Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store

Anastasios Papagiannis, *Foundation of Research and Technology-Hellas (FORTH) and University of Crete;* Giorgos Saloustros, *Foundation of Research and Technology-Hellas (FORTH);* Pilar González-Férez, *Foundation of Research and Technology-Hellas (FORTH) and University of Murcia;* Angelos Bilas, *Foundation of Research and Technology-Hellas (FORTH) and University of Crete*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Tucana: Design and implementation of a fast and efficient scale-up key-value store

Anastasios Papagiannis[1], Giorgos Saloustros, Pilar González-Férez[2], and Angelos Bilas[1]
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {apapag,gesalous,pilar,bilas}@ics.forth.gr

Abstract

Given current technology trends towards fast storage devices and the need for increasing data processing density, it is important to examine key-value store designs that reduce CPU overhead. However, current key-value stores are still designed mostly for hard disk drives (HDDs) that exhibit a large difference between sequential and random access performance, and they incur high CPU overheads.

In this paper we present *Tucana*, a feature-rich key-value store that achieves low CPU overhead. Our design starts from a B^{ε}–tree approach to maintain asymptotic properties for inserts and uses three techniques to reduce overheads: copy-on-write, private allocation, and direct device management. In our design we favor choices that reduce overheads compared to sequential device accesses and large I/Os.

We evaluate our approach against RocksDB, a state-of-the-art key-value store, and show that our approach improves CPU efficiency by up to $9.2\times$ and an average of $6\times$ across all workloads we examine. In addition, *Tucana* improves throughput compared to RocksDB by up to $7\times$. Then, we use *Tucana* to replace the storage engine of HBase and compare it to native HBase and Cassandra two of the most popular NoSQL stores. Our results show that *Tucana* outperforms HBase by up to $8\times$ in CPU efficiency and by up to $10\times$ in throughput. *Tucana*'s improvements are even higher when compared to Cassandra.

1 Introduction

Recently, NoSQL stores have emerged as an important building block in data analytics stacks and data access in general. Their main use is to perform lookups based on a key, typically over large amounts of persistent data and over large numbers of nodes. Today, Amazon uses Dynamo [16], Google uses BigTable [9], Facebook and Twitter use both Cassandra [29] and HBase [2].

The core of a NoSQL store is a key-value store that performs (key,value) pair lookup. Traditionally key-value stores have been designed for optimizing accesses to hard disk drives (HDDs) and with the assumption that the CPU is the fastest component of the system (compared to storage and network devices). For this reason, key-value stores tend to exhibit high CPU overheads. For instance, our results show that popular NoSQL stores, such as HBase and Cassandra, require several tens or hundreds of thousands of cycles per operation. For relatively small data items we therefore need several modern cores to saturate a single 1 Gbit/s link or equivalently a 100-MB/s-capable HDD. Given today's limitations in power and energy and the need to increase processing density, it is important to examine designs that not only exhibit good device behavior, but also improve host CPU overheads.

Our goal in this paper is to draw a different balance between device and CPU efficiency. We start from a B^{ε}–tree [7] approach to maintain the desired asymptotic properties for inserts, which is important for write-intensive workloads. B^{ε}–trees achieve this amortization by buffering writes at each level of the tree. In our case, we assume that the largest part of the tree (but not the data items) fit in memory and we only perform buffering and batching at the lowest part of the tree. Then, we develop a design that manages variable size keys and values, deals with persistence, and stores data directly on raw devices.

Although we still use the buffering technique of B^{ε}–trees to amortize I/Os, we take a different stance with respect to randomness of I/Os. Unlike LSM-trees [43], we do not make an effort to generate large I/Os. LSM-trees produce large I/Os by maintaining large sorted containers of data items in memory, which can then be read or writ-

[1]Also with the Department of Computer Science, University of Crete, Greece.
[2]Also with the Department of Computer Engineering, University of Murcia, Spain.

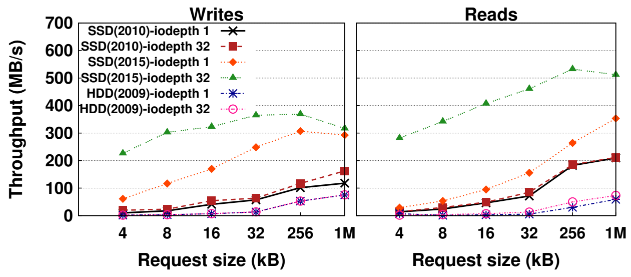


Figure 1: Throughput vs. block size for one HDD and two SSDs, measured with FIO [3].

ten as a whole. These large sorted containers are maintained via a compaction technique that relies on sorting and merging smaller pieces. Although this approach has proven extremely effective for HDDs, it results in high CPU overheads and I/O amplification, as we show in our evaluation for LSM-trees.

New storage technologies, such as flash-based solid-state drives (SSDs) and non-volatile memory (NVM) are already part of the I/O hierarchy with increasing use. Such devices decrease the role of randomness in data accesses. Figure 1 shows throughput for random I/Os on two different generations of SSDs and a HDD for different queue depths and request sizes. We see that HDDs achieve peak throughput for request sizes that approach 1 MB for both reads and writes. Increasing the number of outstanding I/Os does not provide a significant benefit. On the other hand, a commodity SSD of 2015 achieves both maximum write throughput and more than 90% of the maximum read throughput at 32 outstanding requests of 32 KB size. The 2010 SSD has roughly the same behavior with 256 KB requests. Allowing a higher degree of randomness enables us to reduce read and write traffic amplification in the design of the key-value store, which has significant cost in terms of CPU and memory.

We design a full featured key-value store, *Tucana*, that achieves lower host CPU overhead per operation than other state-of-the-art systems. *Tucana* provides persistence and recovery from failures, arbitrary dataset sizes, variable key and value sizes, concurrency, multi-threading, and versioning. We use copy-on-write (CoW) to achieve recovery without the use of a log, we directly map the storage device to memory to reduce space (memory and device) allocation overhead, and we organize internal and leaf nodes similar to traditional approaches [11] to reduce CPU overhead for lookup operations.

To evaluate our approach, we first compare with RocksDB, a state-of-the-art key-value store. Our results show that *Tucana* is up to 9.2$\times$ better in terms of cycles/op and between $1.1 \times$ *to* $7\times$ in terms of ops/s, across all workloads. This validates our hypothesis that randomness is less important for SSD devices, when there is an adequate degree of concurrency and relatively small I/O requests.

To examine the impact of our approach in the context of real systems, we use *Tucana* to improve the throughput and efficiency of HBase [2], a popular scale-out NoSQL store. We replace the LSM-based storage engine of HBase with *Tucana*. Data lookup, insert, delete, scan, and key-range split and merge operations are served from *Tucana*, while maintaining the HBase mapping of tables to key-value pairs, client API, client-server protocol, and management operations (failure handling and load balancing). The resulting system, H-*Tucana*, remains compatible with other components of the Hadoop ecosystem. We compare H-*Tucana* to HBase and Cassandra using YCSB and we find that, compared to HBase, H-*Tucana* achieves between $2-8\times$ better CPU cycles/op and $2-10\times$ higher operation rates across all workloads. Compared to Cassandra, H-*Tucana* achieves even higher improvements.

Our specific contributions in this work are:

- The design and implementation of a key-value data store that draws a different balance between device behavior and host overheads.
- Practical B^{ε}–tree extensions that leverage mmap-based allocation, copy-on-write, and append-only logs to reduce allocation overheads.
- An evaluation of existing, state-of-the-art, persistent key-value stores and a comparison with *Tucana*, as well as an improved implementation of HBase.

The rest of this paper is organized as follows: Section 2 provides an overview of persistent data structures. Section 3 describes our design. Section 4 presents our evaluation methodology and our experimental analysis. Section 5 reviews prior related work. Finally, Section 6 concludes the paper.

2 Background

Persistent stores can be categorized into four groups: key-value stores [18, 22, 25], NoSQL stores [2, 29], document DBs [12], and graph DBs [37, 52]. The last two categories are generally more domain specific. In this work we target the first two and we use the term *stores* to refer collectively to both categories.

The abstraction offered by key-value stores is typically a flat, object-like abstraction, whereas NoSQL stores offer a table-based abstraction, closer to relational concepts. The operations supported by such stores, regardless of the abstraction used, consist of simple *dictionary* operations: get(), put(), scan(), and delete(), with possible extensions for versioned items and management operations, key-range split and merge. Although this is a simple abstraction over stored data, it has proven to be powerful and convenient for building modern services, especially in the area of data processing and analytics.

State-of-the-art stores [2, 18, 22, 29] have been designed primarily for HDDs and typically use at their core an LSM-tree structure. LSM-trees [43] are a write-optimized structure that is a good fit for HDDs where there is a large difference in performance between random and sequential accesses. LSM-trees organize data in multiple levels of large, sorted containers, where each level increases the size of the container. Additionally, small amounts of search metadata, such as Bloom filters, are used for accelerating scan and get operations.

This organization has two advantages. First, it requires little search metadata because containers are sorted and therefore, practically all I/Os generated are related to data items (keys and values). Second, due to the container size, I/Os can be large, up to several MB each, resulting in optimal HDD performance. The drawback is that for keeping large sorted containers they perform compactions which (a) incurs high CPU overhead and (b) results in I/O amplification for reads and writes.

Going forward device performance and CPU-power trends dictate different designs. In this work, we use as a basis a variant of B-trees, broadly called B$^{\varepsilon}$–trees [7].

B$^{\varepsilon}$–trees are B-trees with an additional per-node buffer. By using these buffers, they are able to batch insert operations to amortize their cost. In B$^{\varepsilon}$–trees the total size of each node is B and ε is a design-time constant between [0,1]. ε is the ratio of B that is used for buffering, whereas the rest of the space in each node (1-ε) is used for storing pivots.

Buffers contain messages that describe operations that modify the index (insert, update, delete). Each such operation is initially added to the tree's root node buffer. When the root node buffer becomes full, the structure uses the root pivots to propagate a subset of the buffered operations to the buffers of the appropriate nodes at the next level. This procedure is repeated until operations reach a leaf node, where the key-value pair is simply added to the leaf. Leaf nodes are similar to B-Trees and they do not contain an additional buffer, beyond the space required to store the key-value pairs. The cost of an insertion in terms of I/Os is $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$, where a regular B-Tree has $O(\log_B N)$ [7, 26].

A get operation is similar to a B-Tree. It traverses the path from the root to the corresponding leaf. This results in similar complexity to B-trees, regarding I/O operations. The main difference is that in a B$^{\varepsilon}$–tree we also need to search the buffers of the internal nodes along the path. A range scan is similar to a get, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed. Therefore, buffers are frequently modified and searched. For this reason, they are typically implemented with tree indexes rather than sorted containers.

Compared to LSM-trees, B$^{\varepsilon}$–trees incur less I/O amplification. B$^{\varepsilon}$–trees use an index, compared to LSM-trees, in order to remove the need for sorted containers. This results in smaller and more random I/Os. As device technology reduces the I/O size required to achieve high throughput, using a B$^{\varepsilon}$–tree instead of an LSM-tree is a reasonable decision.

Next, we present the design of *Tucana*, a key-value store that aims to significantly improve the efficiency of data access.

3 *Tucana* Design

Figure 2 shows an overview of *Tucana*. More specifically, Figure 2a shows the index organization, which uses B$^{\varepsilon}$–trees as a starting point (Section 3.1). In Figure 2b we depict *Tucana*'s approach for allocation and persistence, which we discuss in Sections 3.2 and 3.3, respectively.

3.1 Tree Index

Figure 3 shows the differences between *Tucana* and a B$^{\varepsilon}$–tree. On the left side of the figure we show a B$^{\varepsilon}$–tree, which we explain in Section 2. On the right side of the figure we show *Tucana*, where we distinguish nodes that fit in main memory from those that do not. To improve host-level efficiency (in terms of cycles/op), *Tucana* limits buffering and batching to the lowest part of the tree. In many cases today, the largest part of the index structure (but not the actual data) fits in main memory (DRAM today and byte-addressable NVM in the future) and therefore, we do not buffer inserts in intermediate nodes. *Tucana* design provides desirable asymptotic properties for random inserts, where a single I/O is amortized over multiple insert operations. On the other hand, B$^{\varepsilon}$–trees generate smaller I/Os with higher randomness compared to LSM-trees. However, they do not require compaction operations and incur lower I/O amplification. Using fast storage devices we can trade compactions with smaller random I/Os, compared to what an LSM-tree produces, without affecting device performance.

Figure 2a shows the index organization in *Tucana*. The index consists of internal nodes with pointers to next level nodes and pointers to variable size keys (pivots). We use a separate space per internal node to store the variable size keys themselves. Pointers to keys are sorted based on the key, whereas keys are appended to

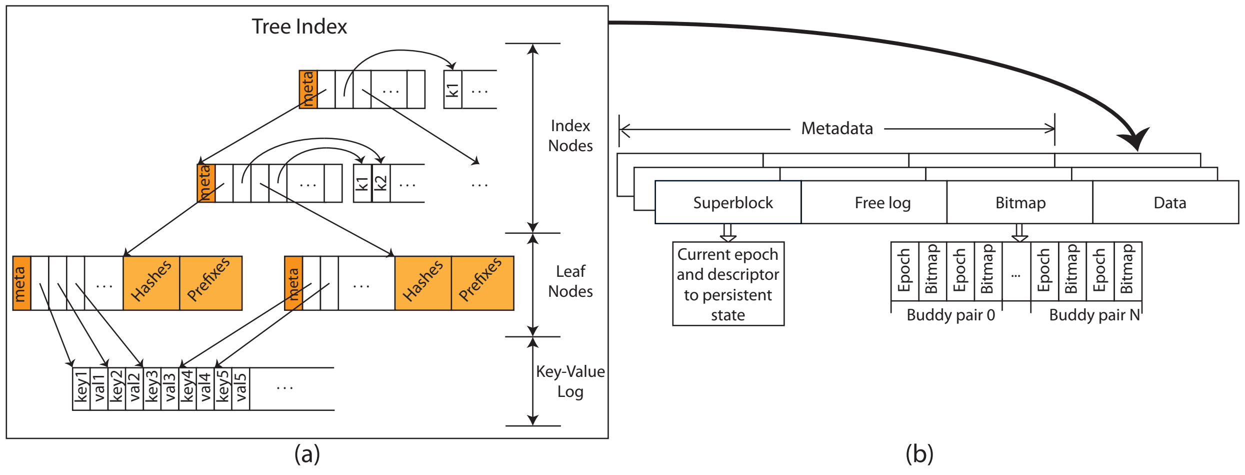


Figure 2: The top-level design of *Tucana*. The left part (a) of the figure shows the tree index. The right part (b) shows the volume layout.

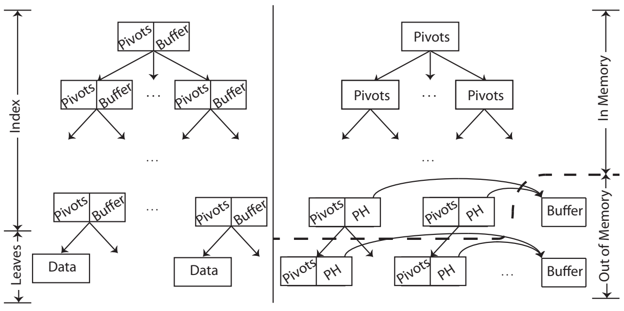


Figure 3: Comparison of B$^{\varepsilon}$–tree (left) and *Tucana* (right). In *Tucana* we distinguish the part of the tree that fits in memory above the dashed line and the rest that does not. PH stands for Prefix-Hash.

the buffer. The leaf nodes contain sorted pointers to the key-value pairs. We use a single append-only log to store both the key and values. The design of the log is similar to the internal buffers of B$^{\varepsilon}$–trees.

Insert operations traverse the index in a top down fashion. At each index node, we perform a binary search over the pivots to find the next level node to visit. When we reach the leaf, we append the key-value pair to the log and we insert the pointer in the leaf, keeping pointers sorted by the corresponding key. Then, we complete the operation. Compared to B$^{\varepsilon}$–trees we avoid the buffering at intermediate nodes. If a leaf is full, we trigger a split operation prior to insert. Split operations, in index or leaf nodes, produce two new nodes each containing half of the keys and they update the index in a bottom-up fashion. Delete operations place a tombstone for the respective keys, which are removed later. Deletes will eventually cause rebalancing and merging [4].

Point queries traverse the index similar to inserts to locate the appropriate leaf. At the leaf, we perform a binary search to locate the pointer to the key-value pair. Since there are no intermediate buffers as in B$^{\varepsilon}$–trees, we do not need to perform searches in the intermediate levels. Finally, range queries locate the starting key similar to point queries and subsequently use the index to iterate over the key range. It is important to notice that in contrast to B$^{\varepsilon}$–trees we do not need to flush all the intermediate buffers prior to a scan operation.

We note that binary search in the leaf nodes and index nodes is a dominant function used by all operations. To reduce memory footprint for metadata, *Tucana* does not store keys in leaves. This means that keys during binary search need to be retrieved from the device. To avoid this, *Tucana* uses two optimizations, prefixes and hashes.

We store as metadata, a fixed-size prefix for each key in the leaf block. Binary search is performed using these prefixes, except when they result in ambiguity, in which case the entire key is fetched from the log. Prefixes improve performance of inserts, point queries, and range queries. In our tuning of prefixes we find that for different types of keys, prefixes eliminate 65%–75% of I/Os during binary search in leaves.

Additionally, a hash value for each key is stored in the leaf nodes. Hashes help with point queries. For a point query we first do a binary search over prefixes. If this results in a tie, then we linearly examine the corresponding (so not all) hashes. We use Jenkins hash function (one-at-a-time) [27] to produce 4-byte hashes. Then the key is read to ensure there is no collision. In our experiments we find that hashes identify the correct key-value pair in more that the 98% of the cases.

The memory footprint can be analyzed as follows. Assume N is the number of keys in the dataset, C is the number of pointers in internal nodes and L in leaves, B is the block size used for internal nodes and leaves, and $h = \log_C N/L$ is the height of the tree. S_k and S_v are the average sizes for keys and values respectively. The size (bytes) of the different components of the index are:

$$dataset = (S_k + S_v)N, \tag{1}$$

$$\begin{aligned} full\ index &= internal\ nodes + leaves \\ &= (B + CS_k)\sum_{i=0}^{h-1} C^i + B\frac{N}{L}, \end{aligned} \tag{2}$$

$$\frac{index}{dataset} = \frac{L(B + CS_k) + BC}{CL(S_k + S_v)}. \tag{3}$$

Equation 3 shows the ratio of the index to the dataset. Now, if we consider that C and L are similar (e.g. in our implementation we use C=230 and L=192) and that B/C is the pointer size, which typically is 8 bytes, we have:

$$\frac{index}{dataset} = \frac{16 + S_k}{S_k + S_v} \tag{4}$$

If we consider that the size of values (S_v) is at least 20× larger than the size of the keys (S_k) and that key size is at least 16 bytes, the index size is around 10% the size of the dataset. Given current cost per GB for DRAM and FLASH, and if we assume that a server spends roughly the same cost for DRAM and FLASH, it is reasonable to assume that the index fits in memory, especially in servers used for data analytics.

For cases when the index fits in memory and the dataset does not, then each search requires one I/O. For inserts, I/O operations for consecutive random inserts are amortized due to the append log.

In case where the index starts to exceed memory, more I/Os are required for each search and insert. Since internal nodes and leaves store pointers to keys and keys are stored in a log, we need two I/Os to read or update an arbitrary item at the bottom of the tree. In this case we need to introduce buffering at additional levels of the index.

3.2 Device layout and access

Figure 2b depicts the data layout in *Tucana*. *Tucana* manages a set of contiguous segments of space to store data. Each segment can be a range of blocks on a physical, logical, virtual block device, or a file. To reduce overhead, segments should be allocated directly on virtual block devices, without the use of a file system. Our measurements show that using XFS as the file system results in a 5-10% reduction in throughput compared to using a virtual block device directly without any file system.

Each segment is composed of a metadata portion and a data portion. The metadata portion contains the superblock, the free log, and the segment allocator metadata (bitmap). The superblock contains a reference to a descriptor of the latest persistent and consistent state for a segment. Modifying the superblock commits the new state for the segment. Each segment has a single allocator common for all databases (key ranges) in a segment. The data portion contains multiple databases. Each database is contained within a single segment and uses its own separate indexing structure.

The allocator keeps persistent state about allocated blocks of a configurable size, typically set to 4 KB, and multiples of it. For this purpose, it uses bitmaps because in key-value stores allocations can be in the order of KBs, as opposed to filesystems that typically do larger allocations. Moreover, allocator bitmaps are accessed directly via an offset and at low overhead, while for searches there are efficient bit parallel techniques [8]. It also maintains state about free operations and performs them lazily in a log structure named *Free log*.

In all persistent key-value stores, including *Tucana*, the index includes pointers to data items in the storage address space. During system operation, part of the index and data are cached in memory. When traversing the index to serve an operation, there is a need to translate storage pointers to pointers in memory. This leads to frequent cache lookups that cannot be avoided easily. Essentially, the cache serves as a mechanism to translate pointers from the storage to the memory address space. Previous work [24] indicates that when all data and metadata fit in memory, managing this cache requires about one-third of the index CPU cycles.

Most key-value stores today follow this caching approach [2, 18, 22, 29, 41]. This allows the key-value store to also control the size and timing of I/O operations between the memory cache and the storage devices, as well as the cache policy.

Instead, *Tucana* uses an alternative approach based on `mmap`. `mmap` uses a single address space for both memory and storage and virtual memory protection to determine the location (memory or storage) of an item. This eliminates the need for pointer translation at the expense of page faults. We note that pointer translation occurs during index operations regardless of whether items are in memory or not, whereas page faults occur only when items are not in memory. The use of `mmap` also allows *Tucana* to use a single allocator for memory and device space management. Additionally, `mmap` eliminates data copies between kernel and user space.

The use of `mmap` has three drawbacks. First, each write operation of variable size is converted to a read-modify-write operation, increasing the amount of I/O. In our design, due to the copy-on-write persistence (see Section 3.3), all writes modify eventually the full page and there can be no reads to unwritten parts of a page. Therefore, we use a simple filter block device in the kernel, which filters read-before-write operations and merely returns a page of zeros. Write and read-after-write operations are not filtered and are forwarded to the actual device. The filter module uses a simple, in-memory bitmap and is initialized and updated by *Tucana* via a set of ioctls. The size of the in-memory bitmap is proportional to the block device size (for 1 TB of storage we need 32 MB of memory).

Second, `mmap` results in the loss of control over the size and timing of I/O operations. `mmap` generates page-sized I/Os (4 KB). To mitigate the impact of small I/Os we use `madvise` to instruct `mmap` to generate larger I/Os. To control their timing we use `msync` for specific items and memory ranges during commit operation.

Third, `mmap` introduces page faults for fetching data. The number of page faults depends on `mmap` kernel page eviction policy. *Tucana* would benefit from custom eviction policies that keep the index and the tail of the append log in memory. In this work, we do not make an attempt to control these policies. However, future work should examine this issue in more detail.

3.3 Copy-on-write persistence

Tucana uses a Copy-on-Write (CoW) approach for persistence instead of a Write-Ahead-Log (WAL). WAL produces sequential write I/Os at the expense of doubling the amount of writes (in the log and later in place). CoW performs only the necessary writes, however, it generates a more random I/O pattern. Therefore, although a WAL is more appropriate for HDDs, CoW has more potential for fast devices. The use of CoW is also motivated by three additional reasons; (a) It is amenable to supporting versioning. (b) It allows instantaneous recovery, without the need to redo or undo a log. (c) It helps increase concurrency by avoiding lock synchronization for different versions of each data item [33], as we discuss in the next subsection.

The state of a segment consists of the allocator, tree metadata, and buffers. CoW is used to maintain the consistency of both allocator and tree metadata. The bitmap in each segment is organized in *buddy pairs*, as shown in Figure 2b. Each *buddy pair* consists of two 4 KB blocks that contain information about allocated space. Each buddy is marked with a global per segment increasing counter named ***epoch***. The epoch field is incremented after a successful commit operation and denotes the latest epoch in which the buddy was modified. At any given point only one buddy of the pair is active for write operations, whereas the other buddy is immutable for recovery. Commits persist and update modified buddy pairs.

The allocator defers free operations with the use of the free log [6]. Directly applying a free operation that could be rolled back in the presence of failures is more complicated as it can corrupt persistent state. We log free operations using their epoch id, and we perform them later after their epoch becomes persistent.

To maintain the consistency of the tree structure during updates, each internal index and leaf node uses epochs to distinguish its latest persistent state. During an update, the node's epoch indicates whether a node is immutable, in which case a CoW operation takes place. After a CoW operation for inserting a key, the parent of the node is updated with the new node location in a bottom-up fashion. The resulting node belongs to epoch+1 and will be persisted during the next commit. Subsequent updates to the same node before the next commit are batched by applying them in place. Since we store keys and values in buffers in an append-only fashion, we need to only perform CoW on the header of each internal node.

Tucana's persistence relies on the atomic transition between consistent states for each segment. Metadata and data in *Tucana* are written asynchronously to the devices. However, transitions from state to state occur atomically via synchronous updates to the segment's superblock with `msync` (commits). Each commit creates a new persistent state for the segment, identified by a unique epoch id. The epoch of the latest persistent state of a segment is stored in a descriptor to which the superblock keeps a reference.

Commits can take place in parallel with read and write operations. To achieve this, a commit is performed in two steps: (1) Initially, it marks the current state as persistent by increasing the epoch of the system. This state includes the bitmap and the tree indexes for this segment. (2) It flushes the state of the segment to the device. In case of a failure during a commit, the segment simply rolls back to the latest persistent state by ignoring any writes that have reached the device but were not committed via the metadata epoch states.

During a commit operation, the bitmap cannot be modified by new allocations (a subset of the write operations) because this may change the state on the device (`mmap` may propagate any write from memory to the device asynchronously). In case the current commit fails, then both *buddy pairs* will be inconsistent. To avoid this, allocations during a commit are buffered in a temporary location in memory and are applied at the end of the commit.

3.4 Concurrency in *Tucana*

Concurrency in key-value stores is important for scaling up as server density increases in terms of CPU, storage, and network throughput. Key-value stores typically operate under high degrees of concurrency, due to the large numbers of client requests.

Similar to most key-value stores, *Tucana* partitions datasets in multiple databases (key ranges). Requests in different ranges can be served without any synchronization. The only exception in *Tucana* is insert operations in different regions that are stored in the same segment. In this case the existence of a single segment allocator requires synchronization across ranges during allocation operations. To reduce the impact of such synchronization, the allocator operates in a batched mode, where a request reserves more space than required for the current operation. Subsequent inserts to the same database do not need to request space from the allocator.

Within each range, *Tucana* allows any number of concurrent reads and a single write without synchronization. To achieve this, *Tucana* uses the versions of the segment created through commits, similar to read-copy-update synchronization [38]. In particular, we serve read operations from the latest persistent version of the segment, which is immutable. Writes on the other hand are served from the modified root which contains all modifications.

Updates applied by an application are visible to readers after a commit. *Tucana*'s API offers additional fence operations to allow higher layers to control when updates become visible.

Finally, in the current state of the prototype, *Tucana* does not allow multiple concurrent writes in the same range. Although there are possible optimizations, especially to allow non-conflicting writes via copy-on-write, or dynamic partitioning of the key-space, we leave these for future work.

3.5 H-*Tucana*

HBase [2] is a scale-out columnar store which supports a small and volatile schema. HBase offers a table abstraction over the data, where each table keeps a set of key-value pairs. Each table is further decomposed into regions, where each region stores a contiguous segment of the key space. Each region is physically organized as a set of files per column, as shown in Figure 4.

At its core HBase uses an LSM-tree to store data [43]. We use *Tucana* to replace this storage engine, while maintaining the HBase metadata architecture, node fault tolerance, data distribution and load balancing mechanisms. The resulting system, H-*Tucana*, maps HBase regions to segments (Figure 4), while each column maps to a separate tree in the segment. In our work, and to eliminate the need for using HDFS under HBase, we modify HBase so that a new node handles a segment after a failure. We assume that segments are allocated over a reliable shared block device, such as a storage area network (SAN) or virtual SAN [39, 50] and are visible to all nodes in the system. In this model, the only function that HDFS offers is space allocation. *Tucana* is designed to manage space directly on top of raw devices, therefore, it does not require a file system. H-*Tucana* assumes the responsibility of elastic data indexing, while the shared storage system provides a reliable (replicated) block-based storage pool.

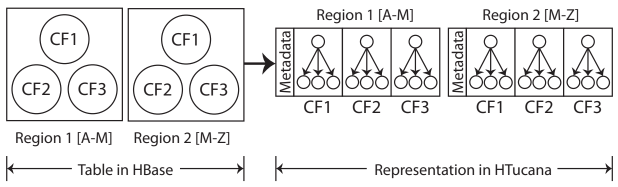


Figure 4: Table storage in HBase and H-*Tucana*. CF stands for column family.

4 Experimental evaluation

In this section, we compare *Tucana* to RocksDB [18] and H-*Tucana* to HBase [2] and Cassandra [29]. *Tucana* and RocksDB support similar features including persistence and recovery, arbitrary size keys and values and versions. In the same category there are other popular key-value stores, such as LevelDB, KyotoDB, BerkeleyDB, and PerconaFT (based on Fractal Index Trees). In our experiments we find that RocksDB outperforms all of them [19, 23] and therefore, we present only the comparison between *Tucana* and RocksDB.

HBase and Cassandra are NoSQL databases that are widely used as a back-end for high throughput systems. HBase and Cassandra use LSM-trees [43].

4.1 Methodology

Our experimental platform consists of two systems (client and server) each with two quad-core Intel(R) Xeon(R) E5520 CPUs running at 2.7 GHz. The server is equipped with 48 GB DDR-III DRAM, and the client with 12 GB. Both nodes are connected with a 10 Gbits/s network link. As storage devices, the server uses four Intel X25-E SSDs (32 GB) and we make a RAID-0 with them using the standard *md* Linux driver. *Tucana* is implemented in C and can be accessed from applications as a shared library. H-*Tucana* is cross-linked between the Java code of HBase and the C code of *Tucana*.

We use the open-source Yahoo Cloud Serving Benchmark (YCSB) [13] to generate synthetic workloads. The default YCSB implementation executes gets as range queries and therefore, exercises only scan operations.

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 1: Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution except for D that follows a latest distribution.

For this reason, we modify YCSB to use point queries for get operations. Range queries are still exercised in Workload E, which uses scan operations.

When comparing RocksDB and *Tucana* we use a low-overhead C++ version of YCSB-C [13, 45]. The original Java YCSB benchmark requires JNI to run with RocksDB and *Tucana*, which are written in C++ and C respectively, incurring high overheads.

In all cases, we run the standard workloads proposed by YCSB with the default values. Table 1 summarizes these workloads. We run the following sequence proposed by the YCSB author: Load the database using workload A's configuration file, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with workload E's configuration file, and run workload E.

When comparing *Tucana* to RocksDB we use 256 YCSB threads and 64 databases (unless noted otherwise) and we choose the appropriate database by hashing the keys. When comparing H-*Tucana* to HBase and Cassandra we use 128 YCSB threads and 8 regions for HBase and H-*Tucana*. Cassandra is hash-based and does not support the notion of region, so we use a single table.

We use a small dataset that fits in memory and a large dataset that does not. The small dataset is composed of 60M or 100M records when using *Tucana* and H-*Tucana*, respectively. The large dataset has 300M or 500M records when using *Tucana* and H-*Tucana*, respectively.

In all the cases, the load phase creates the whole dataset and the run phases issue 5 million operations, bounded also by time (one hour max). With *Tucana*, even in the case of the large dataset the index nodes fit in memory as per our assumptions.

We measure efficiency as cycles/op, which shows the cycles needed to complete an operation on average. We calculate efficiency as:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}}, \quad (5)$$

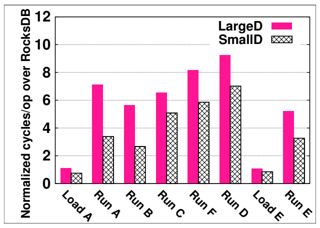
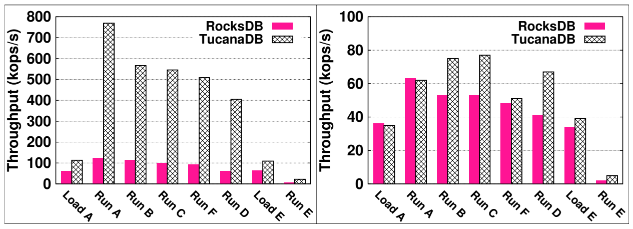


Figure 5: *Tucana* improvement compared to RocksDB in cycles per operation.



(a) Small dataset (b) Large dataset

Figure 6: Performance of *Tucana*DB and RocksDB in ops/s.

where *CPU_utilization* is the global average of CPU utilization among all processors, excluding idle and I/O time, as given by `mpstat`. As cycles/s we use the per-core clock frequency. *average_ops/s* is the throughput reported by YCSB and *cores* is the number of cores including hyperthreads.

4.2 Efficiency of *Tucana*

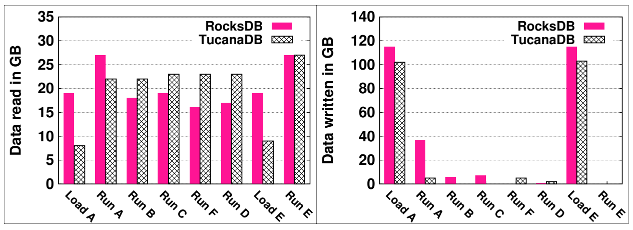
Figure 5 shows the improvement over RocksDB in efficiency. In workloads Load A and Load E that are insert intensive, *Tucana* is similar to RocksDB for both small and large datasets, since both use write-optimized data structures. In all other workloads *Tucana* outperforms RocksDB by 5.2× to 9.2× for the small dataset and by 2.6× to 7× for the large dataset.

We note that increased efficiency can also be achieved with low absolute performance, which is not desirable. Figure 6 shows ops/s for the two systems. We see that for the small dataset *Tucana* outperforms RocksDB by 2× to 7× and by 4.47× on average in absolute performance (throughput) as well. For the large dataset, where both systems are limited by device performance, *Tucana* outperforms RocksDB by 1.1× to 2.1× and by 1.35× on average. Average SSD utilization for all workloads is 93% for *Tucana* and 78% for RocksDB. *Tucana* has on average smaller request size, 86 KB compared to 415 KB for RocksDB. As next generation SSDs close the gap be-

Inserts	Write (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	123	18K	133	31
RocksDB	435	884K	623	100
Speedup			4.68	3.22

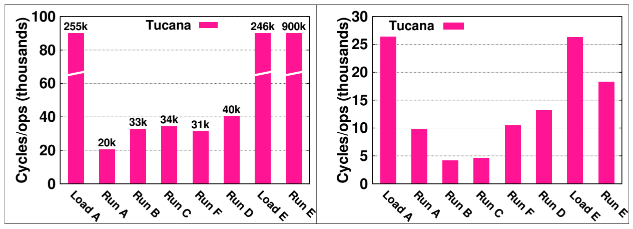
Inserts	Read (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	26	4K	256	140
RocksDB	29	6K	229	171
Speedup			0.89	1.22

Table 2: Performance for the traffic pattern induced by *Tucana* and RocksDB as modeled with FIO to isolate device behavior.



(a) Reads (b) Writes

Figure 7: Total amount of data read and written during each YCSB workload.



(a) 10 qualifiers (b) 1 qualifier

Figure 8: Number of cycles needed for YCSB workloads.

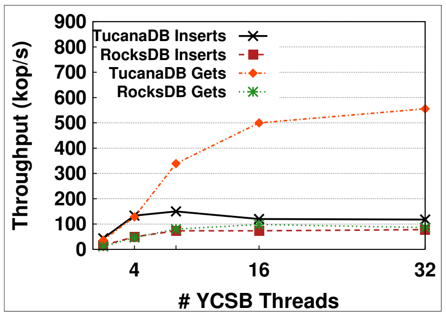


Figure 9: Scalability of RocksDB and *Tucana*DB with increasing threads, using the small dataset.

tween sequential and random performance, we expect even larger performance improvements over RocksDB and similar stores.

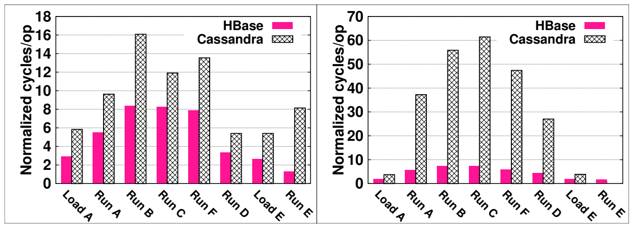
Next, we examine I/O amplification and randomness. We run an insert-only benchmark (random distribution) using a *single* database of size 36.3 GB. RocksDB writes 435 GB while *Tucana* writes 123 GB, thus 3.5× less than RocksDB. Due to compaction operations, RocksDB also reads 2.3× the amount of data read by *Tucana*, 69 GB vs. 29 GB. Table 2 shows the performance difference between these two patterns on two different SSD generations, using FIO (Flexible I/O) [3] to generate each pattern. For inserts, *Tucana*'s I/O pattern is 4.68× faster on the older SSD (2010) and 3.22× faster on the newer SSD (2015), compared to RocksDB's I/O pattern and volume. For gets, the difference in volume size and request size is lower and performance differences are smaller. The I/O pattern of RocksDB is better by 11% for the older SSD, whereas the I/O pattern of *Tucana* is better by 22% for the newer SSD.

Figure 7 shows read and write amplification using 64 databases. Although *Tucana* incurs less I/O on average for both read and write, the difference with RocksDB in this case is smaller. On average RocksDB writes 3.33× and reads 1.25× more data.

Next, we examine the absolute number of cycles/op for each workload (Figure 8a). Each operation is a composite operation over a row with ten qualifiers and therefore a get operation performs ten lookup operations. For this reason, we also present numbers for the same workloads, with one qualifier per row in Figure 8b. In addition, in the case of Workload E the default average length of a range query is fifty. In Figure 8b we change the scan length to five. On average, an insert operation takes about 26K cycles (Load A & Load E), a point query (get) 4K cycles (Run C) and a range query (scan), including initialization and five rows of one qualifier about 18K cycles (Run E). The other workloads are mixes of these operations. If we examine a breakdown of cycles, we see that on average 15% is used by YCSB, 43% by *Tucana*, 38% by the OS kernel, and 4% by other server processes. More specifically, for an insert-only benchmark 35% is used by *Tucana* and 60% by OS kernel. On the other hand for a get-only benchmark 66% is used by *Tucana* and 26% by kernel. System time is due to `mmap` that handles page faults, mappings, and the swapper that evicts dirty pages to devices.

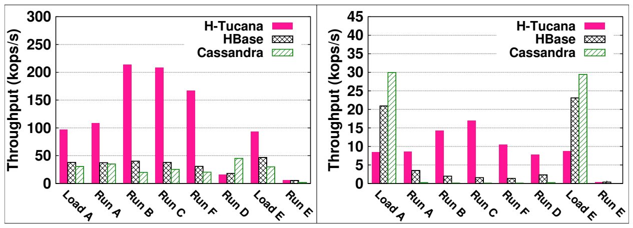
Finally, Figure 9 shows scalability of *Tucana* and RocksDB with the number of server cores. We use the small dataset that fits in memory, partitioned in 64 databases, and we increase the load by increasing the number of YCSB threads that issue requests. For gets, *Tucana* is able to scale and it saturates the full server at 16 YCSB threads. *Tucana* provides lock-less gets and therefore uses all available cores. After warm-up, where data is brought in memory, system utilization is about 100% at 16 YCSB threads. On the other hand, RocksDB, even after warm up, still has about 25% idle CPU time at 8 or more YCSB threads, indicating synchronization bottlenecks.

For inserts, *Tucana* saturates the server at about 8 YCSB threads, where CPU is utilized at 90-95%. RocksDB scales up to 8 threads also, where it saturates the server. Due to its more random I/O pattern, *Tucana* incurs higher device utilization, about 50% vs. 20% for RocksDB. Generally, scaling for puts in both systems is related to the number of databases. In this work, we do not explore this dimension further.



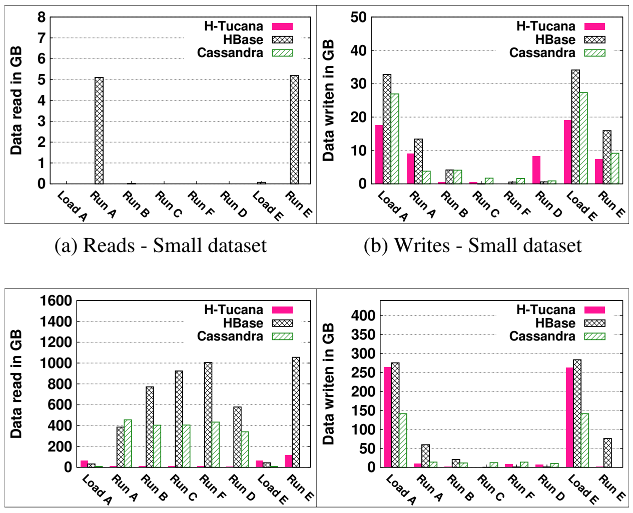
(a) Small dataset (b) Large dataset

Figure 10: Improvement in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra.



(a) Small dataset (b) Large dataset

Figure 11: Throughput (kops/s) achieved by H-*Tucana*, HBase and Cassandra.



(a) Reads - Small dataset (b) Writes - Small dataset

(c) Reads - Large dataset (d) Writes - Large dataset

Figure 12: Amount of data, in GB, read/written by H-*Tucana*, HBase, and Cassandra.

4.3 Impact on NoSQL store performance

In this section, we analyze the efficiency and performance of H-*Tucana*, compared to HBase [2] and Cassandra [29].

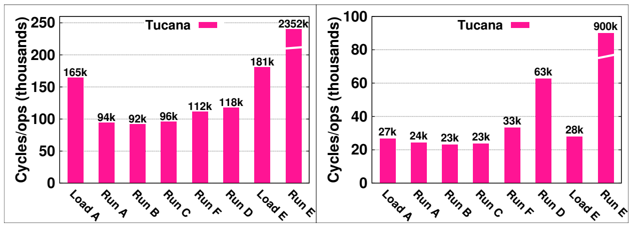
Figure 10 depicts the speedup in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra. We see that H-*Tucana* significantly outperforms both HBase and Cassandra. Compared to HBase, H-*Tucana* uses fewer cycles/op by up to 2.9×, 8.4×, and 5.6× for write-intensive, read intensive, and mixed workloads. Compared to Cassandra, the improvement depends on the size of the dataset. With the small dataset H-*Tucana* outperforms Cassandra by up to 5.8×, 16.1×, and 13.5× for the write, read intensive, and mix workloads, respectively. With the large dataset, H-*Tucana* improves cycles/op over Cassandra by up 3.9×, 61.4×, and 37.2× write, read-intensive and mixed workloads respectively.

Next, we examine throughput in terms of ops/s. Figure 11 shows performance in kops/s whereas Figure 12 depicts the amount of data read and written by each workload.

For the small dataset, H-*Tucana* has up to 5.4× higher throughput compared to HBase, and up to 10.7× compared to Cassandra. In addition, H-*Tucana* does not perform any reads during the run phases. Cassandra does not read any data either, whereas HBase reads 5.1 GB and 5.2 GB when running workloads A and E, respectively. The amount of data written to the device is significantly reduced by H-*Tucana* by 38% and 17% compared to HBase and Cassandra.

For the large dataset, during the run phase, H-*Tucana* outperforms HBase and Cassandra by up to 10.7× and 153.3×, respectively. This improvement is reflected in a significant reduction of the amount of data read from the storage device, by up to 16× and 6.9× compared to HBase and Cassandra, respectively. For read-intensive and mixed workloads, H-*Tucana* is more lightweight not only in CPU utilization but also in the amount of data read. Our modified B$^{\varepsilon}$–tree performs faster lookups than the LSM-trees used by HBase and Cassandra, obtaining significant improvement in throughput.

During the load phase (write intensive workloads) for



(a) 10 qualifiers (b) 1 qualifier

Figure 13: Number of cycles needed by H-*Tucana* for YCSB workloads.

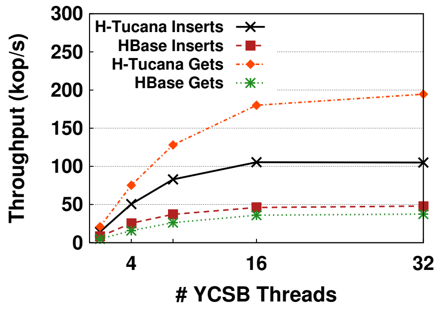


Figure 14: Scalability of H-*Tucana* and HBase with the small dataset.

the large dataset H-*Tucana* exhibits up 2.5× and 3.7× worse throughput than HBase and Cassandra, as shown in Figure 11b). Figure 12 shows that during the load phase, H-*Tucana* writes 264 GB and reads 69 GB, although the size of the dataset including metadata is 77.2 GB. This is not inherent to the design of *Tucana*, as shown by the results in Section 4.2, but rather due to `mmap`, as follows.

With `mmap` modified disk blocks are written to the device not only during *Tucana*'s commit operations, but also periodically, by the flush kernel threads when they are older than a threshold or when free memory shrinks below a threshold, using an LRU policy and `madvise` hints. We believe that due to the increased memory pressure in H-*Tucana* compared to *Tucana* due to the Java HBase front-end, `mmap` evicts not only log pages, but also leaf pages. This reduces the amount of I/Os that can be amortized for inserts due to the limited buffering in our B$^\varepsilon$–tree. To solve this problem, we need to (a) control better which pages are evicted by `mmap`, which will be effective up to roughly the 10-15% ratio of memory to SSD capacity (see Section 3.1), and (b) add buffering one level higher in the B$^\varepsilon$–tree. In the same figure, we notice that in run D phase, using the small dataset, we write more data than the other systems. This is because workload D inserts new key-value pairs and then searches for them. YCSB always searches for keys that exist in the database. In *Tucana* newly inserted keys appear in searches only after a commit operation. If a key is not found, we issue a commit operation to read it. These commit operations cause increased traffic to/from the device. However the other systems retrieve the new values directly from memory. In the large dataset case all systems write them to devices and all of them write about the same amount of data.

Figure 13 shows the cycles/op in H-*Tucana* to execute all the workloads with ten (default configuration) and with one qualifier. With ten qualifiers, write intensive workloads require on average 172K cycles/op and read intensive and mix workloads require on average 115K cycles/op. Workload E that performs scans uses more than 2.3M cycles/op (for retrieving 50 key-value pairs). Figure 13b shows that with a single key, all write-intensive, read-intensive, and mixed workloads require on average 27K cycles/op, whereas workload E requires 900K cycles/op. In more detail, we see that on average 40% of the time is used by the HBase component in H-*Tucana*, 23% by *Tucana*, 33% by the system, and 5% by other processes.

Figure 14 shows the scalability for H-*Tucana* and HBase when increasing the number of YCSB threads at the client. We have not measured the scalability for Cassandra because it is not as competitive. We use the small dataset to avoid accesses to the storage device.

For gets, both systems scale up to 16 YCSB threads. At this point CPU utilization for H-*Tucana* on the server side is 52% and for HBase is 79%, while H-*Tucana* achieves higher throughput. In both cases there is a single thread that reaches 100% CPU utilization. We find that this server thread performs HBase network processing. For inserts, H-*Tucana* scales up to 16 YCSB threads and HBase scales up to 8 YCSB threads. In H-*Tucana*, server CPU utilization is 53%, whereas in HBase 63%. Similar to gets, a single server thread in the HBase front-end limits further scalability.

5 Related Work

B-trees are a prominent structure [4, 40, 41] with good asymptotic behavior for searches and range queries. However, B-trees do not amortize I/Os for inserts and exhibit performance degradation for range queries when they age [17]. This has led to the design of write optimized structures, such as LSM-trees [43], B$^\varepsilon$–trees [7], Fractal trees [44], binomial lists [5], and Fibonacci Arrays [46]. Most of these structures introduce some type of I/O amplification due to compactions. LSM-trees in particular are broadly used today by key-value stores, including LevelDB, RocksDB, HBase, and Cassandra [2, 10, 18, 22, 29]. We categorize related work as follows.

Reducing I/O amplification: WiscKey [32] is based on an LSM-tree and does not require indexes but rather relies on sorted containers and compactions. WiscKey removes the values from the LSM-tree and stores them in an external log. Each key contains a pointer to the corresponding value. This technique improves compactions, because it eliminates the need to sort values. Additionally, it reduces the number of levels in the tree and therefore, the total number of compactions required. *Tucana* is based on a B^{ε}–tree, which has better inherent behavior with respect to amplification. Furthermore, WiscKey inherits explicit I/Os and WAL-based recovery from LevelDB, while *Tucana* was designed to use `mmap` for I/O and CoW for persistency. WiscKey improves performance when values are large compared to keys. *Tucana* on the other hand is designed without particular requirements on sizes of values and keys.

bLSM [47] improves read amplification in LSM-trees with two additional techniques, enhanced use of Bloom filters and efficient scheduling of compactions. VT-Tree [48] tries to reduce I/O amplification by merging efficiently sorted segments of non-overlapping levels of the tree. LSM-trie [53] constructs a trie structure of LSM-trees, and uses a hash-based key-value item organization. B*e*trFS [26], which is based on Fractal Tree Indexes [44], introduces heuristics to reduce write amplification and uses indexes at the buffer level for efficient lookups. Another approach, typically used in distributed NoSQL stores, is to offload compaction to servers managing replicas [1, 21]. *Tucana* starts from a structure that does not require compactions at the expense of more random I/Os. In addition, *Tucana* tries to improve CPU efficiency, which has not been the target of these systems.

SSDs and NVM: FlashStore [14] is a key-value store which builds a storage hierarchy with memory, flash, and disk to provide efficient lookups. NVMKV [35, 36] exploits native FTL capabilities to eliminate write amplification. SkimpyStash [15] stores the key-value pairs in a log-structured manner on flash SSD, to reduce memory footprint. SILT [30] combines three basic structures, a hash, a log, and a sorted store to achieve low memory footprint and reduce read and write amplification. These systems are mainly based on hash structures so they are not able to support efficient prefix and range queries found in analytics. Mercury [20] is an in-memory key-value store that uses a chained hash table and targets real-time applications without scan operations. Masstree [34] uses a trie-like concatenation of B+-trees to handle variable size keys. Masstree does not amortize I/Os for insert operations and scans are challenging to support and inherently expensive with the proposed structure.

At the device level, Wang *et al.* [51] leverage the parallelism in SDF [42], an open-channel SSD whose internal channels can be directly accessed, by providing multi-threaded I/O accesses in the write traffic control policy of LevelDB. They examine the ability to support new operations and interface as is the case with Open Channel SSD [51]. *Tucana* uses the storage device as a black box and it works with off-the-shelf SSDs.

In-memory operation: Silo [49] is inspired by Masstree and it is an in-memory store that does not offer persistence and targets efficient network behavior. HERD [28] is an in-memory key-value cache that leverages RDMA features to deliver low latency and high throughput. Its design is based on MICA [31], an in-memory key-value store that uses a lossy associative index to map keys to pointers and stores the values in a circular log. *Tucana* supports persistence and sits below the network layer.

6 Conclusions

In this work we present *Tucana*, a key-value store that is designed for fast storage devices, such as SSDs, that reduces the gap between sequential and random I/O performance, especially under high degree of concurrency and relatively large I/Os (a few tens of KB). Unlike most key-value stores that use LSM-trees to optimize writes over slow HDDs, *Tucana* starts from a B^{ε}–tree approach to maintain the desired asymptotic properties for inserts. It is a full-feature key-value store that supports variable size keys and values, versions, arbitrary data set sizes, and persistence. The design of *Tucana* centers around three techniques to reduce overheads: copy-on-write, private allocation, and direct device management.

Our results show that *Tucana* is up to $9.2\times$ more efficient in terms of CPU cycles/op for in-memory workloads and up to $7\times$ for workloads that do not fit in memory. In addition, *Tucana* outperforms RocksDB for in memory workloads up to $7\times$, whereas for workloads that do not fit in memory both systems are limited by device I/O throughput. Also, H-*Tucana* is able to improve up to $8\times$ the efficiency of HBase and on average $22\times$ the efficiency of Cassandra.

7 Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Program through the CoherentPaaS (FP7-ICT-611068) and LeanBigData (FP7-ICT-619606) projects, and the NESUS COST programme Action IC1305. We are thankful to Manolis Marazakis for his helpful comments. Finally, we thank the anonymous reviewers and our shepherd Angela Demke Brown for their insightful comments.

References

[1] AHMAD, M. Y., AND KEMME, B. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment 8*, 8 (2015), 850–861.

[2] APACHE. Hbase. `https://hbase.apache.org/`. Accessed: May 23, 2016.

[3] AXBOE, J. Flexible I/O Tester. https://github.com/axboe.

[4] BAYER, R., AND MCCREIGHT, E. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[5] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2007), SPAA '07, ACM, pp. 81–92.

[6] BONWICK, J., AND MOORE, B. Zfs: The last word in file systems.

[7] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 546–554.

[8] BURNS, R., AND HINEMAN, W. A bit-parallel search algorithm for allocating free space. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on* (2001), pp. 302–310.

[9] 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. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[10] 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. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[11] CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. Improving index performance through prefetching. *ACM SIGMOD Record 30*, 2 (2001), 235–246.

[12] CHODOROW, K. *MongoDB: The Definitive Guide*, second ed. O'Reilly Media, 5 2013.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *Proccedings of the VLDB Endowment 3*, 1-2 (Sept. 2010), 1414–1425.

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), pp. 25–36.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[17] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The tokufs streaming file system. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2012), HotStorage'12, USENIX Association, pp. 14–14.

[18] FACEBOOK. Rocksdb. `http://rocksdb.org/`. Accessed: May 23, 2016.

[19] FACEBOOK. Rocksdb performance benchmarks. `https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks`, 2015.

[20] GANDHI, R., GUPTA, A., POVZNER, A., BELLUOMINI, W., AND KALDEWEY, T. Mercury: Bringing efficiency to key-value stores. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, ACM, pp. 6:1–6:6.

[21] GAREFALAKIS, P., PAPADOPOULOS, P., AND MAGOUTIS, K. Acazoo: A distributed key-value store based on replicated lsm-trees. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on* (2014), IEEE, pp. 211–220.

[22] GOOGLE. Leveldb. `http://leveldb.org/`. Accessed: May 23, 2016.

[23] GOOGLE. Leveldb benchmarks. `https://leveldb.googlecode.com/svn/trunk/doc/benchmark.html`, 2015.

[24] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 981–992.

[25] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 435–448.

[26] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 301–315.

[27] JENKINS, B. A hash function for hash Table lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`, 2009.

[28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[29] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[30] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 1–13.

[31] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), pp. 429–444.

[32] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 133–148.

[33] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 183–196.

[34] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 183–196.

[35] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), pp. 207–219.

[36] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, June 2014), USENIX Association.

[37] MARTINEZ-BAZAN, N., GOMEZ-VILLAMOR, S., AND ESCALE-CLAVERAS, F. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on* (April 2011), pp. 124–127.

[38] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

[39] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: Virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 41–54.

[40] MYSQL, A. Mysql, 2001.

[41] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 183–191.

[42] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ' 14)* (2014), pp. 471–484.

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

[44] PERCONA, L. Perconaft. `https://github.com/percona/PerconaFT`. Accessed: May 23, 2016.

[45] REN, J. Ycsb-c. `https://github.com/basicthinker/YCSB-C`, 2015.

[46] SANTRY, D., AND VORUGANTI, K. Violet: A storage stack for iops/capacity bifurcated storage environments. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 13–24.

[47] SEARS, R., AND RAMAKRISHNAN, R. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 217–228.

[48] SHETTY, P. J., SPILLANE, R. P., MALPANI, R. R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 17–30.

[49] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[50] VMWARE, I. Vmware virtual san 6.1 product datasheet. `https://www.vmware.com/files/pdf/products/vsan/VMware_Virtual_SAN_Datasheet.pdf`. Accessed: May 23, 2016.

[51] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 16.

[52] WEBBER, J. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity* (2012), ACM, pp. 217–218.

[53] WU, X., XU, Y., SHAO, Z., AND JIANG, S. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 71–82.