CapsuleDB: A DataCapsule Based Key-Value Store

William Mullen
UC Berkeley

Nivedha Krishnakumar
UC Berkeley

Brian Wu
UC Berkeley

Abstract

Paranoid Stateful Lambda (PSL) implement a stateful function-as-a-service framework built on the Global Data Plane that can unify edge and cloud computing resources through an in-memory key-value store (KVS). However, PSLs have several limitations: they are constrained by Intel SGX's memory limits, they have no mechanism to quickly find key-value pairs within a DataCapsule, and they are slow to recover. We present CapsuleDB, a DataCapsule-backed KVS that removes these limitations. We show that DataCapsules are uniquely suited to provide a level-tree based storage system, which we implement through the use of CapsuleBlocks and an index that tracks active data. Our results show that read/writes into CapsuleDB are performant even for large data sets, thereby removing the in-memory constraint imposed by Intel SGX enclaves.

1 Introduction

The explosion of wireless sensor devices along with the ability to use cloud services has led to a rise in distributed computing and a new paradigm of computing called the Internet of Things (IoT). However, there are many privacy, security, scalability, durability, and latency concerns. In particular, protecting data from potentially untrusted cloud providers has emerged as a difficult problem [23]. A user may not want to send their personal data to a service provider, and the provider likely would not want to send proprietary algorithms to the user's local device. For many IoT devices, there may not even be enough computation resources locally to complete complex tasks. One way to solve this problem is to leverage existing edge computing resources in a distributed fashion. Unfortunately, edge computing presents several challenges [17]. Rather than attempt to fix every problem, there are a number of new paradigms that natively address these concerns. One such system is the Global Data Plane (GDP) [13], a data-centric system that enables secure management (transport, storage) of data among clients on both the edge and in the cloud. This federated infrastructure enables the seamless access and management of computational resources through a structure called a DataCapsule.

However, while the GDP introduces a comprehensive communications layer, it does not natively contain any frameworks for building applications on top of it. The Paranoid Stateful Lambda (PSL) [4] project is a function-as-a-service (FaaS) framework that leverages the GDP's architecture to enable large-scale parallel, secure, and stateful computation among edge and cloud environments. PSLs use DataCapsules and Trusted Execution Environments (TEEs) [11] to provide integrity, confidentiality, and provenance to data. For a lambda to be stateful, it should be able to communicate with other lambdas to share and store communicated key-value pairs. These lambdas are also lightweight enough to run on edge devices in a secure environment while the GDP enables access to additional storage and computation resources. Using PSLs, functions can be run in a cloud environment without needing to trust the underlying hardware provider.

While the PSL project achieves our goal of secure computation on the edge, it does have several major limitations. Currently, key-value (KV) pairs are stored in-memory in a data structure called a memtable. As the size of the memtable grows, the memory footprint used by each enclave increases, eventually hitting the memory bottleneck imposed by Intel SGX [11]. Even if we used Intel SGX2 with its dynamic memory allocation, we would still eventually run out of space for large data sets. Hence, the scalability of PSL becomes an issue. Second, since PSL stores each KV pair directly into the DataCapsule, even if each lambda was not constrained by the size of the memtable it would be slow to read a value. This latency would only increase as the number of KV pairs increased, since the lambda may have to search the entire

capsule. Finally, when an enclave crashes, PSL replays all the past records in the DataCapsule to reconstruct the memtable, leading to long recovery times. To overcome the above issues, we propose a new key-value store.

In this paper we present CapsuleDB, a KVS inspired by level-tree databases that seeks to solve the aforementioned issues with PSL and is backed by DataCapsules. CapsuleDB exports a standard KVS interface with `put` and `get` functions. However, we designed a novel indexing and data management scheme that is DataCapsule native and takes advantage of a capsule's structure. The rest of this paper is organized as follows. Section 2 provides additional information on the GDP, PSL project, and level-tree databases. Section 3 introduces the design of CapsuleDB while Section 4 discusses the specifics of our implementation. Section 5 discusses the results of our tests. We then discuss related and future work in Sections 6 and 7 respectively. Finally, Section 8 concludes.

1.1 Metrics of Success

We use the following metrics to evaluate the success of the project:

1. Define a robust process for reading values that does not constrain worker nodes from operating.
2. Rewrite PSLs to use CapsuleDB rather than relying on the multicast tree for key-value pairs.
3. Match or improve current implementation's read / write latency.
4. Demonstrate faster recovery from failure than the current PSL recovery process.

2 Background

We begin with background on the GDP, DataCapsules, PSLs, and level databases.

2.1 Global Data Plane

The GDP [13] is a federated data-centric infrastructure network. There is no central authority. Instead, GDP servers route requests to specific services and clients using GDP names as identifiers. The basic unit of data in the GDP is a structure called a DataCapsule, a data-agnostic and cryptographically secure container for data. These capsules are replicated and distributed throughout the GDP, acting as storage, communication channels, transaction records, and entity identifiers. Together with the GDP infrastructure, they provide a robust distributed computing architecture, able to draw on computing resources regardless of physical location. Because each capsule is in a standardized format, though they may each contain different types of data, they are universally routable and manageable. Through additional locality optimizations, clients are also able to satisfy performance requirements by pulling DataCapsule replicas close to them. This paper will not discuss details of or improvements to the GDP's implementation (see Fall 2021 CS 262 projects 15 and 17), but we will discuss the structure of a DataCapsule.

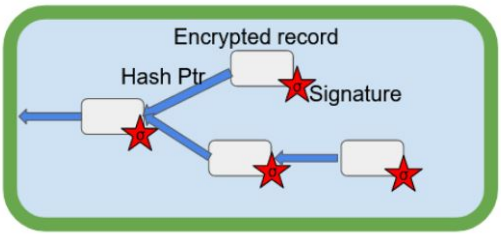

Figure 1: A basic DataCapsule with hash pointers and signatures.

2.2 DataCapsules

A DataCapsule is a cryptographically hardened append-only data structure that is globally addressable in the GDP framework and acts as the atomic unit of data transfer. Each capsule is made up of a metadata wrapper and a series of records. In their most basic form, these records contain arbitrary data, a hash over the data, and a series of hash pointers. Each record must also specifically contain a signed hash pointer of the previous record, thus creating a well-ordered chain of hashes as shown in figure 1. The first record in the capsule is a special metadata record that uniquely defines the DataCapsule itself. It contains ownership information, a public key for signing records, and other application specific information. The GDP name of this capsule is defined as the hash of this metadata record. In turn, each record in the capsule can also be uniquely identified by their hash. DataCapsules are also extremely flexible in the type and amount of data each can carry, constrained only by the available resources.

The structure of DataCapsules enforce several powerful security guarantees. Records are immutable, well-ordered, and cryptographically signed. In addition, they contain hashes over their data as well as hash pointers to previous records. Thus, it is straightforward to quickly verify the integrity of records and the overall capsule. This structure also establishes a chain of provenance, storing the data's history for the life of the capsule. For use cases that require even stricter security, proofs can be returned that verify a DataCapsule's validity. Since each

record is data agnostic, it is also straightforward to encrypt data, providing confidentiality without disrupting the capsule's structure. Finally, because DataCapsules have the owner's public key embedded into the metadata, only authorized users are able to write to it. Thus, the structure of DataCapsules provides strong integrity, confidentiality, authenticity, and provenance guarantees without needing to rely on third-party infrastructure.

2.3 Paranoid Stateful Lambda

The Paranoid Stateful Lambda project (PSL) is an implementation of a function-as-a-service (FaaS) framework similar to AWS Lambda [1] and Google Cloud Functions [2], but built on the GDP. However, in contrast to the aforementioned services PSLs can maintain state between remote nodes. Each worker also maintains an internal eventually consistent and durable KVS, called a memtable. These memtables enable the statefulness of PSLs. Through a system called the Secure Concurrency Layer (SCL), workers can multicast new KV pairs to other workers.

However, PSLs may be operating on sensitive data. For example, in an AI use case a user may not want their data uploaded to an third-party's server, but the service provider may not want to put their proprietary model directly onto a user device for fear of intellectual property loss. To overcome this problem, PSL executes almost entirely in TEEs, specifically Intel SGX [11] enclaves. These TEEs allow for service providers and users to attest the enclave as well as that the code inside is being executed as expected. Furthermore, enclaves protect the data they contain from the host OS and applications. As a result, PSLs can safely run on the edge or in the cloud, regardless of who owns the underlying hardware, without leaking any information. The only portion of the system that is not inside the enclave is the DataCapsule itself, as its structure natively protects its data as discussed above.

Finally, the SCL also is responsible for connecting the various components of the PSL system together. In addition to the worker enclaves, there is also a coordinating enclave that takes requests from clients and manages the overall execution. There is also a key-management enclave that distributes identity information to remote enclaves and manages the attestation process. With all these components, PSLs provides a robust and secure FaaS framework that has a wide range of applications. However, as discussed in Section 1, CapsuleDB is needed to overcome several key limitations of the base system.

2.4 Log-Structured Merge-Trees

A log-structured merge-tree (LSM-tree) [14] is a data structure designed to quickly serve the most recent version of data while also preserving older versions. In its most basic form for KVS's, data is split into different levels labelled as level 0 (L0), level 1 (L1), level 2 (L2), and so on. Each level is larger than the previous, with L0 being the smallest and often kept in memory. Data enters the structure through L0, being first stored there. Once L0 is full, a process called compaction is triggered which moves data from L0 into L1. Any overlapping keys in L1 are replaced with the fresher data incoming from L0. This process then repeats down the tree until no levels are full or have overlapping keys. LSM trees are beneficial because they naturally sort data into more and less recently used groups. Fresher keys are kept at the top in L0 and can be quickly returned, whereas less often used keys are stored in larger levels that take longer to search. We chose LSM trees as the basis for the design of CapsuleDB because they are uniquely suited to match well with DataCapsules, as discussed in Section 3. A brief note on notation: various sources have discussed LSM trees using different terminology for the same design. In this paper, we consider "lower levels" to have higher numbers. Thus, L2 is lower than L1.

3 CapsuleDB

CapsuleDB is a key-value store inspired by LSM trees and backed by DataCapsules. It is built to specifically take advantage of the properties of DataCapsules to mitigate the limitations of PSLs as discussed in Section 1.

3.1 Interface

CapsuleDB exports a simple interface for developers already familiar with other KVS's. `put` accepts a KV pair along with a timestamp to be stored in the database. `get` takes a key and returns the associated value and timestamp of when the pair was originally inserted. This interface abstracts the storage process and DataCapsule operations from clients. Currently, PSL interacts with this interface through the SCL, though these functions were also designed for CapsuleDB to be used as a standalone system.

3.2 Structure

CapsuleDB has two main data structures, CapsuleBlocks and indices, as well as its own memtable. CapsuleBlocks are groups of keys, each of which represents a single record in the DataCapsule backing the database. In L0, each CapsuleBlock represents a memtable that has been

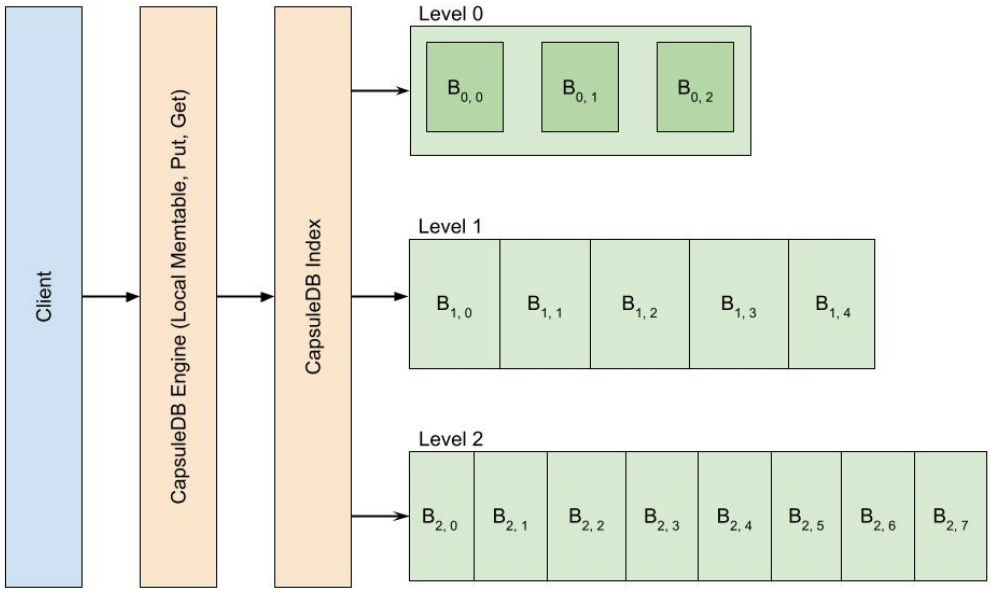

Figure 2: The structure of CapsuleDB. The index contains block ranges and hashes that help quickly retrieve blocks from the DataCapsule which may contain the requested key.

filled and marked immutable. Blocks in lower levels each contain a sorted run of keys, such that the keys in the level are monotonically increasing.

The index manages which CapsuleBlocks are in each level, the hashes of each block, and which blocks contain active data. The index also acts as a checkpoint system for CapsuleDB, as it too is stored in the DataCapsule. While the main purpose of this is to ensure CapsuleDB can quickly restore service after a failure, it has the added benefit that old copies of the indices serve as snapshots of the database over the lifetime of its operation. We now provide further detail on the different components of CapsuleDB, as well as the reading and writing processes.

Together, these structures create the system shown in figure 2. The client on the left of the diagram only interacts with the engine through the aforementioned `put` and `get` functions. The request is processed through the memtable with the index handling any interactions with the levels.

3.3 Indexing the Blocks

The index is at the core of CapsuleDB and serves several critical roles. Primarily, it tracks the hashes of active CapsuleBlocks to quickly lookup keys. It does so by organizing blocks into levels. Each level beyond L0 sorts all of its contained keys in ascending order. This run of keys is broken into chunks and stored in CapsuleBlocks, which the index tracks via the hash of the associated record. L0 simply contains immutable memtable copies, and thus must be searched linearly. However, it is kept relatively small for this specific reason. Whenever a block is added, removed, or modified, the hash mapping is updated in the index. When compaction occurs, the index updates which levels the moved CapsuleBlocks are now associated with. In this way, CapsuleDB can always quickly find the most recently updated CapsuleBlock that may have a requested key.

At first glance, it may seem strange that the list of hashes must be updated for any operation that modifies a CapsuleBlock. However, recall that DataCapsules are an append-only data structure with immutable records. If a CapsuleBlock needs to be modified, a new record containing the block must be appended to the end of the chain. This is the second major function of the index, it determines which blocks have active data. Since nothing can ever be deleted from a DataCapsule, old data that no longer holds the correct KVS state is still present. By updating the record hashes in the index, CapsuleDB effectively removes any references to those old blocks. Although still technically accessible by scanning through the DataCapsule, a user does not have to worry about

accidentally retrieving stale data. The index acts as an effective source of truth on the most up to date CapsuleBlocks and, by extension, the KV pairs they contain.

Finally, the index also assists in failure recovery. Without CapsuleDB, a crashed PSL worker would have to scan back through the entire DataCapsule to bring their memtable up to date. While technically feasible, this is slow and impractical, especially for larger data sets. CapsuleDB solves this problem by acting as stable storage that can quickly retrieve requested values. However, what happens if CapsuleDB fails? The final major function of the index is that it protects against this very situation. Since the index is written out to the DataCapsule whenever it is modified, the CapsuleDB instance can be instantly restored simply by loading the most recent index. Then, only the records since the last index need to be played forward to restore the most recent KV pairs that were in CapsuleDB's memtable. However, without the index CapsuleDB would be forced to traverse through the DataCapsule in the same way as the PSL workers to manually retrieve the hash for each CapsuleBlock and restore the levels of the database. Thus, the index is an indispensable part of CapsuleDB's fast restoration process and provides critical recovery benefits.

3.4 Compaction

Compaction is critical to managing the data in any level-based system. CapsuleDB's compaction process uses key insights from the flush-then-compact strategy of SplinterDB [5] to limit write amplification. Recall each level has a maximum size; we say a level is full once the summed sizes of the CapsuleBlocks in that level meets or exceeds the level's maximum size. This triggers a compaction.

All writes to CapsuleDB are first stored in the in-memory memtable. Once the memtable fills, it is marked as immutable and written to L0 as a CapsuleBlock, also simultaneously appending the block as a record to the DataCapsule. Once L0 is full, compaction begins by sorting the keys in L0. They are then inserted into the correct locations in L1 such that after all the keys are inserted L1, is still a monotonically increasing run of keys. Any keys in L0 that are already present in L1 would replace that tuple in L1, since the L0 value and timestamp would be fresher. Finally, the CapsuleBlocks and their corresponding hashes are written out to the DataCapsule and updated in the index, marking the end of compaction.

Now consider a slightly more complex situation. When compaction starts after L0 is full, we also evaluate the size of L1. If the the size of L0 and L1 together would exceed the maximum size of L1, then CapsuleDB knows L1 would also need to be compacted into L2. Now, rather than flushing and compacting L0 into L1 and then the new L1 into L2, we first determine which blocks in L1 would be affected by the incoming L0 blocks. These blocks are first flushed from L1 and compacted into L2. After this, the new blocks from L0 would be flushed and compacted into the now smaller L1. Finally, the index is updated to reflect the changes made to the levels. If L2 would fill up as a result of L1's compaction, the same process would occur between L2 and L3, and so on.

Through this strategy, each level only goes through a single round of changes, substantially reducing write amplification. This is critical since any modification to a CapsuleBlock would result in a new record being appended to the DataCapsule, a relatively slow process. If we fully compacted a level and then had to potentially edit those CapsuleBlocks to reflect some keys being moved to lower levels, that could be double the number of record appends than would otherwise be needed.

3.5 Writing to CapsuleDB

All writes to CapsuleDB are first stored in the local memtable. Once the memtable fills, it is marked as immutable and appended to the DataCapsule. The resulting record's hash is then stored in L0 in the index. If this write causes L0 to become full, the compaction process described above is triggered.

3.6 Reading from CapsuleDB

Retrieving a value from CapsuleDB begins at the memtable, as it holds the most recently updated KV pairs. If the requested key is not in the memtable, the request moves to the index for CapsuleBlock retrieval. The index first checks L0. If the key is found after scanning through each block, then the corresponding tuple is returned.

If the search fails, the process is repeated at L1. However, L1 is sorted, so our search can be performed substantially faster. In addition, L1 is likely too large to bring fully into memory, especially given the tight memory constraints imposed by secure enclaves. As such, only the requested block is retrieved from the DataCapsule. Again, it is checked to see if the requested KV pair is present. If not, the same procedure is run at lower levels until it is found, or CapsuleDB determines it does not have the requested KV pair.

3.7 Recovery and Security

One of CapsuleDB's major metrics of success is reducing the mean time to recovery after a PSL node failure. Without CapsuleDB, a failed node would be forced to play back every record in the DataCapsule to reconstruct the state of its memtable. For long-running systems, this

could present a major problem that is only exasperated by other nodes continuing to publish KV pairs on the multicast tree. Instead, with CapsuleDB each node's local memtable becomes akin to a cache. Recovered nodes can then direcly proceed with function execution and simply request needed values from the database, rather than having to painstakingly rebuild their memtable. In the event CapsuleDB fails, restoring the most recent index restores the entire structure of the database. Only KV pairs that have been appended to the DataCapsule since that last index would need to be replayed to reconstruct CapsuleDB's memtable.

In the event of catastrophic failure, there are several other restoration avenues. Playing the entire DataCapsule back is still an option, as the PSL nodes still append every KV pair as they execute. As a result, CapsuleDB has no need for a separate write ahead log, instead able to fall back on the DataCapsule if necessary. In addition, even the loss of a DataCapsule can be resolved, as the GDP maintains several replicas by default. Using DataCapsules as the backing storage system for CapsuleDB also provides powerful security guarantees. As discussed in Section 2, simply by using DataCapsules we gain integrity, authenticity, and provenance of the data with additional confidentiality guarantees if needed.

One interesting behavior to highlight is how CapsuleDB responds to failures during compaction. When compacting, no changes are made to the index until an entire level's references are ready to be updated, instead being collected in a vector. Thus, even if CapsuleDB failed during this process no data would be left in an unusable or undefined state. Now consider if CapsuleDB failed during a size check. In this case, the database would restore the most recent index as normal. Once a new block is pushed to L0, CapsuleDB would detect that L0 is now overfilled. It would thus immediately start the compaction process to free space. Therefore, CapsuleDB will always eventually complete failed compactions.

4 Implementation

We now discuss specific implementation details of CapsuleDB. Our KVS is written in C++ to match the PSL project's code base. We use SHA-256 hashes and leverage the existing memtable implementation used by PSL workers.

4.1 Blocks and Levels

CapsuleBlocks are our base unit of data transfer. Within each block, KV pairs and their associated timestamps are stored in a vector of tuples. The block also stores which level it is associated with, as well as the minimum and maximum keys it contains.

Each level follows a similar structure. It contains a vector of structs, each corresponding to the hash of the records containing the CapsuleBlocks for this level. This struct also contains the min and max key for the block. Each level also stores the min and max key in the level to accelerate look-ups. Finally, the level contains additional metadata about which number level it represents, the number of blocks currently in the level, and the level's maximum size. By default, each level in CapsuleDB is ten times larger than the previous level.

4.2 Index

The index follows a similar design to the levels and CapsuleBlocks. It contains a vector of levels, corresponding to the active levels in the KVS. Each index also contains the number of levels, the size of the CapsuleBlocks, and the hash of the previous index. This hash provides a fast way to find the last snapshot of CapsuleDB, as well as a powerful optimization to keep new indices small. We discuss this optimization further in Section 7.

Note that any time a block is modified, added, or deleted, the corresponding entry in a level must be updated to reflect the newly appended record's hash in the DataCapsule. While the index does handle searching through the levels and compaction, each individual level is responsible for the direct tracking of each CapsuleBlock. Thus, the index can still serve requests to other levels while a different level is applying updates to its list of associated hashes.

4.3 Compaction

After a new CapsuleBlock is written to L0, CapsuleDB always checks whether a compaction is necessary. If a compaction is needed and would not cause L1 to fill, we perform a simple merge sort to combine the two runs of blocks. If it would cause L1 to fill completely, we first flush any blocks that would be affected by the merge from L1 into L2. We do this by evaluating minimum and maximum keys to determine which blocks overlap. L1 and L2 are then merged using the same merge sort implementation as before. Then, since space is now available in L1, we perform the original merge. This process could recursively occur several times at L2 and below before space is finally available for the L0 – L1 merge. As previously mentioned, this process requires writing out new CapsuleBlocks which causes new records to be appended to the DataCapsule. Once all the newly created CapsuleBlocks have been written and hashed, we update the index to reflect the new active blocks.

4.4 Managing Blocks

In general, CapsuleBlocks are stored as records in the DataCapsule. However, there are a few notable exceptions. The entirety of L0 is kept in memory, due to its small size and relatively high rate of access. CapsuleBlocks in lower levels may also be present in memory because they recently served a `get` request. In this case, we do not evict the block for performance reasons until the space is needed. If a later `get` request requires a key in the same block, we can avoid making a comparatively costly DataCapsule access. After reading in so many blocks that we run out of memory in the enclave, we evict the least recently used to make space for an incoming new block. We also evict the block if it is no longer considered active by the index.

Due to this method of managing the CapsuleBlocks, we make one slight adjustment in our key retrieval process. When the hash of the block that may contain the requested key is determined, we first check to see whether the block is already present in memory. If so, we skip requesting the block from the DataCapsule, as we know it will not have changed.

4.5 Writing to CapsuleDB

Writes to CapsuleDB are formatted as a PSL `kvs_payload`, containing the key, the value, a timestamp, and the message type. CapsuleDB utilizes the same memtable implementation as PSL, only modifying it to check if it is full and needs to be written to a CapsuleBlock. This includes the locking mechanism used by PSL to ensure correct concurrent behavior. As in the PSL memtable, any incoming payloads with the same key as existing payloads but an earlier timestamp are also automatically discarded. If the memtable is full, it is written to a CapsuleBlock and a compaction check occurs. At the same time, a new memtable is spawned and begins servicing incoming writes.

4.6 Reading from CapsuleDB

CapsuleDB takes in a key that corresponds to the desired value. We first check whether this exists in the memtable through the existing memtable functions, again utilizing a lock for correct concurrent operations. If not present, the key goes to the index to sequentially check each level, terminating early if the key is found. In levels other than L0, the index evaluates which block to pull by checking the requested key against the maximum and minimum keys contained in each block. If found, the tuple of key, value, timestamp, and message type is returned. Otherwise, CapsuleDB informs the requester that the key is not present.

5 Evaluation

We evaluate the performance of CapsuleDB as a standalone database, and measure latency and throughput of reads and writes.

5.1 Setup

We use the following hardware, benchmark, and DataCapsule setups for our evaluation.

5.1.1 Hardware

We ran our benchmarks on a Macbook with a 6-Core Intel Core i7 @ 2.6 GHz, but only on 4 cores and with 2GB of RAM.

5.1.2 Benchmark Design

End-to-end evaluation starts when the a reader/writer issues the first request, and ends when the reader/writer receives its last request's response from CapsuleDB. We evaluate the performance using a workload generated by YCSB workload generators. Due to the difference between get and put operations, we evaluate different read-write ratios: 100:0, 95:5, 50:50 , and 0:100. All workloads comply with a zipfian distribution, where keys are accessed at non-uniform frequency. We use a total of 10000 records for our read/write operations.

5.1.3 DataCapsules

We implemented the interface for DataCapsules by serializing CapsuleBlocks, computing their hashes, and writing them out to disk. Notably, we do not have signatures on our records. However, due to the cost associated with disk operations, we feel this is still a representative test.

5.2 Results

We measured the standalone performance of CapsuleDB using the YCSB benchcmark. Table 1 shows our throughput is 17857 ops per second for 100% writes and 1149ops per second for 50% reads and writes. We also plotted these results in figure 3. Currently our reads are less performant than writes, but we are considering several optimizations to improve the read throughput which we discuss in Section 7.

YCSB Benchmark	Throughput (ops/s)
100 Writes::0 Reads	17857.14286
50Writes::50 Reads	1149.425287
5 Writes::95 Reads	587.5440658
0 Writes::100 Reads	632.9113924

Table 1: YCSB Benchmarks

Qualitatively we find that we can handle workloads with large amounts of reads and writes (limited only by disk space). This accomplishes a primary goal of CapsuleDB, which was to allow PSLs to manage data sets that eclipse their local enclave memory pools.

6 Related Work

CapsuleDB draws substantially from both the GDP [13] and PSL projects. It is designed to be fully compatible with DataCapsule replication as well as the GDP routing layer. Furthermore, though it is planned to eventually be converted into a standalone system, the current iteration is desgined to be tightly integrated with the PSL architecture. However, CapsuleDB's design takes inspiration from a rich body of literature.

6.1 Level Databases

CapsuleDB follows many of the same conventions as other level databases, most prominently LevelDB [9], PebblesDB [16], RocksDB [18], and SplinterDB [5].

LevelDB is one of the earliest production level databases developed by engineers at Google. It also uses blocks as the base unit of data and is based on the implementation of Google Bigtable's [3] tablets. Unfortunately, LevelDB has a few major limitations. Only a single process can access the database at a time and there is limited client-server support. There have been several projects to extend LevelDB's functionality and improve its performance [20, 19, 10, 21], though their effect is hampered by the weaknesses in LevelDB's base implementation.

RocksDB is a high performance level database produced by Facebook and used in their production systems. It draws on many of the ideas introduced by LevelDB but modernizes them into a more robust system. It supports high levels of concurrency and optimizes the compaction process via multi-threading, allowing the system to efficiently handle terabytes of data. Furthermore, RocksDB offers many customization options for users to tune the database to their precise workload. RocksDB's SST files, which store data on disk, gave us a strong basis for the design of CapsuleBlocks. The general read / write process also was inspired by RocksDB.

However, RocksDB suffers from two major issues. First, it experiences major write amplification – a measurement of how many times the same data is written. This is due to Rocks' compaction process moving KV pairs into a lower level, compacting that level, and then evaluating that lower level again to determine whether it too needs to be compacted. This can result in data being written and moved between different levels multiple times within the same round of compaction. Some work has already been done on this issue [6] though there is still much to be done. Unfortunately, the second issue is not so easily solved. RocksDB has a fairly monolithic code base which makes it difficult to modify to be DataCapsule compatible. One of the authors of this paper actually attempted this, but was unable to complete the modification. Furthermore, we felt that even if Rocks was modified to use DataCapsules, the database would not take advantage of the unique properties of the capsule itself. Also noting that RocksDB has limited security features, we elected to develop CapsuleDB instead.

PebblesDB is a write-optimized KVS designed to be a drop-in replacement for LevelDB or RocksDB. It uses a novel structure called a Fragmented Log-Structured Merge Tree which substantially improves write throughput. However, Pebbles has known memory management errors and is not used in any known production environment. Instead we chose to focus our research on a promising KVS from VMware, SplinterDB.

SplinterDB is a recently published KVS designed for NVMe storage. It uses a novel tree structure and compaction strategy based on pointers to accelerate data management while simultaneously reducing read / write latency. We drew major inspiration from SplinterDB, most notably the flush-then-compact algorithm to reduce write amplification during compaction. Since the record hashes in a DataCapsule can also be thought of as pointers, the way SplinterDB manages its storage strongly influenced the design of our index. The engine is also designed to be small enough to fit inside a TEE, an important design goal for CapsuleDB to match the PSL nodes. Unfortunately, SplinterDB's codebase is not publicly accessible and thus could not be used as a starting point for CapsuleDB. In addition, because our goal is to run CapsuleDB both on the edge and in the cloud, we could not be sure the end devices would have NVMe storage. Using SplinterDB thus would have likely incurred performance penalties on non NVMe storage. However, we were able to correspond with the SplinterDB team for feedback on our design and to answer lingering questions.

6.2 Secure Databases

While not a direct influence on CapsuleDB, we did also consider the design of several KVS's that prioritized security during our initial design phases. BigSecret [15] is a data management framework that can securely distribute sensitive data across multiple clouds, including privately owned resources. There are also other projects examining secure distributed KVS's [22, 8] as well as projects examining the feasibility of KVS's within enclaves [12, 7]. The majority of the security and distributed execution capabilities of CapsuleDB come from

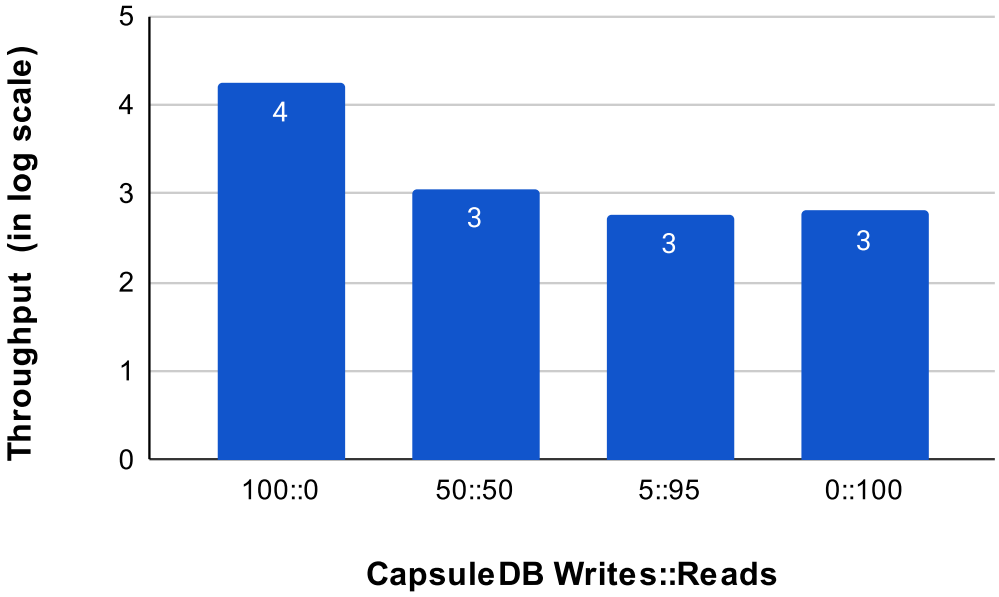

Figure 3: Chart showing CapsuleDB throughput across different workloads, in operations per second.

the GDP and DataCapsules. While related, we consider this work out of scope for this paper on CapsuleDB.

7 Future Work

During the development and implementation of CapsuleDB, we identified several areas for future work. First, our implementations of the index and compaction were somewhat limited. Future iterations of the index could potentially distinguish between full and partial indices. A full index would be as implemented currently, each entry contains the direct hash of the listed CapsuleBlock's record in the DataCapsule. A partial index would only refer to the most recently written blocks. For older blocks in lower levels, it would traverse through previous indices to find the relevant hash. Periodically, a full index would be generated to reduce the number of hops needed to find data. This way, the index kept in memory could be smaller, regaining space for other operations in the small amount of usable memory in an enclave. It is unclear how this change would impact performance, so additional testing would be necessary. As a separate improvement, we could implement a more efficient search algorithm when traversing a level to find the CapsuleBlock that would contain a requested key. While a linear scan works for smaller levels, a binary search or similar algorithm would be straightforward to implement and provide notable performance improvements. SplinterDB also includes quotient filters to accelerate lookups. This would also be a useful addition to our system, though we would have to evaluate whether it should be for each CapsuleBlock or for each level, as both have major performance tradeoffs.

For compaction, a clear change for the next iteration of CapsuleDB is compacting each level in parallel. In the event multiple compactions are needed, this would drastically lower the time for completion. In addition, we currently incur an expensive penalty for inserting KV tuples. Rather than shifting entire blocks, it would be more efficient to break modified blocks into two blocks. However, we would need to change our system to allow for storing underfilled CapsuleBlocks, as well as adjust the logic used to calculate the current size of a level. Fragmentation and underfilled block cleanup operations would also need to be added to our compaction algorithm.

There are also several other potential extensions that offer improvements to users. First, adding additional user control variables, such as level size, compaction strategy, or data management rules, would make it much easier for users to customize CapsuleDB to their specific workload. Similarly, shifting CapsuleDB from this version built for the PSL project to a standalone version would give users access to a secure DataCapsule backed KVS without needing to use PSLs. Additionally, a tool to transfer data in and out of CapsuleDB from other formats, and in the process automatically store the data in a DataCapsule, would improve CapsuleDB's usage in a wide variety of applications.

Finally, a main line of future work in this area would be to integrate CapsuleDB with PSLs. This would involve rewriting PSLs to read from CapsuleDB when a key is not found locally, as well as accessing CapsuleDB for data on failure recovery instead of scanning through the entire DataCapsule. Both were original goals of this project that we weren't able to achieve in time due to various implementation complications, but would allow for more complete end-to-end testing to analyze the effect of CapsuleDB on the PSL project's performance.

8 Conclusion

We presented CapsuleDB, a novel KVS that leverages the unique properties of DataCapsules to efficiently store KV pairs. Using a level-tree based design, we remove several limitations from PSLs to enable a wider range of potential applications. We also discussed several design choices around CapsuleBlocks and CapsuleDB's indexing of active data, namely the management of hashes as pointers to accelerate index updates. When evaluated using YCSB, we found CapsuleDB to be highly performant for our use case, especially on writes.

9 Acknowledgements

We would like to thank Professor John Kubiatowicz and Stephanie Wang for their invaluable support on our project, as well as the GDP research group for their feedback on our design. We would like to especially thank Rob Johnson, Alex Conway, and Chris Ramming from VMware for taking the time to discuss SplinterDB with us.

References

[1] Aws lambda. https://aws.amazon.com/lambda/.

[2] Google cloud functions. https://cloud.google.com/functions/.

[3] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 205–218.

[4] CHEN, K., THOMAS, A., LU, H., MULLEN, W., ICHNOWSKI, J., KUBIATOWICZ, J., JOSEPH, A., AND GOLDBERG, K. Scl: A secure concurrency layer for paranoid stateful lambdas.

[5] CONWAY, A., GUPTA, A., CHIDAMBARAM, V., FARACH-COLTON, M., SPILLANE, R., TAI, A., AND JOHNSON, R. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)* (2020), pp. 49–63.

[6] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing space amplification in rocksdb. In *CIDR* (2017), vol. 3, p. 3.

[7] FEHER, M., LUCANI, D. E., FONSECA, K., ROSA, M., AND DESPOTOV, B. Secure and scalable key value storage for managing big data in smart cities using intel sgx. In *2018 IEEE International Conference on Smart Cloud (SmartCloud)* (2018), IEEE, pp. 70–76.

[8] GEAMBASU, R., LEVY, A. A., KOHNO, T., KRISHNAMURTHY, A., AND LEVY, H. M. Comet: An active distributed key-value store. In *OSDI* (2010), pp. 323–336.

[9] GHEMAWAT, S., AND DEAN, J. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb, December 2021.

[10] LIM, H.-S., AND KIM, J.-S. Leveldb-raw: Eliminating file system overhead for optimizing performance of leveldb engine. In *2017 19th International Conference on Advanced Communication Technology (ICACT)* (2017), IEEE, pp. 777–781.

[11] MCKEEN, F., ALEXANDROVICH, I., ANATI, I., CASPI, D., JOHNSON, S., LESLIE-HURD, R., AND ROZAS, C. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–9.

[12] MESSADI, I., NEUMANN, S., ALMSTEDT, L., AND KAPITZA, R. A fast and secure key-value service using hardware enclaves. In *Proceedings of the 20th International Middleware Conference Demos and Posters* (2019), pp. 1–2.

[13] MOR, N., PRATT, R., ALLMAN, E., LUTZ, K., AND KUBIATOWICZ, J. Global data plane: A federated vision for secure data in edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 1652–1663.

[14] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[15] PATTUK, E., KANTARCIOGLU, M., KHADILKAR, V., ULUSOY, H., AND MEHROTRA, S. Bigsecret: A secure data management framework for key-value stores. In *2013 IEEE Sixth International Conference on Cloud Computing* (2013), IEEE, pp. 147–154.

[16] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China, October 2017).

[17] SHI, W., CAO, J., ZHANG, Q., LI, Y., AND XU, L. Edge computing: Vision and challenges. *IEEE internet of things journal 3*, 5 (2016), 637–646.

[18] TEAM, F. D. E. Rocksdb: A persistent key-value store for flash and ram storage. https://rocksdb.org/, May 2021. Accessed: 2021-05-25.

[19] TULKINBEKOV, K., PIRAHANDEH, M., AND KIM, D.-H. Cleveldb: Coalesced leveldb for small data. In *2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN)* (2019), IEEE, pp. 567–569.

[20] WANG, L., DING, G., ZHAO, Y., WU, D., AND HE, C. Optimization of leveldb by separating key and value. In *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)* (2017), IEEE, pp. 421–428.

[21] XU, R., LIU, Z., HU, H., QIAN, W., AND ZHOU, A. An efficient secondary index for spatial data based on leveldb. In *International Conference on Database Systems for Advanced Applications* (2020), Springer, pp. 750–754.

[22] YUAN, X., WANG, X., WANG, C., QIAN, C., AND LIN, J. Building an encrypted, distributed, and searchable key-value store. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), pp. 547–558.

[23] ZHANG, B., MOR, N., KOLB, J., CHAN, D. S., LUTZ, K., ALLMAN, E., WAWRZYNEK, J., LEE, E., AND KUBIATOWICZ, J. The cloud is not enough: Saving iot from the cloud. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (2015).