Magma: A High Data Density Storage Engine Used in Couchbase

Sarath Lakshman, Apaar Gupta, Rohan Suri, Scott Lashley, John Liang, Srinath Duvuru, Ravi Mayuram
Couchbase, Inc
{sarath,apaar.gupta,rohan.suri,scott.lashley,john.liang,srinath.duvuru,ravi}@couchbase.com

ABSTRACT

We present Magma, a write-optimized high data density key-value storage engine used in the Couchbase NoSQL distributed document database. Today's write-heavy data-intensive applications like ad-serving, internet-of-things, messaging, and online gaming, generate massive amounts of data. As a result, the requirement for storing and retrieving large volumes of data has grown rapidly. Distributed databases that can scale out horizontally by adding more nodes can be used to serve the requirements of these internet-scale applications. To maintain a reasonable cost of ownership, we need to improve storage efficiency in handling large data volumes per node, such that we don't have to rely on adding more nodes. Our current generation storage engine, Couchstore is based on a log-structured append-only copy-on-write B+Tree architecture. To make substantial improvements to support higher data density and write throughput, we needed a storage engine architecture that lowers write amplification and avoids compaction operations that rewrite the whole database files periodically.

We introduce Magma, a hybrid key-value storage engine that combines LSM Trees and a segmented log approach from log-structured file systems. We present a novel approach to performing garbage collection of stale document versions avoiding index lookup during log segment compaction. This is the key to achieving storage efficiency for Magma and eliminates the need for random I/Os during compaction. Magma offers significantly lower write amplification, scalable incremental compaction, and lower space amplification while not regressing the read amplification. Through the efficiency improvements, we improved the single machine data density supported by the Couchbase Server by 3.3x and lowered the memory requirement by 10x, thereby reducing the total cost of ownership up to 10x. Our evaluation results show that Magma outperforms Couchstore and RocksDB in write-heavy workloads.

PVLDB Reference Format:
Sarath Lakshman, Apaar Gupta, Rohan Suri, Scott Lashley, John Liang, Srinath Duvuru, and Ravi Mayuram. Magma: A High Data Density Storage Engine Used in Couchbase. PVLDB, 15(12): 3496-3508, 2022.
doi:10.14778/3554821.3554839

1 INTRODUCTION

Modern-day internet-scale interactive applications generate huge amounts of data through user engagements. These data-intensive applications like ad-serving, internet-of-things, messaging, and online gaming are real-time and write-heavy, requiring large storage capacity and high transaction throughput. As a result, distributed databases that can scale horizontally have become an integral part of the modern data infrastructure stack that needs to operate at scale. The rapid growth of data volumes due to the digital wave has introduced challenges from a manageability and storage cost perspective. These problems have only grown despite the cost of computing and storage hardware like memory and flash dropping because the cost reduction has not kept up with the growth of data. The high throughput and storage capacity can be achieved by scaling out the distributed database by adding more nodes. To maintain a reasonable cost of ownership, we need to improve storage efficiency in handling large data volumes per node, such that we don't have to rely on adding more nodes.

Under the hood, a single node of the distributed database depends on a persistent key-value storage engine for durable storage and retrieval of the database records. B+Trees [7] and Log structured merge trees [26] are two popular access methods for implementing persistent key-value storage engines. B+Tree is a read-optimized data structure while LSM Tree is write-optimized. Both of these data structures can be found in popular distributed databases like Couchbase, Cassandra, MongoDB, CockroachDB, etc. The efficiency and performance of I/O intensive index structures are essentially a balance among three properties. Write amplification, read amplification, and space amplification (RUM Conjecture) [4]. We cannot achieve write-optimized, read-optimized, and space-optimized persistent index structures all at the same time. Write amplification defines the ratio of the amount of data written to disk for every byte of write to the storage engine. Read amplification is the number of reads issued to the disk for every read operation of the storage engine. Space amplification is the ratio of the amount of data stored on a disk to the user input data size.

Key Challenges with High Data Density. We start by identifying the challenges faced by our append-only copy-on-write B+Tree based storage engine to sustain high write throughput with a large volume of data per node with a database size to memory ratio of 100x.

Slow Writes. Updates in a copy-on-write B+Tree are done as read-modify-write, requiring random read I/Os. As the density increases, reads incur large cache misses for the B+Tree pages. Keys are spread out in a large key range distribution, and hence larger B+Tree. The opportunity for deduplication before writing and amortization of page rewrites due to large batches reduces, thereby increasing the write amplification. Write latency increases and throughput drops.

Compaction Challenges. When the database becomes fragmented, a compaction operation needs to be performed to limit space amplification. Compaction performs a full database rewrite

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554839

by copying live documents to a new file and building the B+Tree indexes, taking time proportional to the database size. After copying, it has to run catchup to replay the extra changes that came in during the copying. This introduces high write amplification. Writes can only run at the speed of single-threaded compaction per DB file. Even though we have several DB files per node, as the density increases, the size of individual files increases. Full DB file rewrite and longer duration catchup with larger DB size are no longer scalable.

We present Magma, a hybrid write-optimized storage engine based on LSM Trees [26] and Log-structured storage [28] available in Couchbase 7.1. In this paper, we describe the following key contributions:

- Evaluation of copy-on-write B+ tree for high data density
- Design of a high data density storage engine that blends LSM Trees with Log-Structured storage to achieve low write amplification
- A novel method for garbage collecting the log-structured storage efficiently

This paper is organized into three parts. We initially discuss the background by providing details on B+Trees in the context of challenges faced by our existing storage engine Couchstore in Section 2. Section 3 and 4 discuss the Magma design and our contributions. Section 9 provides experimental evaluation results and discussion.

2 BACKGROUND

2.1 Couchbase Data Service

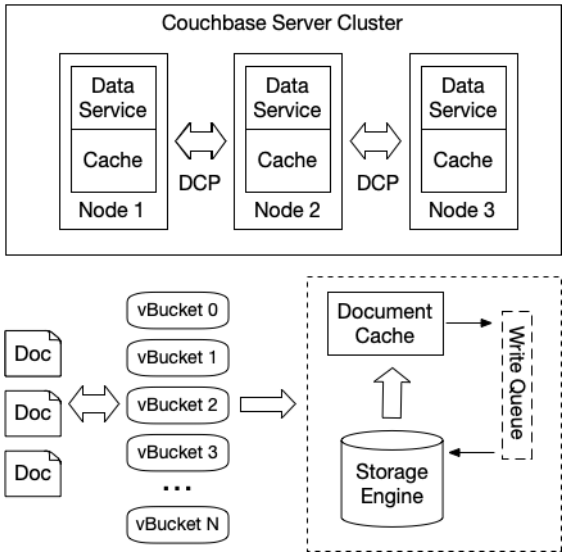

Figure 1: Couchbase server architecture

Couchbase distributed database has a microservices approach called multi-dimensional-scaling [6, 9] to horizontally scale all parts of the database. Data service is a distributed high-performance, replicated and elastic key-value document storage service that spans across several nodes as shown in Figure 1. In Couchbase terminology, a database is called a bucket. A bucket is internally split into 1024 logical partitions called vBuckets. A document is mapped to a vBucket by hashing its key. vBuckets are distributed among several nodes of the data service. As the number of nodes in the cluster increases, the number of vBuckets hosted per node decreases, thereby increasing the data density per vBucket. Documents belonging to a vBucket are stored in a single node key-value storage engine. The data service offers simple key-value lookup operations for the database. The secondary indexing, as well as range query use cases, are served through Global Secondary Index (GSI) service [20].

The primary APIs required by the vBucket key-value store engine are as follows:

(1) WriteDocs(DocsList: List<Key, Seqno, Value/Tombstone>)
(2) ReadDocs(KeyList: List<Key>)
(3) ChangeFeed(startSeqno : Seqno, endSeqno: Seqno, Callback<Key, Seqno, Value>)

The storage engine WriteDocs API accepts a batch of document mutations and persists it to the key-value store. The Data Service performs de-duplication on keys in a batch before writing them to a vbucket. Every document update has a monotonically increasing sequence number, seqno generated by the data service. ReadDocs performs key lookups for a batch of keys. The ChangeFeed API exposes a change log interface to the vBucket which returns the results as a stream of document mutations ordered by seqno. The changelog is the backend for several services within the database to subscribe and consume database document mutations. Couchbase Services like vBucket Replication, Global Secondary Index (GSI), and Analytics [19] consume the changelog via Database Change Protocol (DCP) [11] to create new indexes and keep up to date on the document mutations.

Managed Document Cache. The data service maintains an in-memory hash table-based document cache for each vBucket. The documents which are actively read and mutated are resident in the cache. Each document in the cache can be retrieved by the document key. If the document is not present in the cache, it retrieves the document through the key-value storage engine.

2.2 Couchstore Copy-On-Write B+Tree

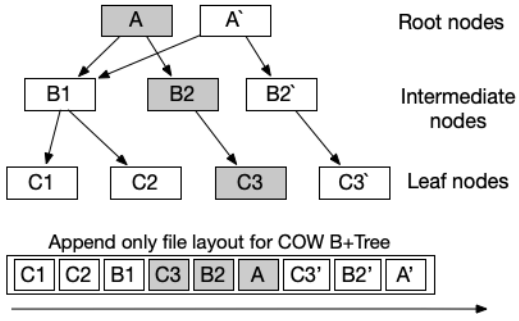

Figure 2: Copy-on-write B+Tree undergoing a modification

Couchstore [10] is the current generation storage engine of Couchbase Server for document storage. The overall architecture

inherits from the storage model of Apache CouchDB [2]. This storage engine is battle-hardened in production and has been serving Couchbase customers for almost 10 years. Couchstore is based on copy-on-write (COW) B+Tree and it follows a log-structured append-only storage model. Each vBucket maintains a couchstore file and stores the documents belonging to the vBucket. The file format consists of documents and B+Tree pages interleaved in the file. Each couchstore file maintains three B+Trees, a byKey index for accessing by key, bySeqno index for accessing by seqno, and a metadata B+Tree for storing vBucket statistics and metadata. To look up a document by key, byKey B+Tree is used to obtain the file offset of the document version, and a read is performed from the offset. Couchstore does not have a dedicated managed cache. Rather it depends on the file system buffer cache.

A copy-on-write B+Tree is an adaptation of B+Tree for the log-structured storage model. Compared to update in-place B+trees, the COW B+Trees can achieve higher write throughput as it performs writes in a sequential access pattern. B+Tree consists of intermediate pages and leaf pages. The leaf page consists of key-record pairs. Intermediate pages consist of key-value pairs with the value being the file offsets of pointing pages within the same file.

2.2.1 Write Operation. B+Tree modification involves a read-modify-write scheme for the B+Tree page. When a record needs to be added or removed to the COW B+Tree, it locates the leaf page where the record key belongs by traversing the tree from the root page and navigating through the intermediate pages. It makes a copy of the leaf page in memory and modifies the page to add or remove the record. The new version of the page is appended to the DB file. Now that the location of the leaf page has changed to a new offset, we need to update the intermediate node that points to the leaf node. Similarly, all the intermediate pages up to the root page need to be rewritten to update the new page locations. As shown in Figure 2, if a record is modified or added to page C3, it has to make the modification in C3 and create C3'. Similarly, the pointing parent pages including the intermediate page B2' and new page A' need to be written. The older version of the pages (C3, B2, A) becomes stale in the file as the current B+Tree points to the recently updated pages. In B+Trees, the unit of modification is a single page. Hence, even if a single record is added or removed, pages in the unit of disk block sizes need to be rewritten. Every leaf page modification results in multiple pages to be rewritten and this can lead to high write amplification.

2.2.2 Read Operation. Read operation in a COW B+Tree is similar to a traditional B+Tree.

2.2.3 Compaction Operations. Since we follow the append-only storage model for writes, every insert, update, and deletion operation in the B+Tree results in multiple page rewrites. Each modification operation generates a few new pages while making the older versions of the pages stale. These stale versions still are present in the DB file. As more data is written, the DB file grows in size. The B+Tree metadata maintains the size of the current live B+tree in the file. Once the stale data size grows above a fragmentation threshold compared to the DB file size, we perform compaction.

A compaction operation runs in a background thread. It obtains the current B+Tree root offset and opens a B+Tree iterator. A new DB file is opened and it performs a B+Tree bulk load operation to the new file, rebuilding the B+Tree. While the compaction is running, the writes continue to operate on the DB file. The compactor operates on a point-in-time snapshot of the B+Tree. After finishing the B+Tree bulk load, it runs a catchup phase to replay the new additions/deletions that happened to the B+Tree from the point-in-time version used by the compactor up to the latest B+Tree in the DB file. On completion of the catchup phase, the old DB file is removed and writers and readers switch to the new DB file. The space is reclaimed. Compaction is a single-threaded process that runs on a DB file.

2.3 Log-Structured Merge Tree

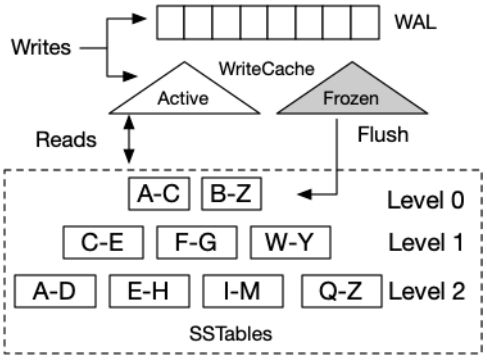

Figure 3: LSM Tree architecture

LSM Tree is a write-optimized persistent index data structure. LSM Trees achieve high write throughput by utilizing superior sequential write bandwidth of SSDs [18] and spinning disks compared to the random I/O access pattern. The large sequential writes are achieved by batching a large number of mutations in memory before writing the index structure. LSM Trees is a hierarchical data structure that consists of a memory component and multiple levels of persistent immutable index components. For the persistent components, it uses the append-only B+Tree index. The on-disk components are organized as levels with exponentially increasing sizes.

2.3.1 Write Operation. All writes in an LSM Tree are buffered in the in-memory component and they are also appended to a write-ahead log as shown in Figure 3. During crash recovery, the in-memory component can be recovered by sequentially scanning the write-ahead log. The in-memory component uses a sort ordered data structure providing fast lookups and range reads. Once the in-memory component reaches a threshold size limit, it is frozen and a new one is initialized for processing incoming writes. The records from the frozen in-memory component are converted into a B+Tree on a new file. This file is called a sorted strings table (SSTable).

2.3.2 Compaction Operation. As the in-memory components are flushed to the disk, more SSTable files are generated. For performing a lookup, it has to search SSTables in the most recent table first order until the key is found. The I/O and CPU cost becomes proportional

to the number of tables to be evaluated. A large number of tables also consume space as they may contain stale key-value pairs. To minimize the cost of reads as well as reduce space usage, we have to periodically merge SSTable files and reclaim space. This process is called compaction.

A level-based compaction strategy popularized by LevelDB [15, 24] is a common compaction strategy for achieving lower read amplification and space amplification. The LSM Tree is organized into multiple levels of exponentially increasing sizes with the smallest size at the top and the largest being the bottom level. Each level can have several SSTable files. The in-memory component is periodically flushed into level-0 as an SSTable file. Level-0 is a special level that accumulates new data. It can have multiple SSTables with overlapping key ranges. All other higher levels have non-overlapping key ranges between the SSTables in the level. Each non-level-0 level has a contiguous key range. When level-0 reaches a size threshold, the SSTable files are picked and merged with sstables from the level-1 and the overlapping key range from the level-1 is replaced with new SSTable files. This involves a k-way merge sort between the source SSTable files. A similar process is followed to manage the size of each level according to the size threshold.

Due to the compactions that periodically rewrite the data, LSM Trees can incur high write amplification. For an LSM Tree with a 10x level multiplier, each level except the level-0 contributes a write amplification of 10. When data is flushed to level-0 from the write cache, it contributes to a write amplification of 1. Similarly, the write-ahead log also contributes a write amplification of 1. Hence, for an LSM Tree with 5 levels, the worst-case write amplification can go up to 42. For skewed workloads, observed write amplification will be lower than the worst-case amplification.

2.3.3 Read Operation. The levels in the LSM Tree hold data in the order of recency. The lower levels have the most recent data. When a lookup operation needs to be performed, the read starts looking up the key from the in-memory component. If it finds the key in the in-memory component, it can return immediately as it contains the most recent version of the record. Otherwise, it keeps looking for the key in each next higher level. It has to go through every SSTable in the level-0 and one SSTable each from each of the other levels. Each SSTable has an immutable B+Tree incurring $log_B(n)$ lookup cost (read amplification) where n is the number of items in the SSTable. This can be expensive in terms of CPU andI/O operations.

To optimize the lookup, LSM Trees generally maintain a bloom filter per SSTable with high accuracy. This avoids I/O reads from SSTables which do not have the key. Using a bloom-filter with high accuracy, it can service the lookup operation using a single SSTable or a single B+Tree. This makes the cost of lookup similar to that of a traditional B+Tree.

3 MAGMA DESIGN

Magma is the next-generation single node document storage engine of Couchbase Server designed for improving write performance and data density per node. The data gathered from our customers with large data density use cases reveal that average document sizes range from 1 KB to 16 KB and our design optimizes for taking advantage of this. The core idea is to separate index and document data to minimize write amplification. Magma also builds a scalable and incremental compaction method to maintain stable space amplification.

In this section, we describe the goals, system architecture, and design of the storage engine.

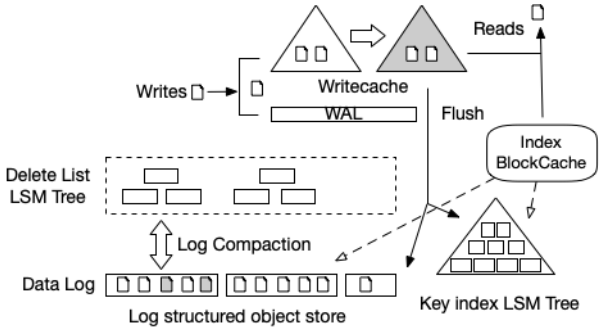

Figure 4: Magma storage engine

3.1 Design Goals

The design goals for the Magma storage engine are as follows:

Minimize Write Amplification. To achieve high transaction write throughput, write amplification needs to be lowered as the I/O bandwidth available on the storage devices is limited. Reducing the write amplification is also crucial to extend the lifetime of SSDs as the underlying flash media has a limited number of write-erase cycles [25].

Scalable Concurrent Compactions. As we observed with Couchstore, running a database level full compaction can be expensive and cause second-order effects like throttling the writers and resulting in lower write throughput. Having the ability to run multi-threaded smaller concurrent compactions that can incrementally perform space reclamation, is required for managing a large density database.

Optimize for SSDs. We need to leverage sequential read and write I/O access patterns [3] to utilize the full bandwidth of the SSDs. Random I/O should be only incurred during point lookup operations. The storage engine also needs to leverage I/O concurrency to utilize the higher IOPS offered by modern fast NVMe SSDs.

Low Memory Footprint. To support large database sizes per node, the database must optimize for a small memory footprint. The opportunity for read caching and write caching reduces significantly with a larger density of data.

3.2 Architectural Components

The Magma key-value storage engine consists of the following key components as shown in Figure 4.

Write-Cache. A write-cache is an in-memory component used to buffer key-value pairs and provide large sequential writes to the persistent storage. The write-cache is also used during lookup for key-value pairs. It is implemented using a lock-free skiplist [27][20][30]. Fixed memory is configured for the write-cache and it is internally split into two skiplists as active and immutable. When

it reaches the configured memory limit, it is flushed to the key index and log-structured object storage on the SSD.

Write-Ahead Log (WAL). A write-ahead log is an append-only log where the incoming key-value pair writes are initially written to provide durability. Writes are initially buffered in the write-cache and also written to the write-ahead log. The write API returns only after issuing an fsync on the write-ahead log file. Periodically, the space used by the write-ahead log is reclaimed when the write-cache is flushed.

LSM Tree Index. An index for the documents stored on a log-structured object store is organized as an LSM Tree. The LSM Tree index stores document key, document seqno, and size metadata as key-value pairs. For document read operations, the LSM Tree is initially looked up to obtain the document seqno which is used to read the document value from the log-structured object storage. The LSM Tree index maintains bloom filters with an accuracy of 99% to optimize the lookup I/O.

Log Structured Object Store. The log-structured object store provides persistent storage for the documents by organizing them on an append-only segmented log. The log-structured store maintains an index that allows querying of a document by seqno. The log-structured store also allows range query by seqno essentially providing a changelog for the document database.

Index Block Cache. A read cache is maintained in RAM for caching the recently read index blocks from the LSM Tree as well as the log-structured object store. Least Recently Used (LRU) [29] eviction policy is used to manage the cache. This cache does not keep the actual document data, but only the index blocks used to locate the documents on log-structured storage. Couchbase server maintains a vBucket level document cache. This object-level managed cache is more efficient than the block-level cache for document objects. The unit of caching in a block cache is of size, 4 KB. The document cache can perform the caching at a single document level and avoid wasting memory for keeping cold documents grouped in a physical block on the SSD.

3.3 Separation of Index and Data Organization

Persistent key-value index data structures like B+Trees and LSM Trees organize key-value pairs in sorted order. The ordering arrangement allows fast lookup by providing a bounded number of disk reads per lookup in the order of $log_B(n)$. For maintaining the sort order, the index data structure has to perform frequent rearrangement of the internal data structures on the disk. For B+Trees, it comes in the form of page splits and merges, along with multiple page rewrites whenever a record is added. LSM Trees require frequent level merges or compaction operations to limit the cost of lookup and range query operations. Records are initially written to the level-0 of an LSM Tree. Each level of the LSM Tree is arranged in exponentially increasing sizes. During compactions, records are rewritten several times as they moved from lower levels to higher levels of the tree. For an LSM Tree with a size multiplier of 10 and 5 levels, it may be rewritten up to a worst-case of 42 times. This is very high write amplification. If we store the large document records in an LSM Tree, the rewrites through the levels as part of compactions would be significantly wasting the SSD bandwidth. The write amplification incurred is the cost of maintaining sorted ordering. If data can be stored separately in a different log-structured storage [22] organization we can avoid the high write amplification. We still need an index that can provide quick access to the document stored on the log-structured storage. Based on this observation, we organize our index data structure separate from the document data storage.

We use LSM Tree as our persistent index. The index consists of records with key, document seqno, and document size. The seqno acts as a record locator for fetching the document from the log-structured object store. Considering an example of 20 byte key and 1 KB document size, the key is only less than 2% of the document size. By separating the index structure from data storage, the high write amplification cost of LSM Tree index is only applicable to a small fraction of the database. We leverage key prefix compression to further reduce the size of the keys stored in the index. A compact index structure is beneficial for minimizing the number of I/Os required to serve document read operations as a significant part of the index could be cached in the index block cache.

Wisckey. The idea of separating key and value for LSM Tree based key-value store has been proposed by Wisckey [23]. Wisckey maintains a sequential log for storing values and puts a direct file offset pointer in the LSM Tree. This approach requires index lookup to be performed during the log cleaning for every record to validate/invalidate a live record. In a limited memory setup like 100x database size to memory ratio, most reads will incur cache miss and every lookup can lead to multiple random I/Os slowing down the log cleaning process. The read I/O and CPU required to perform a lookup during log cleaning can be expensive. Magma takes a different approach leveraging sequential I/O access patterns by avoiding the index lookup during garbage collection. The design tradeoffs of Magma also allow for the implementation of an efficient changelog for the database.

3.4 Read and Write Operations

The interaction among various components in the storage engine design can be understood by illustrating the read and the write operation paths in the storage engine.

3.4.1 Lookup Operation. A key lookup operation first looks up into the active and immutable write-cache component in memory. If the key does not exist, then it proceeds to search in the persistent LSM Tree key Index in order to obtain the document seqno. To locate the key, LSM Tree identifies the candidate SSTables having a key range that overlaps with the key. It uses the bloom filter to probabilistically eliminate all SSTables which does not have the key present. Once an SSTable is identified, it reads through the B+Tree pages to read the record for the key. While reading the B+Tree pages from the SSD, it makes use of the Index Block Cache to eliminate the read I/O operation. If the key cannot be found, it returns not-found. Otherwise, it uses the document seqno to locate the document value from the log-structured object store. For searching within the log-structured object store, it identifies the log segment with the overlapping seqno range. Similar to the SSTable, a B+Tree index is used to retrieve the location of the data block where the document is present. Whenever an index block I/O operation is performed, the index block is inserted into the Index

Block Cache and a non-recent block is evicted from the cache if it is full.

3.4.2 Write Operation. A document write or mutation operation can be either insert, update or delete for a key. Each document mutation has a monotonically increasing sequence number. The document mutation is initially inserted into the active write-cache. Within the write-cache, it maintains two skiplists. One indexed by key and the other by seqno. The document mutation is also written to the write-ahead log for durability. The writer thread returns with a success status after this step. When the write-cache accumulates a sufficient number of document mutations and exceeds the configured memory limit, the write-cache is flushed to the persistent storage. The in-memory key-value pairs are converted into key index SSTable components for the LSM Tree and also documents are appended to the tail log-segment of the log-structured object store. After this step, the write-cache memory is freed. This is processed by a background flusher thread. The procedure for removing obsolete document versions from the log is described in section 4.

3.4.3 Changelog. A changelog read operation reads a stream of document versions starting from startSeqno to endSeqno. The operation starts by locating the starting log segment in the log-structured object store where the log-segment seqno range overlaps with the startSeqno. The index B+Tree within the log segment is used to locate the data block where the document with seqno >= startSeqno is present. The contiguous data blocks from the location are read sequentially until the end-of-file of the log-segment or until the endSeqno for the document is reached. Once it reaches the end-of-file, the subsequent log segments that fall within the requested sequence number range are read. Once we have finished going over the on-disk log segments, the in-memory write-cache is used to retrieve the rest of the document versions up till the endSeqno. The write-cache has a skiplist data structure which provides sort ordered documents by seqno.

3.5 Log Structured Object Store

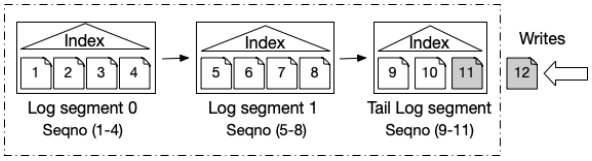

Figure 5: Log segment storage

Magma uses the segmented log concept from log-structured file systems [28][21] to build our append-only log-structured object store. The log-structured object store consists of several log segment files organized as a sequentially growing log with a tail log segment accepting the incoming writes. Each log segment maintains a COW B+Tree index to locate the documents by seqno. When the write-cache reaches the configured memory limit, the background flusher thread appends the document mutations to the tail log segment. Appended document mutations are organized as 4KB blocks. Each document mutation has a unique seqno. Each data block offset is added to the B+Tree index as part of the flush operation. When the size of the tail log segment reaches a segment log size limit, it is made immutable and a new empty tail log segment file is created. The tail log segment size limit is derived based on a percentage of the current total database size. The size of the log segment is the minimum unit size for performing log compaction.

As the document versions are appended in seqno order, the index COW B+Tree does not have to perform expensive read-modify-write of random pages. The writing log segment maintains a tail page per B+Tree level in memory and data block offsets are added as they arrive. The B+Tree pages are appended to the log segment file when the page becomes full or once all the document versions are appended from a write-cache flush session. The B+Tree write amplification is minimal due to the monotonic order of insertions. The B+Tree mainly consists of 8-byte seqnos and data block offsets. The index size is extremely compact (<0.4%) compared to the data stored on the log segment. This makes it easily cachable in limited memory. The log-segment storage maintains an in-memory array of log segments sorted by segment start seqno in memory. The candidate segment for any document lookup by seqno can be identified by searching in this list.

We preallocate the log segment files to minimize the file system metadata overheads. This also allows the physical layout on the SSD media to be contiguous apart from logically contiguous from the file system point of view.

The log can contain multiple document versions with the same key. When a new version is appended, that makes the older versions of the same document logically stale. When a key lookup is performed in the key index LSM Tree, it returns the latest version seqno for the document. Hence, the read operations being navigated through the key index, always end up reading the latest document version. In the next section, we discuss how to garbage collect stale document versions for reclaiming space.

Compression. Magma uses block-level compression using the LZ4 [1] algorithm. Due to block-level compression, we observe a significant size reduction for the same dataset compared to Couchstore. Couchstore uses document-level compression. The older log segments tend to have colder documents compared to the recent log segments. This provides an opportunity to use compression schemes with better compression ratios for cold log segments. Periodically, the blocks within the cold log segments can be recompressed with a higher compression scheme like ZStandard [13] in the background to save space.

4 GARBAGE COLLECTION

In a log-structured append-only storage model, when a record is deleted or a new version of the record is updated, the stale records are not immediately removed from the storage. A separate log cleaning or compaction process is used to reclaim space. As the Magma key-value store accepts update or delete operations, the prior versions of the documents stored on the log-structured object store become logically stale. Over a period of time, the stale document versions are accumulated in the log and they will consume a considerable amount of space. The newer versions of the documents are appended towards the tail of the log while invalidating older versions that get accumulated towards the head of the log. To limit

space usage and maintain stable space amplification, we have to perform periodic log cleaning.

To reclaim space, we need to identify the percentage of data consumed by stale documents in the log-structured store. We define fragmentation as the ratio of the total size of stale documents to the space consumed. A compaction operation needs to be initiated once the fragmentation of the log-structured store exceeds a configured threshold. The compaction operation has to pick a few log segments, copy all the live document versions to a new log-segment file and replace the candidate log-segment files. This essentially means we need to devise a method to estimate the fragmentation in the log segments and also check the validity of the document versions in the log segment during log compaction. The straightforward technique to check whether a document version is live is to lookup in the key index LSM Tree using the document key. If the retrieved seqno matches the current version seqno, it is a live document and it needs to be retained during compaction. The lookup operation for every document version during compaction can be an expensive process as it involves random I/O in the key index LSM Tree as well as the CPU cost for evaluating SSTables and bloom filters. When we have sufficient enough memory to cache the whole key index LSM Tree, the random I/O can be avoided. But, for a high data density use case with very low memory, often we cannot cache the key index in memory. The random I/Os can significantly consume the read bandwidth, otherwise available for key-value store read operations.

We introduce a novel approach for accurately estimating the fragmentation in the log-structured object store as well as performing garbage collection efficiently. The key idea is to maintain a logically sorted delete list of stale document seqnos and their size per log segment. If we take the sum of all sizes from the delete list, we can calculate the garbage size per log segment and thereby calculate the fragmentation. By triggering compactions when the fragmentation reaches a threshold of 50%, we can maintain a space amplification of 2 and low write amplification. If we pick log segments with higher fragmentation than 50%, the write amplification will be even lower. If we assume that we have a sorted delete list available per log-segment, the compaction process can perform a streaming sort-merge operation between the delete list and the log-segment. The seqnos which are both present in the delete list and log-segment get canceled out and discarded, while other live document versions are retained. For minimizing write amplification, the compaction process must pick the log segment with the highest fragmentation.

Now let's discuss how to generate the stale document seqnos list without performing an expensive key index LSM Tree lookup operation. The key index LSM Tree performs periodic compaction operations to maintain the shape of the tree to minimize read amplification and space amplification. As part of the process, multiple SSTables are merged into new SSTables and older versions or deleted key-value pairs are discarded as part of key version deduplication. We can use this deduplication step in the key index LSM Tree to generate the stale document seqno list. We implement a callback function in the key index compaction for the discarded documents. The callback function receives the seqno and size of the discarded versions and it can be used to populate the stale documents seqnos list for the log-structured store. The stale seqnos list

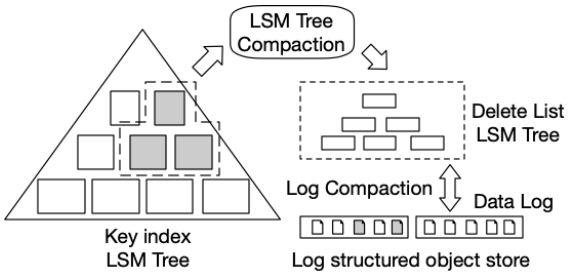

Figure 6: Delete propagation from LSM Tree to Log Structured Object Store

is populated asynchronously as the compaction happens on the key index LSM Tree. The sequence of delete list propagation operations is shown in Figure 6

Maintaining an in-memory list of stale seqnos per log segment can be expensive. A bitmap with one bit flag per document can be maintained to keep the list of stale seqnos in-memory with lower memory overhead. Even then, the memory requirement and the overhead to persist them consistently can be significant for a high data density key-value storage. Magma implements the persistent delete seqnos list by using an LSM Tree. The key-value pairs in this delete list LSM Tree are stale seqno and size. The size information allows computing the fragmentation efficiently. The extra space used for maintaining the delete list LSM Tree is tiny as it contains only an 8-byte seqno and 4-byte size. Since the size of the delete list data structures is small, the overhead of the LSM Tree in terms of write amplification as well as space usage is only a small fraction of the overall disk usage and bandwidth consumed.

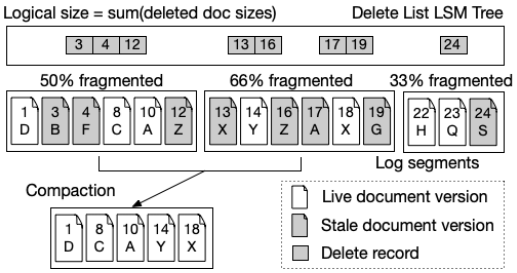

Figure 7: Log segment garbage collection

We arrange the delete list LSM Tree on top of the log-structured storage log segments to form a master LSM Tree with a few tweaks as shown in Figure 7. Instead of using the SSTable file sizes for the LSM Tree compaction level size thresholds, we compute a logical size using the document size information encoded in the delete list LSM Tree records. Each SSTable now has a logical size which is the sum of all document sizes encoded in the delete seqno records stored in the SSTable. This logical size is used for computing level size and also triggers compactions based on size thresholds. This modification normalizes the size units of log-structured object stores and the delete list LSM tree. The delete list LSM Tree level size targets

are now derived based on the total file size of the log-structured store.

The sum of all record values in the delete list LSM Tree provides an estimate of the total garbage size in the log-structured storage. We implement a fragmentation threshold above which compaction is triggered to rewrite log segments from the log-structured store to bring the fragmentation ratio below the threshold limit. We adapted the LSM Tree compaction procedure based on the level size threshold to perform log-segment compactions. The algorithm picks a few overlapping delete list SSTables that falls into the candidate log segment's seqno range and performs a merge operation. The candidate log segment is replaced with a new log segment discarding the stale seqnos merged from the delete list LSM Tree SSTables. Instead of picking a single log segment, we use additional heuristics to pick multiple log segments to avoid generating too many smaller log segments after rewrite. The default fragmentation threshold is configured to be 50% to minimize write amplification. By configuring a lower threshold, we can reduce our space amplification by trading off with higher write amplification.

In addition to the stale document versions, the index COW B+Tree rewrites during the write operations can also contribute to fragmentation within a log segment. When we operate with a tiny amount of memory for write-cache, every write-cache flush could potentially contain only very few documents. Hence, B+Tree blocks may need to be rewritten without enough fill factor generating stale blocks. We store this stale block's space usage within the log segment metadata. This estimate is also added to compute the fragmentation ratio along with the delete list. This LSM Tree arrangement between log segments and delete list also optimizes the changelog API. The changelog API can now run a merge iterator between the delete list and the data log. It can discard all the stale values while returning the stream of documents.

5 CRASH CONSISTENCY

A write ahead log (WAL) is used to ensure durability for the write operations by logging every mutation in the WAL while they are buffered in the LSM Tree in-memory write-cache. The Magma database maintains a metadata file to store a point in time snapshot of the live SSTable and the seqno based log-structured object store's segment files. Along with the file list, we also record the point in time WAL offset up to which the LSM Tree and Log-Structured Store have recorded the data in the snapshot metadata file. Periodically we run a checkpointing process which writes a new metadata file with the current live set of the data files from LSM Tree and Log-structured store. For the log segments, we also record the last-written consistent tail offset in the metadata file, in-order to restart writing after crash-recovery. Each checkpoint is a point-in-time snapshot of the database to which we can revert in case of a rollback. The database maintains periodic checkpoint snapshots up to 10 minutes old by default for allowing the data service to rollback in case of any failures. During crash recovery, the latest snapshot metadata file is read to reconstruct the LSM Tree levels and the corresponding SSTables. The same procedure is used for the log-structured object store as well. Using the WAL offset recorded in the snapshot metadata, we replay the document mutations that occurred after a metadata file was created from the WAL up to the end of the WAL into the write-cache. Since we use key index LSM Tree compaction to generate deleted seqnos list for the log-structured store, the point in time consistency of the key index LSM Tree and delete list LSM Tree is important. When compaction is performed in key index LSM Tree, the generated stale seqno list is added to the write-cache of the log-structured store delete list LSM Tree. Let's consider a case where the process crashes after persisting the snapshot metadata file for key index, while the generated seqnos from the compaction are not yet persisted in the delete list LSM Tree. This can result in inflight stale seqnos never getting garbage collected. To handle this case, we need to make sure that the delete list LSM Tree snapshot metadata is always persisted before persisting the key index snapshot metadata. This invariant ensures that we never lose stale seqnos metadata. Let's consider the case where we persist the seqnos in delete list LSM Tree, but the compaction SSTable changes on the key index LSM Tree are not persisted during a crash. The compaction will be rerun after the Magma recovery and the stale seqnos are regenerated. Adding the stale seqnos again to the delete list LSM Tree is an idempotent operation.

6 ASYNCHRONOUS I/O FOR READS

Modern fast NVMe SSDs have very high random read IOPS and bandwidth capacity. Scaling with very high IOPS can be challenging. For utilizing the available IOPS, we must queue a large number of requests to the SSDs to generate a queue depth of 64-128. Especially when we combine 3-4 NVMe SSDs, we have the capacity of 2M-3M IOPS available. We may have to generate a queue depth of 256 to 512. To generate such a high queue depth, we need large number of OS threads. A single thread can only generate one queue depth at a time if we use the blocking I/O model using pread64. To achieve high throughput with a minimal number of threads, we leverage asynchronous I/O through Linux kernel asynchronous I/O [5].

Magma implements a coroutine framework to dispatch concurrent reads within a single reader thread. When the reader thread receives a batch of keys to be fetched, we launch a limited number of coroutines equal to the SSD queue depth that we need to achieve. Each coroutine issues an async read request. The coroutine scheduler waits on the queued requests once each coroutine has at least queued one request. The coroutine-based framework allows to reuse the synchronous read APIs of the storage engine but internally swaps out pread based system calls with asynchronous system calls under the hood. This model allows magma to scale with the modern SSDs with a smaller number of threads. We demonstrate the read throughput scaling of Magma with the asynchronous I/O enabled in the evaluation section.

7 INTEGRATION WITH COUCHBASE SERVER

The Magma storage engine is designed to be a replacement for our current generation storage engine, Couchstore to service high data density workloads. We wanted to minimize changes in the Couchbase Data Service to simplify integration. The data service hosts many vBuckets within a server node. The database is internally sharded to improve the concurrency of read and write operations. Each shard hosts several vBuckets. The vBuckets follow a single writer, multiple readers concurrency model. Each magma storage

engine shard hosts multiple vBuckets while sharing a common write-ahead log. Each vBucket key-value store also requires a metadata store for storing local metadata for the database. We use a separate local metadata LSM Tree in addition to the key index to store the metadata.

7.1 Expiration of Documents

Couchbase Server supports document expiration which allows users to automatically remove a document after the expiration period elapses. To identify and remove expired documents, a periodic full key index scan has to be performed, which would be expensive in terms of I/O and CPU cost. Magma amortizes this cost by lazily evaluating and removing documents whenever compaction of the key index occurs. This can cause cases where some SSTables are not frequently compacted resulting in an accumulation of expired documents. To mitigate this, Magma maintains a histogram per SSTable with time intervals and count of future expirations. These histograms are periodically checked to estimate the percentage of keys that expired within an SSTable. If the proportion of expired items is beyond a configured threshold, compaction is triggered targeting those SSTables to remove expired documents.

7.2 Accurate Item Count Statistics

The Couchbase Data service provides accurate document count statistics for the databases to help customers with monitoring and troubleshooting. The item count statistics are maintained per vBucket by identifying whether each document mutation is either an insert or an update. For Couchstore, during the write operation, copy-on-write B+Tree has to perform a read-modify-write operation of the B+Tree pages. We can easily identify whether a document key already exists or not while performing the page modification. Items counts are incremented when a modifying document key is not already present.

For the Magma storage engine, we need to perform an extra key index lookup for every document mutation to identify whether a document mutation is an insert or update. For the insert, the bloom filter helps in identifying a non-existent key without incurring an I/O operation. By default, Couchbase Server integrates Magma with accurate item count. In the future, we plan to provide an optional feature to disable accurate item count to help customers choose better write performance tradeoffs by avoiding expensive read I/O during the writes. When item count is enabled, we piggyback on the same key index lookup operation to propagate stale seqnos to the log-structured store in real-time. We do not depend on the compaction operation.

8 FUTURE OPTIMIZATIONS

8.1 Skipping write-ahead log for large transactions

Write-ahead log contributes to write amplification of 1. For large writes, we can skip writing to the write-ahead log. Instead, along with write-cache buffering we can persist the document mutations to the log-structured store. We can replay from the log-structured store similar to the write-ahead log during the crash recovery operation. This optimization will allow write-heavy transactions to reduce the write amplification by 1 in addition to the existing significant improvements to lower write amplification.

8.2 Avoiding seqno based index lookup

The document read operation fetches the document seqno from the key index and then performs an index lookup into the log structured object-store to obtain the document block offset. The extra indirection using the seqno into the log-structured object store index can be avoided by keeping <logSegmentNo, blockOffset> in the key index records. This additional information can be populated during the write cache flush. These offsets stored in the key index records may become invalid as the compactions occur in the log structured object-store and log segments are replaced with new ones. The read operation can fallback to seqno based indirection if it cannot find the corresponding log segment in the object store. The invalid <logSegment, offset> location in key index records can be lazily updated with new locations during key index compactions by doing a lookup into the log structured object-store index.

9 EVALUATION

In this section, we evaluate Magma's performance and demonstrate the performance benefits of the hybrid design which combines an LSM Tree with Log-Structured Storage. During this evaluation, we focus on Magma's throughput, write and space amplification characteristics under varied workloads in a larger-than-memory setup. The experiments also compare Magma with RocksDB [14] and Couchbase's current storage engine, Couchstore.

9.1 Hardware setup

Our experiments run on a machine that has an Intel Xeon Gold 6230 2.10GHz processor with 80 CPUs, 125 GB of RAM, and an Intel DC P4610 SSD of 3.2TB capacity. The machine is running Linux 5.13 with XFS [17] as the file system.

9.2 Microbenchmarks

This section evaluates performance on a microbenchmark under load, update and read workloads on a single instance of the storage engine.

9.2.1 Experimental Setup. Evaluation is done on a randomly generated data set of 100M items with a key size of 40 bytes and a value size of 1024 bytes, making the total dataset size 100GB. For data 100x larger than the memory setup, we configure a 1GB memory quota for Magma and RocksDB. For read workload, Couchstore is also configured with a 1GB memory quota. For load and update workloads, Couchstore is configured with a 10GB memory quota as it has higher memory requirements during compaction. Magma and RocksDB are run with Direct I/O to get an accurate picture of their disk I/Os. They are configured with their block caches to compensate for the Linux page cache. Couchstore is run with Buffered I/O as it does not support Direct I/O and does not have its block cache. The available Linux page cache is limited to the memory quota using cgroups for Couchstore. The load and update workloads are executed using a single thread. Read workloads are executed by varying the number of threads.

RocksDB version 5.18.3 is used. Table 1 has the list of configurations set. Configurations that correspond in Magma are set to the same value for a fair comparison. RocksDB is set up with two column families. The byKey column family stores the (key, value) pair, and the bySeqno column family stores the (seqno, key) pair.

Table 1: RockDB configuration

Compression for levels > 0	LZ4
db_write_buffer_size	200MB
Block cache size	800MB
Block cache high_pri_pool_ratio	1
Bloom filter (only for byKey)	10 bits per key
max_background_jobs	5
Low/High priority thread pool size	4:1
level_compaction_dynamic_level_bytes	true
Partitioned index/filters	true
WriteOptions.sync	true

Couchstore uses Snappy compression [16] as it does not support LZ4. The fragmentation threshold for both Couchstore and Magma is set to 30%.

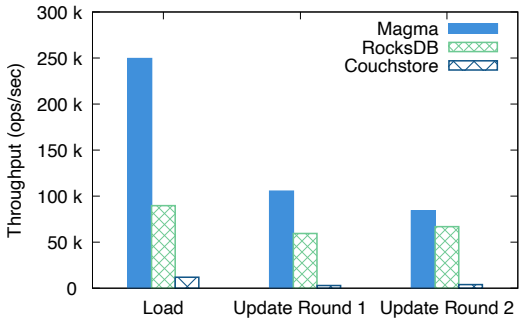

Figure 8: Write throughput for load and update workload

9.2.2 Load Workload. We generate 100M unique items and load them in random order with a batch size of 100 items. The results are presented in Figure 8 and 9. Magma is 2.78x faster and has 3.2x less write amplification than RocksDB. RocksDB is slower as writes are stalled for 50% of the time. The stalls happen when there are more than 20 files on the L0 level or when the aggregate level size exceeds their aggregate target size by 64GB or more and hence require a compaction [12]. The reason for frequent stalls is by storing keys and values together, it needs to rewrite more data during compaction which takes more time. This is also the cause for the higher write amplification. Magma's compactions are faster due to index and data separation as it rewrites only the keys. Since there are no updates in this workload, the delete-list LSM tree is empty and values are never compacted. Hence its write amplification is much lower.

Magma is 21x faster and has 3.3x less write amplification than Couchstore. Couchstore is slower and has higher write amplification because of multiple page rewrites and read I/Os as described before in section 2.2.

9.2.3 Update Workload. We generate 100M random items to update the initially loaded dataset. The write batch size is 100 items. We measure peak space amplification in addition to throughput and write amplification. Measuring space amplification makes the comparison fair as it can be traded off for write amplification [4]. It is also an important metric as it helps decide how much storage capacity needs to be provisioned. We repeat this workload twice. The first round evaluates the performance when the fragmentation is being built up. The second round evaluates the performance when the fragmentation has to be continuously maintained under limits. The results are presented in Figure 8 and 9.

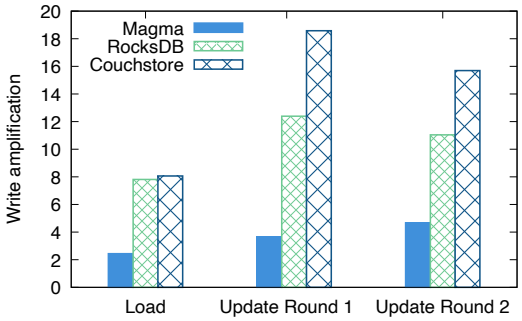

Figure 9: Write amplification for load and update workload

Round 1. Magma is 1.77x faster and has 3.38x less write amplification than RocksDB. The throughput for both RocksDB and Magma drops in comparison to the load workload. This is due to write stalls. RocksDB has an increase in write stalls and is stalled for 60% of the time. Magma also now stalls writes due to the delete-list LSM tree's L0 level exceeding either its size target or the number of files target. RocksDB has a peak space amplification of 2, slightly more than Magma's 1.93. With RocksDB's default level multiplier of 10, the expected peak space amplification is 1.111 (assuming 4 non-empty levels). However, its dynamic leveling feature decreases level multipliers dynamically to decrease write amplification trading off for increased space amplification. Magma's peak space amplification is 2 even when the delete-list LSM tree is only 30% fragmented. The actual space amplification is more than the expected 1.42 because of two reasons. First, the key index LSM tree is yet to compact and generate new deletes into the delete-list. Second, until a log segment compaction finishes, values in both the older log segment and the newer one will exist on the disk.

Magma is 36x faster and has 5x less write amplification than Couchstore. Couchstore's write amplification increases in comparison to load workload due to more compactions. More compactions are required since it gets fragmented faster than before. Apart from stale index pages, old values also contribute to fragmentation. Couchstore's peak space amplification is 3.05 which is 1.6x of Magma's 1.93. This is because of its much slower compaction process that rewrites the entire data set as described in section 2.2.

Round 2. Magma is 1.25x faster and has 2.36x less write amplification than RocksDB. It is 21x faster and has 3.37x less write amplification than Couchstore. Compared to the previous round, Magma has a higher write amplification because this round starts

with the fragmentation already built up triggering compactions much earlier. The throughput also drops due to an increase in write stalls. The peak space amplification for all three engines is similar to what it was in the previous round.

9.2.4 Read Workload. We randomly generate keys that exist in the dataset and issue point reads in the data initially generated by the load workload. We measure read I/O amplification and bytes per read which help in explaining read performance. These metrics are derived from /proc/diskstats. Results are presented with single and multiple reader threads in Figure 10. Every instance of this workload is run for 30 minutes.

Magma's read amplification is 1.21x less and reads 1.11x fewer bytes than RocksDB. Magma does lesser reads because it stores compressed blocks in block cache and hence can cache more of them. However, the read throughput is slightly less than RocksDB because of Magma's higher CPU usage.

Magma's read amplification is 2.06x less, reads 2.33x fewer bytes, and has a read throughput up to 1.76x higher than Couchstore. Couchstore does more reads due to its caching inefficiency. Its reliance on Linux page cache is not as efficient as having its own managed cache for two reasons. First, Couchstore cannot prioritize caching of index blocks over data blocks even though they tend to be accessed more. Second, by default, the kernel issues read aheads asynchronously which brings in more blocks than requested which could evict other useful blocks. Both these reasons lead to cache pollution and result in higher read amplification and thereby lower throughput.

Table 2: Metrics captured during read workload

Metric	Magma	RocksDB	Couchstore
Read amplification	1.79	2.18	3.7
Bytes per read	12.6k	14k	29.4k

Next, we demonstrate Magma's improvement in read performance while needing lesser threads when leveraging asynchronous I/O as described in section 6. In these experiments, the read batch size and queue depth for Magma's coroutine framework is 32. Results are presented in Figure 10.

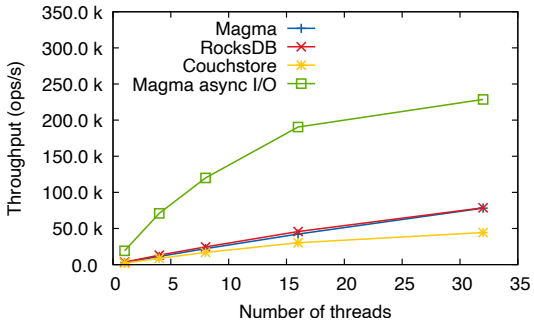

Figure 10: Read throughput with varying number of reader threads

With 32 threads, asynchronous I/O provides 2.92x higher throughput than synchronous I/O. Furthermore, to achieve similar throughput as synchronous I/O does with 32 threads, asynchronous I/O only requires 8 threads.

9.2.5 Various Item Sizes. We demonstrate performance on datasets having values of a particular size in the range 512B to 64KB. Every dataset is of 100GB size and is prepared by loading unique keys randomly.

We run an update workload that overwrites the entire data once randomly. A write batch size of 100KB is used for all value sizes. The results are presented in Figure 11. Magma continues to scale well as its write amplification stays low and reduces from 4.3 to 2.8. RocksDB doesn't scale well because its write amplification stays high and only reduces from 13.5 to 10.4. Couchstore shows some scaling as its write amplification significantly reduces from 41 to 2.5. The source of write amplification and its effect on throughput is the same as described in section 9.2.3.

We run a read workload as described in section 9.2.4 with 32 threads. The results are presented in Figure 12. All three storage engines show some scaling because, with increasing value size and fewer items, the total size of index blocks reduces. This improves cache hit ratios and reduces read amplification. The reason why Magma, RocksDB scale similarly and do so well whereas Couchstore does not is the same as described in section 9.2.4.

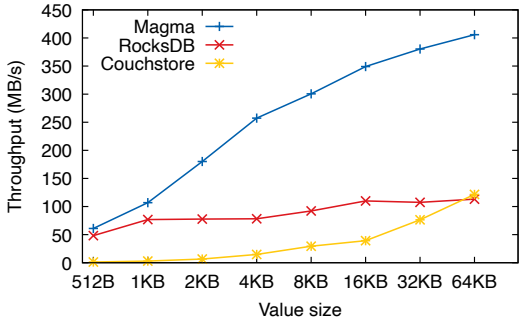

Figure 11: Write throughput for various item sizes

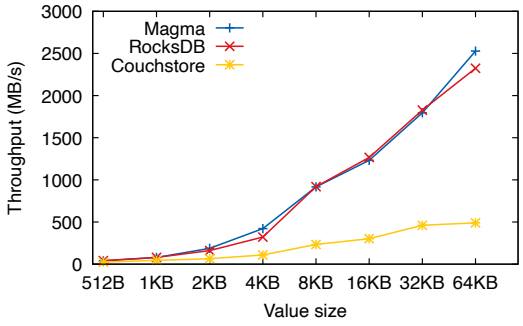

Figure 12: Read throughput for various item sizes

9.3 Full System YCSB Benchmarks

We next evaluate the performance of Magma integrated with the whole Couchbase Server stack. In this test, we aim to assess the performance of the storage engines integrated with Couchbase Server under real-world workloads and a large amount of data per node.

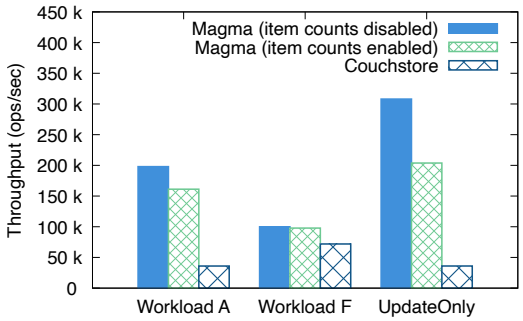

Figure 13: YCSB Write Heavy Tests

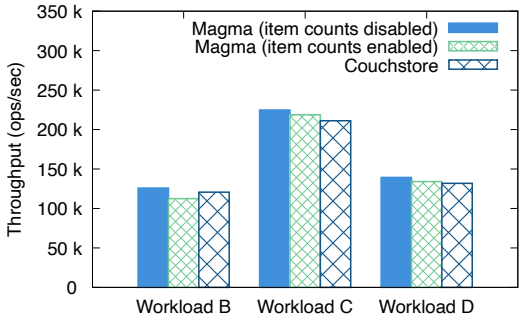

Figure 14: YCSB Read Heavy Tests

9.3.1 Experimental Setup. We evaluate the performance of Magma as a storage engine for Couchbase Server simulating a single node in an 8 node cluster by lowering the number of vBuckets to 64.

We compare Magma's performance using the YCSB workloads [8]. We chose YCSB Workload A, Workload B, Workload C, Workload D, Workload F, and an Update only workload. These workloads are used to compare Magma with item counts enabled, Magma with item counts disabled, and Couchbase's current storage engine Couchstore.

The cluster is loaded with 1 Billion records of size 1KB for a total of 1 Terabyte. Each record consists of a 24 bytes key and value composed of 10 fields of 100 bytes each. Since we are simulating a high data density scenario, the memory of the OS is restricted to 20GB by setting Linux kernel boot parameter mem and Couchbase's memory quota is set to 10GB. This configuration results in a 1% memory to data size ratio. We also set Couchbase Server's Reader and Writer thread settings to Disk I/O Optimized.

Before each workload, an update-only workload with zipfian distribution is run for 30 minutes. This allows for the storage engines to be evaluated while there is fragmentation and compaction operations are running in the background. After the update phase, the workload is run for 1 hour and monitored. We ensure the throughput of update and write tests is sustainable ie. disk size is not growing unbounded and compaction can keep up with fragmentation.

9.3.2 Write Heavy Workloads. We ran an Update Only workload, YCSB Workload A (50% read and 50% update workload), and Workload F (50% read and 50% read-modify-write workload) with zipfian key distribution. As shown in Figure 13, the throughput of Magma with item counts disabled (ie. Magma does not do a lookup on every set) is better than Magma with item counts enabled and significantly higher than Couchstore. The difference in throughput narrows as the proportion of reads increases in Workload A (50% reads) and Workload F (75% reads, 50% for the gets, and 25% as part of the read-modify-write operation); reads in all three storage engines have approximately the same cost as the key distribution is zipfian. Couchstore's update throughput is significantly lower due to compaction being unable to keep up with the rate of writes and the write operations are throttled to a rate at which disk size is stable. Couchstore's compaction is slow due to higher write amplification, compaction running on the entire vBucket database at once, and lack of concurrency. Magma without item counts is faster than with item counts since it avoids extra read I/Os for item count maintenance.

9.3.3 Read Heavy Workloads. YCSB Workload B is a 95% read and 5% update workload, Workload C is 100% read and Workload D is 95% read and 5% insert. In these read-heavy workloads, we observed similar performance in all three storage engines due to zipfian key distribution. Magma with item counts enabled was slightly slower than the other two since some of the read bandwidth was lost to the lookups performed before writing to maintain the item counts. The results of all three workloads can be seen in Figure 14.

10 CONCLUSION

In this paper, we presented the design of a single node key-value storage engine used in the Couchbase distributed database. We reviewed the challenges of supporting write-heavy workloads with large data density per node. We discussed how to combine LSM Tree index with a log structured storage and perform garbage collection efficiently. Finally, we compare and evaluate the performance of Magma through microbenchmarks and YCSB full system tests to demonstrate that it outperforms other engines in write-heavy workloads.

ACKNOWLEDGMENTS

We thank Ritesh Agarwal, Ankush Sharma, Ritam Sharma, Balakumaran Gopal, Bo-Chun Wang, Dave Rigby, Dave Finlay, Daniel Owen, Ben Huddleston, Jim Walker, Trond Norbye, Richard de Mellow, James Harrison, Paolo Cocchi, Sriram Melkote and Tai Tran for their support and contributions to the project. We also thank Jeelan Poola and Shivani Gupta for providing feedback on the paper.

REFERENCES

[1] [n.d.]. *LZ4 - Extremely fast compression.* Retrieved June 27, 2022 from https://lz4.github.io/lz4/

[2] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.".

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2018. *Operating systems: Three easy pieces.* Arpaci-Dusseau Books LLC Boston.

[4] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture.. In *EDBT*, Vol. 2016. 461–466.

[5] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*. 371–386.

[6] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have your data and query it too: From key-value caching to big data management. In *Proceedings of the 2016 International Conference on Management of Data*. 239–251.

[7] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[9] Couchbase. [n.d.]. *Couchbase Multi-Dimensional Scaling.* Retrieved March 1, 2022 from https://docs.couchbase.com/operator/current/concept-mds.html

[10] Couchbase. [n.d.]. *Couchstore.* Retrieved March 1, 2022 from https://github.com/couchbase/couchstore

[11] Couchbase. 2015. *Couchbase Database Change Protocol.* Retrieved March 1, 2022 from https://blog.couchbase.com/inside-couchbase-server-database-change-protocol-the-super-conductor-that-wires-couchbase-server

[12] Facebook. [n.d.]. *RocksDB write stalls.* Retrieved March 1, 2022 from https://github.com/facebook/rocksdb/wiki/Write-Stalls

[13] Facebook. [n.d.]. *Zstandard - Realtime data compression algorithm.* Retrieved June 27, 2022 from https://facebook.github.io/zstd/

[14] Facebook. 2013. *RocksDB.* Retrieved March 1, 2022 from http://rocksdb.org

[15] Sanjay Ghemawat and Jeff Dean. 2011. *LevelDB.* Retrieved March 1, 2022 from http://code.google.com/p/leveldb

[16] Google. [n.d.]. *Snappy, a fast compressor/decompressor.* Retrieved June 27, 2022 from https://github.com/google/snappy

[17] Christoph Hellwig. 2009. XFS: the big storage file system for Linux. *; login:: the magazine of USENIX & SAGE* 34, 5 (2009), 10–18.

[18] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. 1–9.

[19] Murtadha AI Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase analytics. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2275–2286. https://doi.org/10.14778/3352063.3352143

[20] Sarath Lakshman, Sriram Melkote, John Liang, and Ravi Mayuram. 2016. Nitro: a fast, scalable in-memory storage engine for nosql global secondary index. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1413–1424.

[21] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. {F2FS}: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.

[22] David Lomet. 1995. The case for log structuring in database systems. In *Int'l Workshop on High Performance Transaction Systems*. Citeseer.

[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[24] Chen Luo and Michael J. Carey. 2020. LSM-Based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (jan 2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[25] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid state drives.. In *FAST*, Vol. 12. 1–16.

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree).

[27] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[28] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[29] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2006. *Operating system concepts.* John Wiley & Sons.

[30] Håkan Sundell and Philippas Tsigas. 2005. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005), 609–627.