Peer-to-peer storage engine for schemaless immutable data

José Ghislain Quenum*

Namibia University of Science and Technology,
Windhoek, Namibia
Email: jquenum@nust.na
*Corresponding author

Alexander Brown Shipena

Motor Vehicle Accident Fund,
Windhoek, Namibia
Email: alexander@mvafund.com.na

Abstract: We present TaYo, a peer-to-peer storage engine explicitly designed for *immutable data* which bypasses the file system during the storage. TaYo uses content-addressable storage (CAS) and Cuckoo hashing to generate a hash of the content that then serves as its identity. In TaYo, we have two I/O operations: read and write. To write data to TaYo, we split it into eight chunks, record the structure in a separate index and assign the chunks to worker processes that write concurrently. Each chunk is replicated twice. To read data, the client has to provide the identifier which the index uses to locate and re-assemble the chunks. TaYo uses a *semi-active* replication technique, a blend of active and passive replication while storing the data. It uses a consensus protocol built on top of Raft to guarantee consistency among the replicas.

Keywords: storage systems; storage engines; data management; peer-to-peer; immutable data; content-addressable storage; CAS.

Reference to this paper should be made as follows: Quenum, J.G. and Shipena, A.B. (2021) 'Peer-to-peer storage engine for schemaless immutable data', *Int. J. Cloud Computing*, Vol. 10, Nos. 5/6, pp.655–668.

Biographical notes: José Ghislain Quenum received his BSc in Computer Science from the National University of Benin, MSc in Computer Science from Paris Dauphine University and PhD in Computer Science from University Pierre and Marie Curie, in 1996, 2002 and 2005, respectively. He is currently an Associate Professor at the Namibia University of Science and Technology. His research interests include multi-agent coordination, software architecture, distributed systems, big data infrastructure and artificial intelligence.

Alexander Brown Shipena received his Bachelor's in Computer Science from the Namibia University of Science and Technology and Master's in Computer Science from the same university, in 2015 and 2018, respectively. He is currently pursuing his PhD. His research interests include distributed systems, concurrency data structures and storage systems.

1 Introduction

A storage engine is one of the central components of a data management system (relational database management system, NoSQL database) It provides several functions, including an abstract layer to execute the input/output (I/O) operations (read, write, update, delete) and an interface to the hardware medium hosting the data. Most storage engines [e.g., lightning memory-mapped database (LMDB) (Henry, 2019), Berkeley DB (Olson et al., 1999), LevelDB (LevelDB: A Fast and Lightweight Key/value Database Library, 2011), RocksDB (Dong et al., 2017) and Wisckey (Lu et al., 2017)] cater for *immutability*. However, they still encompass functions that prove superfluous when dealing with *immutable* data. For example, consider a storage engine that uses a log-structured merge tree (LSM tree) (O'Neil et al., 1996) as its underlying data structure. A typical implementation includes *compaction*, an operation that reclaims copies (or versions) of a piece of data due to previous updates. In the case of immutable (static) data, this function proves to be unnecessary, given that there will always be a single version of any data. More generally, all functions introduced in the design or the implementation to optimise updates and deletes are not needed for immutable data. We argue that baking a set of generic functions in the inner workings of a storage engine increases its complexity while limiting its efficiency.

Moreover, storage engines usually rely on file systems, local or remote [e.g., Google File System (Ghemawat et al., 2003) and inter-planetary file system (IPFS) (Benet, 2014; Hasan et al., 2019)] as an interface to the storage medium. Aghayev et al. (2019) discuss various limitations of this approach, especially in the context of a distributed storage. There is a high-performance overhead introduced by the implementation of transactional support in file systems, the performance burden incurred by the metadata of these file systems and their inability to adopt new developments in storage hardware [e.g., the introduction of shingled magnetic recording (SMR) for hard disk drives (HDD) as well as zoned namespaces ZNS for solid-state drives (SSD)]. Rather, a raw storage approach, which provides more flexibility in managing the storage hardware, is recommended.

In our view, the ongoing data revolution, the growth of the internet of things (IoT) and internet applications have resulted in a proliferation of static data. There is, therefore, a need to design storage systems (and their underlying storage engine) to manage static data efficiently.

In this paper, we introduce TaYo, a peer-to-peer storage engine specifically designed for static data. Also, we do not consider the schema accompanying the data. We treat it as a large binary object (BLOB). To guarantee a unique identifier to some data in TaYo, we follow a content-addressable storage (CAS) approach (Chisvin and Duckworth, 1989; EMC Corporation, 2002), which we blend with a cuckoo hashing (Pagh and Rodler, 2004) mechanism. We hash the original data into a much shorter string that becomes its identifier. In TaYo, we limit the set of I/O operations to exactly two basic operations, *read* and *write*. Finally, we adopt a raw storage approach and write our data directly to the storage medium.

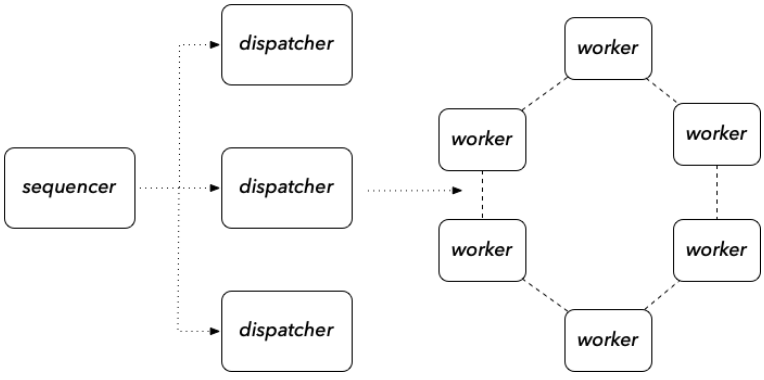
TaYo uses a *semi-active* replication technique, a blend of active and passive replication while storing the data. It uses a consensus protocol built on top of Raft (Howard et al., 2015; Ongaro and Ousterhout, 2014) to guarantee consistency among the replicas. TaYo uses flexible coordination of the processes involved in the read and write operations of static data. With TaYo, we offer a fault-tolerant, scalable and highly available and performing storage engine for static data.

In the remainder of the paper, Section 2 presents an overview of the engine, while section 3 presents the I/O operations. The properties of the storage engine are discussed in Section 4, followed by an evaluation in Section 5. In Section 6, we discuss related work and finally conclude the paper in Section 7.

2 TaYo – an overview

Three main components make up TaYo's architecture: *sequencer*, *dispatcher* and *worker*. Figure 1 depicts their interconnection. Based on the content, the *sequencer* generates a unique identifier (the signature). In our current design, we only use a single instance of the *sequencer*. As such, the generation of the signature is an *asynchronous* operation; most modern frameworks adopt a similar approach [e.g., Node.js (Tilkov and Vinoski, 2010)]. The output of the operation is a *future* (Henrio and Khan, 2010), a programming language construct that supports asynchronous programming. It represents a value that will eventually become available. The future returned by the sequencer indicates whether the operation failed or succeeded. In the latter case, the new identifier will eventually be available.

Figure 1 TaYo – overall architecture

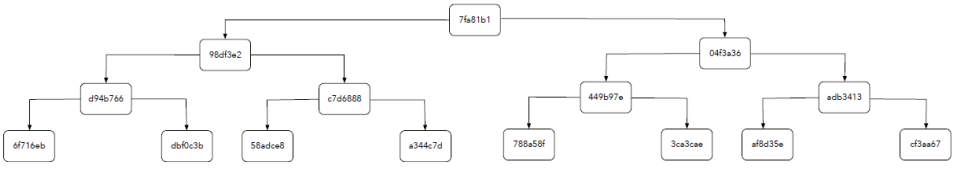


The *dispatcher* maps a data identity to the processes storing the data. During a *write* operation, a *dispatcher* splits the original data into eight chunks and keeps an index of the chunks. The index is a Merkle (1988) tree (see Figure 2 for an example). The *dispatcher* constructs the index by breaking down the original data into two and then repeats the operation until it reaches the eight chunks. Along the process, a k-bit cryptographic hash function hashes each chunk, and the resulting hash value is stored in the tree. The leaves of the tree contain the identifiers of the chunks to be stored. Note that in order to make TaYo *scalable* and *highly available*, we support several instances of the *dispatcher* running concurrently. These three instances share a *global* LSM tree where the indexes are stored. We implement the memtable part of the LSM tree as a skip list.

The *workers* are organised around a ring using a *consistent hashing* (Karger et al., 1997) approach. It allows us to fix the number of nodes involved in the distribution of workload instead of facing the hazards of dynamic changes that can occur. The nodes in the consistent hashing algorithm are referred to as *virtual* nodes since several of them can

be hosted on a single physical machine. The consistent hashing technique comes with sophisticated techniques to contain the dynamic changes that can happen to the infrastructure (a physical node going down, a new node been added to the pool). The workers in our ring are regarded as virtual processes, which we abstract from the physical ones running on the infrastructure. The primary function of the workers is to read and write data chunks; they encapsulate our I/O operations. In Section 3, we elaborate on these operations.

Figure 2 Chunk index



Overall, as the architecture in Figure 1 suggests, TaYo follows three main steps. First, the *sequencer* generates the signature of a static data being added to the store. Next, an instance of the *dispatcher* handles the index representing how the original data is broken down into chunks. Finally, the *workers* actively participate in writing and reading the chunks to and from the storage devices connected to the physical hosts they run on.

3 I/O operations in TaYo

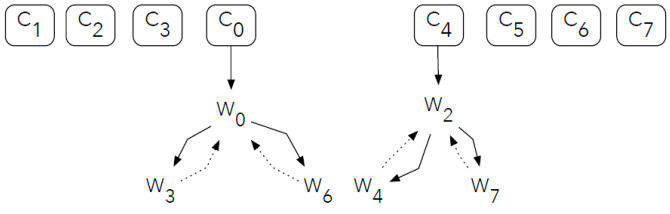
As indicated in Section 2, there are two I/O operations in TaYo: *read* and *write*. We use a *remote procedure call* interface (specifically *gRPC* 3 that runs over *HTTP/2*).

When the *dispatcher* receives a request to write some new data, it first checks if the identifier accompanying the data does not exist already. We use the global LSM tree discussed in Section 2 to perform this check. If the check is successful, i.e., some data with the same identifier already exists in the store, the *dispatcher* stops the operation and sends the notification back. Note that the identifier here is not just one single hash value. Instead, because we are using cuckoo hashing the identifier is a collection of hash values generated by each independent hash function involved. A match is then an exact match for all hash values.

When, on the contrary, the identifier does not exist, the *dispatcher* first creates an entry to a *write-ahead log* (Kim et al., 2016). The value in the log points to a location in the temporary buffer where the original data is stored. Later, after generating the index, the value in the log is replaced with a pointer to the index. The *worker* processes proceed with the *write* operation. First, for each chunk, the replicas elect a *leader* to coordinate the *write* operation. As Figure 3 depicts, *workers* W_0, W_3 and W_6 handle chunk C_0 with W_0 as the *leader*. The three *workers* will then complete the writing of the chunk, and the leader notifies the *dispatcher* that the write operation has completed.

For a *read* operation, the *dispatcher* locates the index corresponding to the data using the identifier. From the index and for each chunk, a *worker* representative is selected to pull the data. The chunks are then assembled by the *dispatcher* and served back in response to the initial request.

Figure 3 Write operation



In building TaYo, we set out to avoid traditional file system usage altogether. We mount the devices as raw devices and directly read from and write to these devices using the Linux/UNIX *dd* utility. Given a raw device located on `196.216.167.110`, Listings 1 and 2 display the commands to write to and read from a storage medium, respectively.

Listing 1 Write command to raw device (see online version for colours)

```
1  $ dd if=/path/to/chunk|ssh sysdev@196.216.167.110
2  'dd of=/path/to/device'
```

Listing 2 Read command from raw device (see online version for colours)

```
1  $ ssh sysdev@196.216.167.110 'dd if=/path/to/chunk'|
2  dd of=/path/to/device
```

4 Properties

4.1 Membership and coordination

The three components in TaYo depicted in Figure 1 (see Section 2) all consist of peer nodes. Because the processes in each of these components can crash or their communication hindered by network partition, we keep a *membership* protocol. Its principle is to monitor all processes involved in the storage system continually and disseminate any failure to the *coordinator* for corrective measures.

In TaYo, we use Consul (https://consul.io), a distributed service mesh, to implement the membership protocol. Consul has several functionalities, including service discovery, KV store, multi-data-centres. However, we use it only for our membership protocol.

Finally, we use ZooKeeper (Hunt et al., 2010) for coordination. First, ZooKeeper complements Consul in membership management. Once Consul detects a failure and disseminates it, Zookeeper then takes the corrective measure to bring the processes back to life. Second, Zookeeper participates in our replication scheme by handling the metadata and organising the leader election.

4.2 Consistency

Brewer (2012) discussed the relationship between *consistency*, *availability* and *network partition* tolerance. The observation, dubbed the CAP theorem, stipulates that only two of these properties can be guaranteed in a distributed system. For example, a distributed

system that favours strong consistency and high availability cannot tolerate network partition. As well, a system that is strongly consistent and tolerates network partition must give up high availability. This observation has influenced and shaped many of the recent NoSQL data stores. However, through a decade of implementing distributed systems following the CAP theorem, many actors have expressed misunderstandings of the theorem. Abadi (2012) argues for a finer-grained delineation between these properties. He further argues that latency should be factored in the absence of network partition. His new idea, PACELC, posits that in the event of a network partition, there should be a trade-off between consistency and availability. However, on the other hand, when there is no network partition, there should preferably be a trade-off between consistency and latency.

In TaYo, we follow the PACELC approach. Note that the individual pieces of data stored in the system will not change due to an update operation. It is somewhat critical in TaYo that over a set period, all replicas acknowledge a copy of a chunk assigned to them. As such, we needed to carefully combine consistency, availability, network partition tolerance and latency.

4.3 Fault tolerance

In TaYo, we only consider *crash* failures and *unreliable links*. A process might fail due to itself crashing or the node hosting it becoming unavailable. As well, the exchanged messages might be lost in transit or messages being duplicated or reordered. Thanks to our membership protocol (see Section 4.1) and the coordination mechanism (see Section 4.1), any failing process will be detected and resumed.

When a *sequencer* process crashes, a new *sequencer* is instantiated and takes over. It is worth noting that this has a limited impact since we use a delegation model (an asynchronous approach to the identity check). However, a *sequencer* might crash before the task is delegated. In that case, we use a log (Apache Distributed Log, http://bookkeeper.apache.org/distributedlog/blog/). This allows for requests to be stored before being handled in the system. The pending tasks that have not been handled are then replayed after a new *sequencer* is instantiated.

When a *dispatcher* crashes, if there is another instance of a *dispatcher* that is not overloaded, the index being created by the failing *dispatcher* is handed over to the other instance. In the meantime, a new instance of a *dispatcher* is created to keep the load under control. If no *dispatcher* is available to take over the task of a failing *dispatcher*, a new instance is created and given the index in construction.

When a *worker* crashes, a new *worker* is instantiated. For all currently running operations, wherever the failing *worker* was acting as a leader, the leadership is transferred to another member of the corresponding replica set. This can be readjusted later when the lease expires. The new *worker* takes over the workload of the failed one. This workload can be inferred from the indices in the skip list.

4.4 Replication

In TaYo, we use a replication factor of 3 and select our replica from the ring. The first replica is identified using a hash function, and the remaining two replicas are selected as its nearest neighbours (see Kalia et al., 2014).

We follow a *semi-active* replication approach (DeCandia et al., 2007) in TaYo, so that when the replicas have to make a non-deterministic decision, the leader does so. Once all replicas have been identified, a leader election (Garcia-Molina, 1982) is organised among them. The new leader is given a lease. When the lease expires, a new leader is elected.

Our replication model is at a chunk level. Thus, once a piece of data has been split into chunks, all chunk leaders receive their copies simultaneously. They are then responsible for sharing their copy with the respective members of their replica set and notify the *dispatcher* when the operation completes. Figure 3 depicts this replication mechanism.

4.5 Consensus

It can be argued that because we do not deal with data update in TaYo, *consensus* is not a mandatory property. In fact, our *consensus* algorithm that ensures that all replicas responsible for a given chunk have an agreement over the content of that chunk is used for *atomic broadcast* (Turek and Shasha, 1992).

Although ZooKeeper has an in-built consensus algorithm, we chose to use it only for coordination. Both Consul and ZooKeeper oversee the membership protocol and coordination mechanism. As such, TaYo's consensus follows *Raft* (Ongaro and Ousterhout, 2014).

5 Evaluation

5.1 Amplification and communication overhead in TaYo

Traditionally, through its usual operations, a storage engine incurs *read*, *write* and *space* amplification. However, thanks to our design, we curb these ratios to almost 1. First, during a *write* operation, the only extra data that is written is to the global LSM. Thanks to the immutability of our data, the disk component of the LSM only has one level, with a single version of each object. Second, during a *read* operation, the global LSM is checked both in memory and on disk to locate the identifier. No extra read operation is needed apart from the actual chunk. Finally, in terms of space, because we consider our schemaless immutable data as BLOBs, there is no extra metadata being stored. Only the chunks and the LSM are stored.

On the other hand, our replication approach, together with our overall design introduces a communication overhead. To read or write a single object, multiple messages need to be exchanged between the *sequencer*, one or several *dispatchers* and several *workers*.

5.2 Experiments

We implemented TaYo in Rust (Rust Programming Language, https://www.rust-lang.org/en-US/; Blandy, 2015), a fast, system-oriented language. We further integrated two tools, Zookeeper (Hunt et al., 2010) to handle the coordination of the nodes and processes in the system and Consul to check the membership of the processes. For our experiments, we deployed TaYo on a cluster of 12 machines. Each machine has an Intel Xeon E5-2680 0 CPU, with 8 GB of RAM and running on CentOS Linux (release 7.5.1884(Core)) OS.

The central machine running our configuration has 8 GB of RAM. The three machines hosting the *sequencer* and *dispatchers* have 16 GB of RAM. Finally, the remaining eight machines have a 32 GB RAM.

We conducted several experiments to evaluate the performance and effectiveness of the *read* and *write* operations in TaYo. Our experiments consist of reading and writing immutable data of different type and size. One should note that although our experiments use objects such as image, audio and video, any type of object applies. We consider all objects as BLOBs.

Our first experiment highlights how each component performs. We store three objects O_1, O_2 and O_3 in the system and record the execution time of the *read* and *write* operations. O_1 is of type PNG with a size of 72 KB, O_2 is of type MP3 with a size of 7.2 MB and finally, O_3 is of type MKV with a size of 947.8 MB. Table 1 shows the time each component takes for each object during the *write* operation, and Table 2 shows the same for the *read* operation. As one can see the *read* and *write* time are acceptable. One noteworthy mention is that during the *read* operation, the *dispatcher* takes a bit longer because the data has to be reassembled from the chunks.

Table 1 Execution time for *write* operation

	Sequencer	*Dispatcher*	*Worker*	*Total*
O_1	0 m 0.271 s	0 m 1.021 s	0 m 0.253 s	0 m 1.550 s
O_2	0 m 1.398 s	0 m 1.040 s	0 m 0.254 s	0 m 2.692 s
O_3	2 m 48.346 s	0 m 4.410 s	0 m 2.065 s	2 m 54.821 s

Table 2 Execution time for *read* operation

	Sequencer	*Dispatcher*	*Worker*	*Total*
O_1	–	0 m 0.518 s	0 m 0.255 s	0 m 0.773 s
O_2	–	0 m 0.519 s	0 m 0.256 s	0 m 0.775 s
O_3	–	0 m 4.157 s	0 m 2.069 s	0 m 6.226 s

In our next series of experiments, we store objects of type JPG, MP3, MP4 and ZIP. For the first three types, we store three objects O_1, O_2 and O_3 of the same type. The execution times for these experiments are recorded in Tables 3 to 6. Generally, the write operation is faster than the read operation. This is mostly due to the assembling time before the data is sent back during the read operation.

Table 3 Experiment 2: JPG objects

Object	*I/O operation*	*Object size*	*Execution time*
O_4	Write	1.4 MB	0 m 1.108 s
O_5	Write	1.9 MB	0 m 1.164 s
O_6	Write	2.9 MB	0 m 1.179 s
O_4	Read	1.4 MB	0 m 1.060 s
O_5	Read	1.9 MB	0 m 1.085 s
O_6	Read	2.9 MB	0 m 1.092 s

Table 4 Experiment 2: MP3 objects

Object	*I/O operation*	*Object size*	*Execution time*
O_7	Write	12.1 MB	0 m 1.119 s
O_8	Write	13.8 MB	0 m 1.192 s
O_9	Write	16.6 MB	0 m 1.206 s
O_7	Read	12.1 MB	0 m 1.163 s
O_8	Read	13.8 MB	0 m 1.173 s
O_9	Read	16.6 MB	0 m 1.235 s

Table 5 Experiment 2: MP4 objects

Object	*I/O operation*	*Object size*	*Execution time*
O_{10}	Write	10 MB	0 m 1.142 s
O_{11}	Write	18 MB	0 m 1.214 s
O_{12}	Write	37.83 MB	0 m 1.277 s
O_{10}	Read	10 MB	0 m 1.188 s
O_{11}	Read	18 MB	0 m 1.220 s
O_{12}	Read	37.8 MB	0 m 1.296 s

Table 6 Experiment 2: ZIP object

Object	*I/O operation*	*Object size*	*Execution time*
O_{13}	Write	1 GB	0 m 5.994 s
O_{13}	Read	1 GB	0 m 6.169 s

Table 7 Experiment 3: concurrent I/O

Object	*Object type*	*I/O operation*	*Object size*	*Execution time*
O_{14}	JPG	Write	1.9 MB	0 m 1.118 s
O_{15}	MP3	Write	12.1 MB	0 m 1.239 s
O_{16}	MP4	Write	18 MB	0 m 1.430 s
O_{17}	ZIP	Write	1 GB	0 m 4.996 s
O_{14}	JPG	Read	1.9 MB	0 m 1.196 s
O_{15}	MP3	Read	12.1 MB	0 m 1.131 s
O_{16}	MP4	Read	18 MB	0 m 1.367 s
O_{17}	ZIP	Read	1 GB	0 m 5.871 s

Finally, we store several objects of different types concurrently and record the results in Table 7. As Table 7 shows, the execution times increase slightly due to the need for coordination between these objects. However, the gap is marginal.

The last experiment compares the performance of TaYo to two distributed storage systems, namely Ceph (Weil et al., 2006) and IPFS (Benet, 2014). The values in Table 8 indicate the execution time for each component of TaYo as well as the execution time for Ceph and IPFS.

Table 8 Results for Experiment 2

I/O op.	*Audio*	*TaYo*						*Ceph*	*IPFS*
		Sequencer	*Dispatcher*	*Worker*					
				W_1	W_2	W_3	W_4		
Write	O_{18}	0 m 0.000044 s	0 m 0.004753 s	0 m 0.005219 s	0 m 0.005211 s	0 m 0.005178 s	0 m 0.005175 s	0 m 0.023 s	0 m 0.031 s
	O_{19}	0 m 0.000040 s	0 m 0.004764 s	0 m 0.005464 s	0 m 0.005223 s	0 m 0.005151 s	0 m 0.005503 s	0 m 0.020 s	0 m 0.025 s
	O_{20}	0 m 0.000040 s	0 m 0.005301 s	0 m 0.015307 s	0 m 0.015121 s	0 m 0.015386 s	0 m 0.015237 s	0 m 0.024 s	0 m 0.032 s
Read	O_{18}	-	0 m 0.005237 s	0 m 0.021440 s	0 m 0.020307 s	0 m 0.021101 s	0 m 0.022304 s	0 m 0.038 s	0 m 0.035 s
	O_{19}	-	0 m 0.005184 s	0 m 0.235223 s	0 m 0.234252 s	0 m 0.235283 s	0 m 0.222123 s	0 m 0.040 s	0 m 0.040 s
	O_{20}	-	0 m 0.005164 s	0 m 0.237989 s	0 m 0.233650 s	0 m 0.227840 s	0 m 0.246553 s	0 m 0.060 s	0 m 0.046 s
Write	O_{21}	0 m 0.000040 s	0 m 0.004930 s	0 m 0.015216 s	0 m 0.015196 s	0 m 0.015355 s	0 m 0.015188 s	0 m 0.022 s	0 m 0.024 s
	O_{22}	0 m 0.000043 s	0 m 0.004949 s	0 m 0.015228 s	0 m 0.014149 s	0 m 0.015205 s	0 m 0.015771 s	0 m 0.021 s	0 m 0.029 s
	O_{23}	0 m 0.000044 s	0 m 0.004773 s	0 m 0.015737 s	0 m 0.015602 s	0 m 0.015583 s	0 m 0.015771 s	0 m 0.039 s	0 m 0.080 s
Read	O_{21}	-	0 m 0.005072 s	0 m 0.246176 s	0 m 0.259000 s	0 m 0.255510 s	0 m 0.251002 s	0 m 0.024 s	0 m 0.038 s
	O_{22}	-	0 m 0.005184 s	0 m 0.281669 s	0 m 0.276001 s	0 m 0.276902 s	0 m 0.272209 s	0 m 0.052 s	0 m 0.045 s
	O_{23}	-	0 m 0.005164 s	0 m 0.866382 s	0 m 0.869119 s	0 m 0.856451 s	0 m 0.869312 s	0 m 0.086 s	0 m 0.116 s

The execution time of a *write* operation in TaYo can arguably be interpreted in two possible ways: one *optimistic* and the second one *pessimistic*. The *optimistic* interpretation stems from the idea that once the *dispatcher* is done constructing the complete index for a piece of data and with the assistance of a buffer to hold the chunks until they are written to the storage medium on the worker nodes, it can respond to the client that a *write* operation has completed. Should an error occur while writing a chunk, another attempt can be made in the background until it succeeds. For an optimistic interpretation, the execution time of a *write* operation is the sum of the time at the *sequencer* and the *dispatcher*. As for the *pessimistic* interpretation, the time of the worst-performing *worker* is added, since the *worker* processes are run in parallel. As one would expect, TaYo's *write* operation is faster following an optimistic interpretation. Following the pessimistic interpretation, it comes second after IPFS and even outperforms IPFS in some scenarios. Regarding the *read* operation, both Ceph and IPFS outperform TaYo. A similar trend can be observed for the rest of the experiments. Overall, following a *pessimistic* interpretation, all three tools complete the *write* operation during almost the same amount of time, while optimistically, TaYo outperforms the two other tools. As for the *read* operation, both Ceph and IPFS complete faster than TaYo. This is due to *seek* operation that takes place within the global LSM. It is also worth noting that Ceph does not check for duplicates, while IPFS combines all functions, instead of coordinating a variety of components.

6 Related work

With the growing interest in storage systems, many storage engines have been introduced in years. Berkeley DB (Olson et al., 1999) is a fast, flexible, reliable and scalable storage engine. Berkeley DB uses a B-tree as one of its underlying data structures to support the access and management of data. LevelDB (LevelDB: A Fast and Lightweight Key/value Database Library, 2011; Forfang, 2014) is a storage engine built on top of an LSM tree. There were several variations introduced in the underlying LSM tree in Slim DB (Ren et al., 2017), Wisckey (Lu et al., 2017) and Rocks DB (Dong et al., 2017). Although these storage engines provide an efficient way of handling read, write and even optimise the space management, they are not explicitly designed for immutable data. As such, they introduce unnecessary complexity in the storage process and hinder its efficiency.

IPFS (Benet, 2014) is a distributed file system built on top of CAS that uses a distributed hash table, block exchange and version control system. IPFS shares many similarities with TaYo. However, they differ on the focus on immutable data as well as the management of the storage medium. Google Colossus (Serenyi, 2017) is the next generation cluster level file system, the successor to the Google File System. It has an automatically sharded metadata layer, sharding partitions data by key ranges and distributes the data among two or more database instances. The data is written using Reed-Solomon (RS) (Fikes, 2010), an error-correcting code. Google Colossus provides a client-driven replication and can combine disks of various sizes and support workloads of varying types. Finally, Ceph (Weil et al., 2006) was originally designed as a reliable and distributed file system that separates the data and its metadata. Over time, Ceph's has evolved from using a file system to adopting Bluestore, a raw storage medium management tool.

Given the increasing number of internet and cloud-native applications that essentially generate immutable data, we argue for a need for an efficient and fault-tolerant storage engine that can manage immutable data. With TaYo, we introduce a peer-to-peer system that addresses that need.

7 Conclusions

In this paper, we discussed a storage engine explicitly built for immutable data. We presented the architecture of the system and discussed its core properties. Although the results we obtained from TaYo's prototype are encouraging, there is still room for improvement. In the future, we wish to pursue the following directions. First, in order to improve the performance of the storage system, we would like to introduce more concurrency, including for the data structures that we manipulate. This should be coupled with an increased level of parallel execution for some tasks. For example, constructing the index can be done in parallel by several processes as the data is being split into chunks. Another improvement we would like to introduce in the future is to enhance the interface with advanced search functionality. As well, we wish to explore other networking paradigms, such as remote direct memory access (RDMA) and assess its impact on the performance. As well, we wish to introduce a fine-grained control of the storage medium through a virtualisation layer.

References

Abadi, D. (2012) 'Consistency tradeoffs in modern distributed database system design: cap is only part of the story', *Computer*, February, Vol. 45, No. 2, pp.37–42.

Apache Distributed Log [online] http://bookkeeper.apache.org/distributedlog/blog/ (accessed 30 November 2019).

Aghayev, A., Weil, S., Kuchnik, M., Nelson, M., Ganger, G.R. and Amvrosiadis, G. (2019) 'File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution', in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, ACM, New York, NY, USA, pp.353–369.

Benet, J. (2014) *IPFS-content Addressed, Versioned, p2p File System*, arXiv preprint arXiv: 1407.3561.

Blandy, J. (2015) *The Rust Programming Language: Fast, Safe, and Beautiful*, O'Reilly Media, Inc., Illinois, USA.

Brewer, E. (2012) 'Pushing the cap: strategies for consistency and availability', *Computer*, February, Vol. 45, No. 2, pp.23–29.

Chisvin, L. and Duckworth, R.J. (1989) 'Content-addressable and associative memory: alternatives to the ubiquitous ram', *Computer*, July, Vol. 22, No. 7, pp.51–64.

Consul [online] https://consul.io (accessed 30 November 2019).

Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T. and Strum, M. (2017) 'Optimizing space amplification in RocksDB', in *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, Chaminade, CA, USA, 8–11 January.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A. and Vogels, W. (2007) 'Dynamo', *ACMSIGOPS Operating Systems Review*.

EMC Corporation (2002) *EMC Centera: Content Addressed Storage System, Data Sheet*, Technical report, April.

Fikes, A. (2010) 'Storage architecture and challenges', *Faculty Summit*, July.

Forfang, C. (2014) *Evaluation of High Performance Key-value Stores*, Master thesis, September, Norwegian University of Science and Technology.

Ghemawat, S., Gobioff, H. and Leung, S-T. (2003) 'The Google File System', in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, ACM, New York, NY, USA, pp.29–43.

Garcia-Molina, H. (1982) 'Elections in a distributed computing system', *IEEE Trans. Comput.*, January, Vol. 31, No. 1, pp.48–59.

Henry, G. (2019) 'Howard Chu on lightning memory-mapped database', *IEEE Software*, November, Vol. 36, No. 6, pp.83–87.

Henrio, L. and Khan, M.U. (2010) 'Asynchronous components with futures: semantics and proofs in Isabelle/HOL', *Electronic Notes in Theoretical Computer Science, Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010)*, Vol. 264, No. 1, pp.35–53.

Hunt, P., Konar, M., Junqueira, F.P. and Reed, B. (2010) 'Zookeeper: wait-free coordination for internet-scale systems', in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10*, USENIX Association, Berkeley, CA, USA, p.11.

Hasan, S.S., Sultan, N.H. and Barbhuiya, F.A. (2019) 'Cloud data provenance using IPFS and blockchain technology', in *Proceedings of the Seventh International Workshop on Security in Cloud Computing, SCC '19*, ACM, New York, NY, USA, pp.5–12.

Howard, H., Schwarzkopf, M., Madhavapeddy, A. and Crowcroft, J. (2015) 'Raft refloated: do we have consensus?', *SIGOPS Oper. Syst. Rev.*, January, Vol. 49, No. 1, pp.12–21.

Kim, W-H., Kim, J., Baek, W., Nam, B. and Won, Y. (2016) 'NVWAL: exploiting NVRAM in write-ahead logging', in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16*, Association for Computing Machinery, New York, NY, USA, pp.385–398.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D. (1997) 'Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web', in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97*, ACM, New York, NY, USA, pp.654–663.

Kalia, D., Ramachandran, P., Li, Y. and Chaudhary, S. (2014) 'Implementation of chord P2P protocol using bidirectional finger tables', *CSE*.

LevelDB: A Fast and Lightweight Key/value Database Library (2011) [online] http://code.google.com/p/leveldb (accessed 2020-05-09).

Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. (2017) 'WiscKey: separating keys from values in SSD-conscious storage', *ACM Trans. Storage*, March, Vol. 13, No. 1, pp.5:1–5:28.

Merkle, R.C. (1988) 'A digital signature based on a conventional encryption function', in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, CRYPTO '87*, Springer-Verlag, London, UK, pp.369–378.

Olson, M.A., Bostic, K. and Seltzer, M. (1999) 'Berkeley DB', in *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '99*, USENIX Association, Berkeley, CA, USA, p.43.

O'Neil, P., Cheng, E., Gawlick, D. and O'Neil, E. (1996) 'The log-structured merge-tree (LSM-tree)', *Acta Inf.*, June, Vol. 33, No. 4, pp.351–385.

Ongaro, D. and Ousterhout, J. (2014) 'In search of an understandable consensus algorithm', in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, USENIX Association, Philadelphia, PA, pp.305–319.

Pagh, R. and Rodler, F.F. (2004) 'Cuckoo hashing', *Journal of Algorithms*, Vol. 51, No. 2, pp.122–144.

Rust Programming Language [online] https://www.rust-lang.org/en-US/ (accessed 30 November 2019).

Ren, K., Zheng, Q., Arulraj, J. and Gibson, G. (2017) 'SlimDB: a space-efficient key-value storage engine for semi-sorted data', *Proc. VLDB Endow.*, September, Vol. 10, No. 13, pp.2037–2048.

Serenyi, D. (2017) 'How we use colossus to improve storage efficiency', *Scalable Intensive Computing Systems*, November.

Turek, J. and Shasha, D. (1992) 'The many faces of consensus in distributed systems', *Computer*, June, Vol. 25, No. 6, pp.8–17.

Tilkov, S. and Vinoski, S. (2010) 'Node.js: using JavaScript to build high-performance network programs', *IEEE Internet Computing*, November, Vol. 14, No. 6, pp.80–83.

Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E. and Maltzahn, C. (2006) 'Ceph: a scalable, high-performance distributed file system', in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, USENIX Association, Seattle, WA, USA, 6–8 November, pp.307–320.