore, when such a region receives a read request covering multiple column families during the incremental warmup, we ensure that a consistent result is returned, since each family is individually read consistently before the results are combined.

	Compaction			Warmup		
	Degradation (%)		Duration (mm:ss)	Degradation (%)		Duration (mm:ss)
	Get	Scan		Get	Scan	
SS	11.5	53	05:20	1,020	642	06:10
WTC	106	274	09:20	667	408	04:30
CO	6.67	23.3	03:25	1,040	642	06:00
RC	6.91	28.8	03:30	121	224	03:20
PSW	86.4	202	09:20	389	238	02:55
IW	6.84	27.6	03:15	26.2	19.7	05:25
TIW	6.51	20.1	03:20	9.30	3.75	07:45

Figure 7: Performance Evaluation: YCSB

As a final improvement, we throttle the warmup phase. The result is shown in Figure 6(c). This essentially means that T advances slower than without throttling, and, therefore, the warmup phase lasts longer. However, as a result, the performance overhead of this phase is virtually eliminated. It reduces the CPU costs for the data transfer and reduces the chances of cache misses caused by current data blocks getting overwritten by the new data too quickly. As a result, we see that there is hardly any noticeable impact left from the compaction and warmup phases.

A summary of our YCSB performance evaluation is presented in Figure 7. For each approach, we show the degradation of read latency during the compaction and warmup phases, respectively, as a measure of the percentage difference from the baseline, i.e., the average latency before the compaction started. The important improvements are highlighted in green. We can see that with our best approach, throttled incremental warmup (TIW), the performance degradation of gets is reduced to only 7%/9% (compaction/ warmup), while that of scans is reduced to only 20%/4%. The duration of the compaction phase is significantly shortened as well. Although the warmup phase is longer than with simple remote caching (RC), the significantly superior performance of TIW makes up for this.

5.5 TPC-C

We use TPC-C, a standard OLTP benchmark, to evaluate the performance of our proposed approaches. On the backend, we ran two region servers and one compaction server, while a total of 80 client threads were launched using two front-end nodes. We recorded the average response time of each transaction type, and also measured the *tpmC* metric (New-Order transactions per minute) averaged over the du-

	Compaction		Warmup		tpmC
	Degrad. (%)	Duration (mm:ss)	Degrad. (%)	Duration (mm:ss)	
SS	20.6	11:40	163	05:00	5201
CO	12.9	08:00	161	05:00	5212
RC	12.4	08:00	13.9	03:00	5706
TIW	12.5	08:00	10.6	05:00	5755

(a) New-Order

	Compaction		Warmup	
	Degrad. (%)	Duration (mm:ss)	Degrad. (%)	Duration (mm:ss)
SS	6.44	14:00	150	02:40
CO	5.67	07:20	152	02:40
RC	5.51	07:20	36.4	02:40
TIW	5.54	07:20	8.92	02:00

(b) Stock-Level

Figure 8: Performance Evaluation: TPC-C

ration of each experiment. In order to observe the adverse impacts of compactions on the standard TPC-C workload, we triggered compactions on the two most heavily updated tables, Stock and Order-Line, in two separate sets of experiments, respectively.

In the first set, we observed the performance of New-Order, which is a short, read-write transaction. Since it reads the Stock table, it is impacted by compactions running on this table. Figure 8(a) shows the effects of this impact under the standard setup (SS) and the improvements offered by each of our main approaches. We can see that with throttled incremental warmup (TIW), the degradation in the average response time of New-Order transactions (against the baseline), is significantly reduced in both the compaction and warmup phases. The duration of the compaction phase is also considerably shortened. The warmup phase is shortest when using simple remote caching (RO). Overall, our best approach, TIW, provides an improvement of nearly 11% in terms of the tpmC metric.

In the second set, we observed the longer-running Stock-Level transaction. Since it reads the Order-Line table, it was impacted by compactions running on this table. In Figure 8(b), we see the performance improvement provided by each of our approaches. While the response time is only slightly better in the compaction phase, its duration is cut down considerably by offloading the compaction. Once again, a significant reduction in response time degradation is seen with our incremental warmup (TIW) approach, even though the warmup duration stays nearly the same.

6. SCALABILITY

By using a compaction manager that oversees the execution of compactions on all compaction servers, we can scale our approach in a similar manner as HBase can scale to as many region servers as needed. In fact, since HBase partitions its data into regions, we conveniently use the same partitioning scheme for our purposes. Thus, the distributed design of our solution inherits the elasticity and load distribution qualities of HBase.

6.1 Elasticity

For application workloads that fluctuate over time, HBase offers the ability to add or remove region servers as the need arises. Along the same lines, our compaction manager is able to handle many compaction servers at the same time. It uses the same ZooKeeper-based mechanism as HBase for managing the meta-information needed for mappings between regions and compaction servers.

6.2 Load Distribution

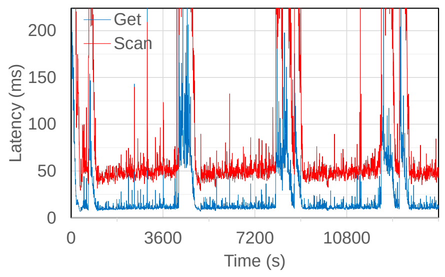
As the application dataset grows, HBase creates new regions and distributes them across the region servers. Our compaction manager automatically detects these new regions and assigns them to the available compaction servers. We inherit the modular design of HBase, which allows us to plug in custom load balancing algorithms as required. We currently use a simple round-robin strategy for distributing regions across compactions servers. However, we can envision more complex algorithms that balance regions dynamically based on the current CPU and memory loads of compaction servers – metrics that we publish over the same interface that HBase uses for its other components.

6.3 Compaction Scheduling

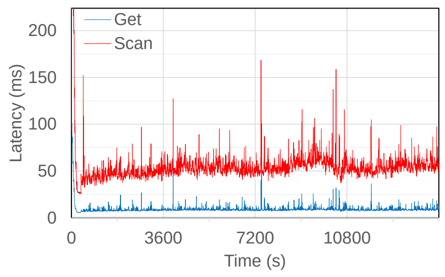
Scheduling compactions is an interesting problem. Currently, we let the region server schedule its own compactions based on its default exploring algorithm (see Section 2.2.1). However, our design allows for the compaction manager to perform compaction scheduling based on its dynamic, global view of the loads being handled by compaction servers.

An important parameter is how many compactions a compaction server can handle concurrently. As we use its main memory as a remote cache, the sum of the compacted data of all regions it is concurrently compacting should not be larger than the server's memory. A rough estimation of this limit can be calculated as follows.

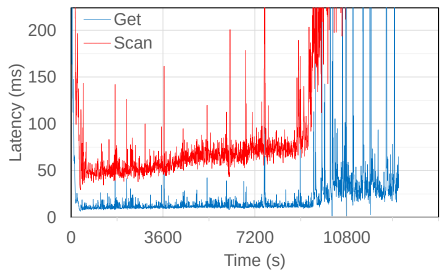
Given an estimate of the rate c (in bytes/s), at which a compaction server can read and compact data, and an estimate of the rate w (in bytes/s), at which the compacted data is transferred back to the region server over the network (with throttling), we can calculate the duration, $D(b)$ (in seconds), of a compaction as a function of its size, b (in bytes): $D = b/c + b/w$. Moreover, a compaction server with m bytes of main memory cache at its disposal can handle l compactions of average size b at a time, where $l = \lfloor m/b \rfloor$. Thus, one compaction server will have the capacity to compact up to h regions of average size b per interval of t seconds, where $h = t/D(b) * \lfloor m/b \rfloor$. Therefore, given an update workload that triggers T compactions per region per interval of t seconds, we can assign up to $\lfloor h/T \rfloor$ regions per compaction server. And, for a dataset of R regions, we will need to provision at least $C = \lceil R/\lfloor h/T \rfloor \rceil$ of these compaction servers for the given application dataset size and workload. For example, consider a setup with the following parameters: $c = 20\ MB/s$, $w = 8\ MB/s$, $m = 6\ GB$, $b = 4\ GB$, $T = 1/hour$, and $R = 10\ regions$. This gives us $C = 2$ compaction servers.



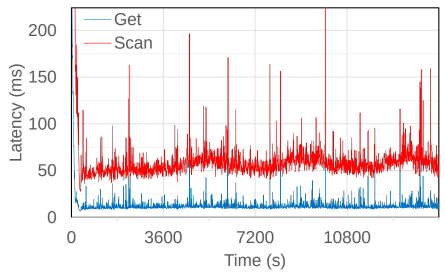
(a) Standard Setup: 5 RS



(b) Compaction Offloading: 5 RS / 1 CS



(c) Compaction Offloading: 10 RS / 1 CS



(d) Compaction Offloading: 10 RS / 2 CS

Figure 9: Performance Evaluation: Scalability

6.4 Performance Evaluation

Using YCSB, we demonstrate the scalability of our solution by scaling our setup from five region servers up to ten. The five-node setup served 10 million rows split into five regions supported by one compaction server. We launched two read/write clients (with 40 read threads and two write threads each). The ten-node setup doubled both the dataset size and workload; i.e., 20 million rows split into ten regions, stressed with four read/write clients. At first, we provisioned only one compaction server, overloading it beyond its maximum capacity. Next, we ran the same experiment with two compaction servers to demonstrate the capability of our architecture to effectively distribute the load between the two servers. The experiments run for four hours; multiple major compactions are triggered in this duration.

Figures 9(a) to 9(d) show the results. Figure 9(a) shows the average response time of reads over the four-hour period on the five region servers (no compaction servers). We can see the same latency spikes as in our smaller scale experiments where compactions were not offloaded. Figure 9(b) shows the five-node setup with one compaction server, which can handle the compactions triggered by all five region servers, eliminating the performance overhead seen in the standard setup. In Figure 9(c), ten region servers are served by a single compaction server. In this case, the compaction server becomes overloaded. As our compaction server only has enough main memory cache available (6 GB) to compact a single region's data (4 GB) at a time, we cannot allow several compactions to run concurrently. Thus, compactions are delayed, and read performance on the region servers gets increasingly worse, as more store files are created that have to be scanned by reads, and the region servers start running out of block cache space as well. Finally, we can observe in Figure 9(d) that with two compaction servers, we can handle the compaction load of ten region servers comfortably, and response times remain smooth over the entire execution.

7. FAULT-TOLERANCE

Our approach offers an efficient solution for offloading compactions while ensuring their correct execution even when components fail. This section addresses several important failure cases and discusses the fault-tolerance of our solution.

7.1 Compaction Server Failure

When the compaction manager detects that a compaction server has failed, it reassign its regions to another available compaction server. A compaction server can be in one of three states at the time of failure: idle, compacting some region(s), or transferring compacted data back to the region server(s). If it was performing a compaction, then its failure will cause a remote exception on the region server and the compaction will be aborted. Note that no actual data loss occurs, since the compaction server was writing to a temporary file, and the region server does not switch over to the compacted file until the compaction has completed. The region server can retry the compaction and it will be assigned to another compaction server. If no compaction servers are available, then the region server can simply perform the compaction itself.

If the compaction server was in the process of transferring a compacted file back to the region server when the failure occurs, this will also cause a remote exception on the other end. In the case of incremental warmup, some requests will already have started reading the partially transferred compacted data. Therefore, the region server needs to finalize loading the compacted data, which it can do by simply reading the storefiles from the filesystem instead, as the compaction server completed writing the new storefiles to HDFS before beginning the transfer to the region server. However, since the remaining portion of the compacted data now needs to be fetched from HDFS, read performance might suffer during the remainder of the warmup phase (as under the standard setup).

7.2 Compaction Manager Failure

In our current implementation, in order to offload a compaction, the region server must go through the compaction manager to be forwarded to the compaction server that will handle the compaction. Thus, the compaction manager becomes a single point of failure in our setup. However, this is only an implementation issue. Since we use ZooKeeper for maintaining the compaction servers' region assignments, our design offers a reliable way for a region server to contact a compaction server directly.

As with the HBase Master, if the compaction manager fails, we lose the ability to add or remove compaction servers and assign regions, so it would need to be restarted as soon as possible to resume these functions. However, ongoing compactions are not affected, since the region server and compaction server communicate directly once connected.

7.3 Region Server Failure

If a region server fails while waiting for an offloaded compaction to return, the compaction server detects the disconnection in the communication channel via a timeout, and the compaction is aborted. Once the HBase Master has assigned the affected regions to another region server, they can simply retry the compaction and it will be handled by the compaction server as a new compaction request. If a region server fails during the incremental warmup phase, the new parent region server must ensure that it loads only the newly compacted file(s) from HDFS, and not any of the older files, which should be discarded at this point. Although we currently do not handle this failure case, we intend to implement a simple solution for it by modifying the file names prior to initiating the incremental warmup. In this way, if the region is reopened by another region server, it can detect which files in the region's HDFS directory can be discarded due to being superseded by the newer compacted files.

8. RELATED WORK

The number of scalable key-value stores, as well as more advanced datastores, providing more complex data models and transaction consistency, has increased very quickly over the last decade [2, 3, 5, 7, 10, 12]. Many of these datastores rely on creating multiple values/versions of data items rather than applying updates in-place, in order to handle high write throughput requirements. However, read performance can be severely affected as trying to find the right data version for a given query takes increasingly longer over time. Therefore, compactions are a fundamental feature of these datastores, helping to regularly clean up expired versions, and thus keep read performance at acceptable levels.

Various types of compaction algorithms exist. In order to make compactions more efficient, these algorithms generally attempt to limit the amount of data processed per compaction by selecting files in a way that avoids the repetitive re-compaction of older data as much as possible. For instance, tiered compactions were first used by Bigtable and also adopted by Cassandra. Rather than selecting a random set of storefiles to compact, this algorithm selects only a fixed number (usually four) of storefiles at a time, picking files that are all around the same size. One effect of this mechanism is that larger storefiles may be compacted less frequently, thereby reducing the total amount of I/O taken up by compactions over time. The leveled compactions algorithm was introduced in LevelDB[9] and was recently also implemented in Cassandra. The aim of this algorithm is to remove the need for searching through multiple storefiles to answer a read request. The algorithm achieves this goal simply by preventing updated values of a given row from ending up across multiple storefiles in the first place. The overall I/O load of leveled compactions is significantly larger than standard compactions, however, the compactions themselves are small and quick, and so tend to be much less disruptive to the datastore's runtime performance over time. On the other hand, if the cluster is already I/O-constrained, or if the workload is very update-intensive (e.g., time series), then leveled compactions become counter-productive. Striped compactions[10], a variation of leveled compactions, have been prototyped for HBase as an improvement over its current algorithm. Yet another variation is implemented in bLSM [15], which presents a solution for fully amortizing the cost of compactions into the workload by dynamically balancing the rate at which the existing data is being compacted with the rate of incoming updates. In our approach, we take the compaction approach itself as a black box. In fact, all but the incremental warm-up approach do not care at all what is the actual content of the storefiles. The incremental warmup approach needs rows to be sorted in key order, but is also independent of the compaction algorithm.

Other data structures that perform periodic data maintenance operations in the same vein as the LSMT include R-trees [11] and differential files [16]. As with LSMT datastores, updates are initially written to some short-term storage layer, and subsequently consolidated into the underlying long-term storage layer via periodic merge operations, thus bridging the gap between OLTP and OLAP functionality. SAP HANA [17] is a major in-memory database that falls in this category. A merge in HANA is a resource-intensive operation performed entirely in-memory. Thus, the server must have enough memory to simultaneously hold the current *and* compacted data. In principle, our incremental warmup algorithm offers the same performance benefits as a fully in-memory solution, while requiring half the memory.

Both computation offloading as well as smart cache management are well-known techniques in many distributed systems. But we are not aware of any other approach that considers offloading compactions with the aim of relieving the query processing server of the added CPU and memory load. However, the concept of separating different tasks that need to work on the same data is prevalent in replication-based approaches, which affords an opportunity to run different kinds of workloads simultaneously on different copies of the data. As long as potential data conflicts are efficiently handled, this has the advantage that the different workloads do not interfere with each other. For instance, in approaches that use primary copy replication, update transactions are executed on the primary site only, while the other copies are read-only. In the Ganymed system [14], for instance, the various read-only copies are used for various types of read-only queries, while the primary copy is dedicated to update transactions. In a similar spirit, we separate compactions from standard transaction processing to minimize interference of these two tasks.

[9]http://leveldb.googlecode.com/svn/trunk/doc/impl.html
[10]https://issues.apache.org/jira/browse/HBASE-7667

Techniques for the live migration of virtual machines, such as [4, 18], deal with transferring a machine's state and data to another and switching over to it without drastically affecting the workload being served. Similarly, techniques for live database migration deal with efficiently transferring the contents of the cache [6] and potentially the disk as well [8]. Thus, similar data transfer considerations arise as for compaction offloading. However, in these migration approaches, one generally does not need to consider the interference between two workloads (i.e., the query processing and the offloaded compaction, in our case).

9. CONCLUSIONS

In this paper, we took a fresh approach to compactions in HBase. Our primary goal was to eliminate the negative performance impacts of compactions under update-intensive OLTP workloads, particularly with regards to read performance. We proposed offloading major compactions from the region server to a dedicated compaction server. This allows us to fully utilize the region server's resources towards serving the actual workload. We also use the compaction server as a remote cache, since it already holds the freshly compacted data in its main memory. The region server fetches these blocks over the network rather than from its local disk. Finally, we proposed an efficient incremental warmup algorithm, which smoothly transitions from the current data in the region server's cache to the compacted data fetched from the remote cache. With YCSB and TPC-C, we showed that this last approach was able to eliminate virtually all compaction-related performance overheads. Finally, we demonstrated that our system can scale by adding more compaction servers as needed.

For future work, we would like to make the compaction manager more aware of the load balancing requirements of regions, region servers, and compaction servers. If one compaction server is assigned more regions that it can handle effectively, the compaction manager should re-balance regions accordingly among the available compaction servers, while taking into consideration their current respective loads.

10. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for useful feedback to improve this paper. This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ministère de l'Enseignement supérieur, Recherche, Science et Technologie, Québec, Canada (MESRST).

11. REFERENCES

[1] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind Facebook Messages: Using HBase at scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012.

[2] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[6] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[8] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, pages 301–312, 2011.

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

[10] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB*, 7(12):1259–1270, 2014.

[11] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *Data Engineering.*

[12] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[13] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[14] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *VLDB J.*, 17(4):657–682, 2008.

[15] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. SIGMOD, pages 217–228, 2012.

[16] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, Sept. 1976.

[17] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. SIGMOD, pages 731–742, 2012.

[18] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.