Fast Scans on Key-Value Stores

Markus Pilman*
Snowflake Computing
markus.pilman@snowflake.net

Kevin Bocksrocker*
Microsoft
kebocksr@microsoft.com

Lucas Braun*
Oracle Labs
lucas.braun@oracle.com

Renato Marroquín
Department of Computer Science, ETH Zurich
marenato@inf.ethz.ch

Donald Kossmann*
Microsoft Research
donaldk@microsoft.com

ABSTRACT

Key-Value Stores (KVS) are becoming increasingly popular because they scale up and down elastically, sustain high throughputs for get/put workloads and have low latencies. KVS owe these advantages to their simplicity. This simplicity, however, comes at a cost: It is expensive to process complex, analytical queries on top of a KVS because today's generation of KVS does not support an efficient way to scan the data. The problem is that there are conflicting goals when designing a KVS for analytical queries and for simple get/put workloads: Analytical queries require high locality and a compact representation of data whereas elastic get/put workloads require sparse indexes. This paper shows that it is possible to have it all, with reasonable compromises. We studied the KVS design space and built TellStore, a distributed KVS, that performs almost as well as state-of-the-art KVS for get/put workloads and orders of magnitude better for analytical and mixed workloads. This paper presents the results of comprehensive experiments with an extended version of the YCSB benchmark and a workload from the telecommunication industry.

1. INTRODUCTION

Key-Value Stores (KVS) are becoming increasingly popular. Unlike traditional database systems, they promise elasticity, scalability, and easy deployment and management. Furthermore, the performance of a KVS is predictable: Each get/put request finishes in *constant* time. This feature helps to support service level agreements for applications built on top of KVS.

Recent work [14, 21, 30] has shown that KVS can run OLTP workloads in an efficient and scalable way. All that work adopted a "SQL-over-NoSQL" approach where the data is stored persistently and served using a KVS (i.e., NoSQL) and the application logic (with SQL support) is carried out in a separate processing layer. The big question that we would like to address in this work is whether such a "SQL-over-NoSQL" architecture can support both analytical workloads and OLTP workloads using the same KVS.

*Work performed while at ETH Zurich.

Proceedings of the VLDB Endowment, Vol. 10, No. 11

Key-Value Store	Scan Time
Cassandra [28]	19 minutes
RAMCloud [34]	46 seconds
HBase [20]	36 seconds
RocksDB [16]	2.4 seconds
Kudu [19]	1.8 seconds
MemSQL [31]	780 milliseconds
TellStore-Row	*197 milliseconds*
TellStore-Log	*133 milliseconds*
TellStore-Column	*84 milliseconds*

Table 1: YCSB# Query 1, 50M Records, 4 Storage nodes

This question is relevant because the access patterns of OLTP and analytical workloads are different. The get/put interface of most KVS is sufficient for OLTP workloads, but it is not a viable interface for analytical workloads because these workloads involve reading a large portion, if not all, of the data. As a result, systems for analytical workloads provide additional access methods: They allow data to be fetched all at once (*full table scan*) and to *push down* selection predicates and projections to the storage layer. Most KVS do not have such capabilities and those that do, cannot execute scans with acceptable performance.

To illustrate that current state-of-the-art KVS are not well suited for analytics, Table 1 shows the running times of executing a simple scan over 50 million key-value pairs and returning the maximum value of a specific field (YCSB# Query 1 in Section 7.2.1). For this experiment, we used four machines (configured as explained in Section 7.1). Because RocksDB is an embedded storage engine we ran it in a single process with only one quarter of the data. It took Cassandra [28] about 19 minutes to process this simple query. RAMCloud [34] and HBase [20] needed about half a minute. For these three KVS, we used server-side processing of the aggregate value because shipping all data and executing the aggregation at the client would have made the response times even worse. Given that the entire dataset fits into main memory, these running times are not acceptable. The only systems that had acceptable performance in this experiment were RocksDB [16], MemSQL [31], and Kudu [19]. RocksDB is a highly-tuned, embedded open-source database that is popular for OLTP workloads and used, among others, by Facebook. MemSQL is a distributed, in-memory, relational database system that is highly optimized (among others with just-in-time compilation of expressions). Kudu is a column-oriented KVS specifically designed for mixed workloads (analytics and get/put). But, even these systems are a far cry from good if real-time (sub-second) latency is expected. The different variants of TellStore, which will be presented in this paper, achieved much lower response times.

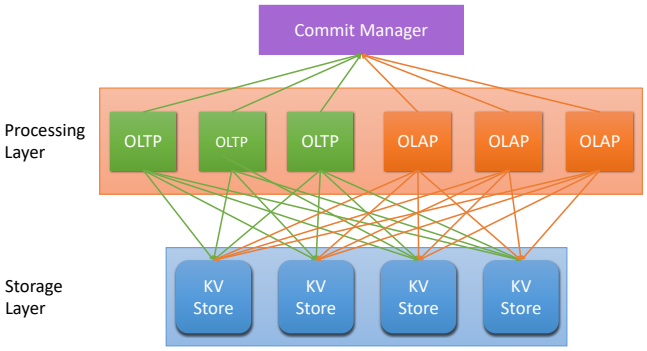


Figure 1: SQL-over-NoSQL Architecture

The poor performance of Kudu which was specifically designed to perform well for such queries shows that it is not easy to achieve fast scans on KVS. The problem is that there are conflicting goals when supporting *get/put* and *scan* operations. Efficient scans require a high degree of spatial locality whereas get/put requires sparse indexes. Versioning and garbage collection are additional considerations whose implementation greatly impacts performance. This paper shows that with reasonable compromises it is possible to support both workloads as well as mixed workloads in the same KVS, without copying the data.

Specifically, this paper makes the following contributions: First, we amend the SQL-over-NoSQL architecture and show how it can be used to process mixed workloads (Section 2). Second, we present the design space for developing a KVS that supports both point operations (*get/put*) and bulk operations (*scans*) efficiently (Section 3). Third, we present TellStore, a distributed, in-memory KVS, and our implementation of two different KVS designs (Sections 4 to 6). Finally, we give the results of performance experiments using an extended YCSB benchmark and a workload from the telecommunications industry that study the tradeoffs of the alternative variants to implement a KVS supporting efficient scans (Section 7). The main result of this work is that it is indeed possible to build a KVS that has acceptable performance for get/put workloads and is highly competitive for analytical workloads. It is important to address all the design questions in a holistic and integrated way as simply composing concepts known from the literature does not work.

2. REQUIREMENTS

2.1 SQL-over-NoSQL Architecture

Figure 1 depicts the SQL-over-NoSQL architecture to support mixed OLTP/OLAP workloads on top of a KVS. In this architecture, the data is stored in a distributed KVS which features a *get/put* and *scan* interface. Transactions and queries are processed by machines in the processing layer. The processing layer is also responsible for synchronizing concurrent queries and transactions. Throughout this work, we use *Snapshot Isolation* [6] for synchronization which can be implemented in a distributed setting using a *Commit Manager* as shown in Figure 1 [30, 12]. Snapshot Isolation (or other forms of Multi-Version Concurrency Control) have emerged as the defacto-standard for synchronization, in particular in distributed systems and for mixed database workloads because OLTP transactions never block or interfere with OLAP queries. With Snapshot Isolation, the commit manager simply assigns transaction timestamps and keeps track of active, committed, and aborted transactions and, thus, rarely becomes the bottleneck of the system [30].

The big advantage of the SQL-over-NoSQL architecture is that it is elastic. Additional machines can be added to both layers (storage and processing) independently. For instance, additional OLAP nodes can be added at the processing layer to process a complex analytical query; these nodes can be shut down or repurposed for other tasks once the query is completed. This architecture also enables the efficient processing of mixed transactional/analytical workloads: Both kinds of workloads can run concurrently with dedicated resources on a single, fresh copy of the data.

To implement this SQL-over-NoSQL architecture efficiently, the distributed KVS must meet the following requirements:

Scans In addition to get/put requests, the KVS must support efficient scan operations. In order to reduce communication costs, the KVS should support selections, projections, and simple aggregates so that only the relevant data for a query are shipped from the storage to the processing layer. Furthermore, support for shared scans is a big plus for many applications [38, 50, 46].

Versioning To support Multi-Version Concurrency Control, the KVS must maintain different versions of each record and return the right version of each record depending on the timestamp of the transaction. Versioning involves garbage collection to reclaim storage occupied by old versions of records.

Batching and Asynchronous Communication To achieve high OLTP performance, it is critical that OLTP processing nodes batch several requests to the storage layer. This way, the cost of a round-trip message from the processing to the storage layer is amortized for multiple concurrent transactions [30]. Furthermore, such batched requests must be executed in an asynchronous way so that the processing node can collect the next batch of requests while waiting for the previous batch of requests to the KVS to complete.

2.2 Why is it Difficult?

The big problem of these three requirements is that they are in conflict. This is why most KVS today (with the notable exception of Kudu) have been designed to support get/put requests only (e.g., Cassandra and HBase), possibly with versioning (e.g., RAMCloud and HBase) and sometimes with asynchronous communication. All these features are best supported with sparse data structures for get/put operations. When retrieving a specific version of a record, it is not important whether it is clustered and stored compactly with other records. Scans, however, require a high degree of data locality and a compact representation of all data so that each storage access returns as many relevant records as possible. Locality is important for both disk-based and in-memory scans. Specifically, adding scans to the feature list creates the following locality conflicts:

Scan vs. Get/Put Most analytical systems use a columnar storage layout to increase locality [2]. KVS, in contrast, typically favor a row-oriented layout in order to process get/put requests without the need to materialize records [4].

Scan vs. Versioning Irrelevant versions of records slow down scans as they reduce locality. Furthermore, checking the relevance of a version of a record as part of a scan can be expensive.

Scan vs. Batching It is not advantageous to batch scans with get/put requests. OLTP workloads require constant and predictable response times for get/put requests. In contrast, scans can incur high variability in latencies depending on selectivities of predicates and the number of columns that are needed to process a complex query.

Fortunately, as we will see, these conflicts are not fundamental and can be resolved with reasonable compromises. The goal of this paper is to study the design space of KVS and to demonstrate experimentally which compromises work best.

3. DESIGN SPACE

This section gives an overview of the most important design questions to build a KVS that supports bulk operations and scans.

3.1 Where to Put Updates?

There are three possible designs to implement updates (*put, delete,* and *insert*): *update-in-place*, *log-structured*, and *delta-main*.

Update-in-place is the approach taken in most relational database systems. New records (inserts) are stored in free space of existing pages and updates to existing records (puts or deletes) are implemented by overwriting the existing storage for those records. This approach works great if records are fixed-size because there is little or no fragmentation. However, this approach is trickier with versioning. If versions are kept in place (i.e., at the same location as the records), versioning can result in significant fragmentation of the storage and loss of locality. Another problem with the update-in-place approach is that it limits concurrency: To create a new version of a record, the whole page needs to be latched. (A latch is a short-term lock that can be released once the page has been updated.)

Log-structured storage designs were first introduced by [40] for file systems. This approach has been adopted by several KVS; e.g., RAMCloud [34]. The idea is to implement all updates as appends to a log. Log-structured storages have two important advantages: (1) there is no fragmentation; (2) there are no concurrency issues as appends can be implemented in a non-blocking way [29]. A major drawback is that scans can become expensive in a pure log-structured system because scans involve reading (and testing for validity) old versions of records. Furthermore, it is difficult to garbage collect / truncate a log if records are rarely updated. A variation, referred to as log-structured merge-trees (LSM) [35] is used in LevelDB [23], RocksDB [16], and Kudu [19]. This variant involves periodic reorganization of the log to improve read performance.

The third approach, *delta-main*, was pioneered by SAP Hana [17] and has also been used in several research projects; e.g., AIM [10]. This approach collects all updates in a write-optimized data structure (called *delta*) and keeps the bulk of the data in a read-optimized data structure (called *main*). Periodically, these two data structures are merged. This approach tries to combine the advantages of the log-structured approach (fast get/put) with the advantages of update-in-place (fast scans).

3.2 How to Arrange Records?

The two most popular designs are *row-major* and *column-major*. Row-major stores a record as a contiguous sequence of bytes in a page [24]. This layout works well for get/put operations which usually operate on the record as a whole. Column-major vertically partitions the data and stores a whole column of a table (or set of records) as a contiguous sequence of bytes. Such a column-major layout is beneficial for scans as analytical queries often involve only a subset of the columns [25, 3, 2, 45, 9, 8]. In addition, column-major supports vector operations (SIMD) to further speed up bulk operations and scans on modern hardware [47, 49]. A variant of column-major is PAX [5] which stores a set of records in every page, but within the page, all records are stored in a column-major representation. PAX is a good compromise between the pure column and row-major designs.

Column-major performs best on fixed-size values. This is why state-of-the-art systems avoid variable-size values, either by simply disallowing them (as in AIM [10]), allocating them on a global heap and storing pointers (as in MonetDB [8] and HyPer [33]), or using a dictionary and store fixed-size dictionary code words (as e.g. in SAP/HANA [18] and DB2/BLU [39]).

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

3.3 How to Handle Versions?

As described in Section 2, we need to support versioning of records in order to implement Multi-Version Concurrency Control. There are two predominant approaches: (a) store all versions of a record at the same location; (b) chain the versions in a linked list. The first variant is often used in combination with update-in-place. It makes it cheaper to create new versions in that approach at the expense of a more costly garbage collection to compact the pages. The second variant is more suited for log-structured storage, as clustering versions in an append-only data structure would require a costly copy of all previous versions to the head. The pointers linking records together consume space and traversing the list is expensive as it involves multiple cache misses. On the positive side, this second approach simplifies garbage collection because it can be implemented as a truncation of the log of old versions. Furthermore, it involves less fragmentation of data pages.

3.4 When to do Garbage Collection?

With versioning comes garbage collection of old versions. There are two possible strategies: (a) do periodic garbage collection in a separate dedicated thread/process; (b) piggy-back garbage collection with (shared) scans. Approach (b) increases scan time, but it also trades off garbage collection investment for garbage collection benefits: Tables which are scanned frequently and greatly benefit from garbage collection are garbage collected more often than other tables. Another advantage of the piggy-back approach is that it does garbage collection while the data is processed anyway, thereby avoiding additional cache misses to fetch the data.

3.5 Summary

Table 2 gives an overview of the tradeoffs of the alternative approaches in the first three dimensions in terms of storage efficiency (fragmentation), concurrency (additional conflicts), cost to implement versioning and garbage collection, and efficiency for scan and get/put operations. The fourth dimension, garbage collection, is orthogonal to these performance characteristics. The performance of a KVS is determined by the combination of these techniques. Overall, there are a total of 24 different ways to build a KVS using this taxonomy:

$$\begin{aligned}&(\text{update-in-place vs. log-structured vs. delta-main})\\ \times\,&(\text{row-major vs. column-major / PAX})\\ \times\,&(\text{clustered-versions vs. chained-versions})\\ \times\,&(\text{periodic vs. piggy-backed garbage collection})\end{aligned}$$

Furthermore, there are many hybrids; e.g., periodic and piggy-backed garbage collection can be combined. Fortunately, only a subset of these variants make sense: Log-structured updates and a

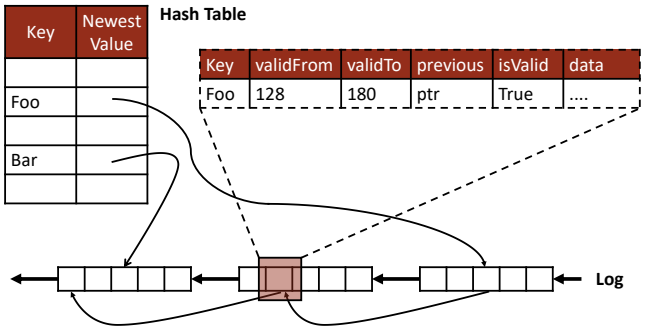


Figure 2: Data Structures of TellStore-Log

column-major layout do not make sense because these variants are clearly dominated by the "delta-main & column-major" variants. Additionally, the four "log-structured & clustered-versions" variants are dominated by the "log-structured & chained-version" variants.

The two most extreme variants are the variant based on log-structured with chained-versions in a row-major format and the variant using a delta-main structure with clustered-versions in a column-major format. The next two sections describe our implementation of these variants in TellStore, called TellStore-Log and TellStore-Col. Section 6 gives implementation details of TellStore that are important for all TellStore variants. Section 7 contains the results of a performance evaluation that compares the tradeoffs of the different variants and uses the performance of existing KVS (see introduction) as baselines. In addition to TellStore-Log and TellStore-Col, Section 7 includes results of a third variant, TellStore-Row, that is identical to TellStore-Col but with a row-major format to specifically study the tradeoffs between row and column stores for mixed OLTP/OLAP workloads.

4. TELLSTORE-LOG

The implementation of TellStore-Log was inspired by RAMCloud [34] with important amendments to support efficient scans.

4.1 Where to Put Updates?

Figure 2 gives an overview of the most important data structures of TellStore-Log: It details the layout of a record (key-value pair) and the hash table used to index records in the log. The log itself is segmented into a linked-list of pages storing all key-value pairs.

The log is an append-only data structure: Memory in the log can be allocated in a lock-free way by atomically incrementing the page head pointer. Once a record is appended to the log, it becomes immutable. This property makes it easy to replicate and restore the log, as the replication process only needs to monitor the head [41]. Because of its lock-free nature, conflicting entries with the same key can be appended to the log concurrently. The hash table is always the point of synchronization. A record is only considered to be valid after the pointer to the record is successfully inserted or updated in the hash table. In case of conflict, the record in the log will be invalidated before the data becomes immutable. Deletes are written as specially marked updates with no data.

The hash table can become a point of contention when implemented with locks. Current lock-free hash table designs are often optimized for a specific access pattern. Specifically, implementing the resize operation involves a trade-off in lookup and update performance. TellStore pre-allocates a fixed-size hash table shared among all tables in the storage node. The implementation uses an open-addressing algorithm with linear probing to exploit spacial locality in case of collisions [37]. On the downside, open addressing tends to perform badly under high load. To keep memory usage small while allowing for a sufficiently sized table, the hash buckets only store the table ID, record key and pointer to the record (24 bytes).

4.2 How to Arrange Records?

The log-structured approach is inherently tied to a row-major data layout. To support efficient scans, records in the log must be completely self-contained. We especially want to avoid lookups in the hash table to determine if a record is still valid (i.e. not deleted or overwritten). This constraint has some implications for versioning, as we will see in Section 4.3.

TellStore-Log allocates a separate log for every table in the storage node. This allows scans to only process relevant pages, further improving data locality. Furthermore, a scan over the log is sensible to the amount of invalid records in the log, impacting the locality requirement, as we will see in Section 4.4.

4.3 How to Handle Versions?

Immutability of the log forces us to append new versions of a record at the log head. To locate an older version of the key, we form a version-chain by storing a *previous* pointer to the preceding element alongside every record in the log. Additionally, the timestamp of the transaction creating the record is stored in a *valid-from* field in the metadata. This version chain is always strictly ordered from newest to oldest according to the snapshot timestamp, with the hash table pointing to the newest element. Given a snapshot timestamp, a get operation can follow the chain until it reaches the first element qualifying for the snapshot. Following the chain involves costly cache misses as a tradeoff for a fast put.

This design seems to be in conflict with the requirement that all records have to be self-contained: When scanning over a record in the log only the creation timestamp is present, leaving the scan unable to determine if the record has expired. To avoid hash table lookups, we add the expiration timestamp (*valid-to*) as an additional mutable field to the metadata of each record. After successfully writing a record, the *valid-to* field of the previous element (if any) is lazily updated with the *valid-from* field of the new element. Given a snapshot timestamp, the scan can decide if an element qualifies for inclusion in the snapshot only by comparing the two timestamps.

The hash table remains the sole point of synchronization and always points to the newest element. There is no race-condition between updating the hash table and setting the *valid-to* field, as Snapshot Isolation in TellStore does not guarantee visibility for in-progress transactions.

4.4 When to do Garbage Collection?

The performance of scanning the log is impacted by the number of outdated elements that are no longer visible to any active transaction. In order to compact frequently scanned tables more often, garbage collection is performed as part of a regular scan. While scanning a page, its health is calculated as the ratio of the total size of valid elements to the total page size. Once this value drops below a certain threshold, the page will be marked for garbage collection. Marked pages will be rewritten on the next scan by copying the remaining active elements to the head of the log and returning the garbage collected page to the pool of free pages. After copying an element to the log head, the pointer in the version chain for that key must be adjusted. To this end, we need to lookup the key in the hash table and follow the version chain to the right place. This operation is expensive because it has poor cache locality. To reclaim space from non-frequently scanned tables, a background agent schedules scans for tables that have not been garbage-collected in a certain period.

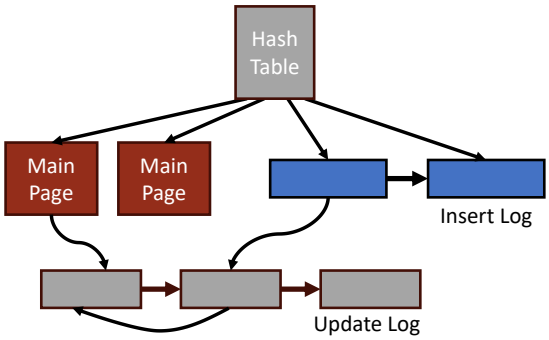


Figure 3: Delta-Main Approach

4.5 Summary

Even though RAMCloud, the poster child of a log-structured KVS, has poor scan performance (Table 1), it is possible to build a log-structured KVS that supports fast get/put requests, versioning, and scans at the same time. The main ingredients are a careful organization of versions in the log, effective garbage collection policies, a lazy implementation of metadata / timestamps to make records self-contained, lock-free algorithms and good engineering.

5. TELLSTORE-COL

The main idea of the delta-main approach, as implemented by TellStore-Col, is to keep two data structures: *main* for reads, and *delta* for updates. As shown in Figure 3, our implementation actually involves four data structures: A list of pages that hold the data of the main, two logs that store the delta (one for inserts and one for updates), and a hash table to index the data.

5.1 Where to Put Updates?

Except for select metadata fields, data in the main is kept read only and all updates are written to an append-only log-structured storage. Unlike TellStore-Log, this delta is split into two separate logs: Updates to records with existing keys are written to the *update-log*, while updates to non-existing keys are written to the *insert-log*. This separation makes it easier to construct the main from the delta, as shown in Section 5.4. The index stores a bit flag that indicates whether a pointer points into the delta or the main. Apart from this flag, the index uses the same hash table as TellStore-Log.

Before writing a record, the index has to be queried for the record's key. In case the key does not exist, the record is appended to the insert-log. As in the log-structured approach, conflicting entries with the same key can be written concurrently to the log. For inserts, the index serves as the point of synchronization. Only after successfully inserting a pointer to the record in the index, the insert becomes valid.

In case the key exists, the record is appended to the update-log. Records in the main and insert-log both contain a mutable *newest* field containing a pointer to the most-recently written element with the same key. Again, conflicting records can be written concurrently to the log. The *newest* pointer constitutes the point of synchronization for updates.

5.2 How to Arrange Records?

While for both delta-logs a row-major format is the only reasonable choice (as discussed in Section 4.2), the main allows us to store records either in row-major or column-major format.

TellStore-Col is the delta-main implementation that stores its *main* pages in a column-major format called *ColumnMap* [10] as depicted in Figure 4. The idea, following the *Partition Attributes Across Paradigm* [5] (PAX), is to first group records into pages and within such a page organize them in column-major format.

In case every field of a table is of fixed size, it is sufficient to know the location of the first attribute of a record. The location of its other attributes can be computed from the number of records in the page and the data type size of each attribute. However, if fields can have arbitrary sizes (as for example variable-length strings), this simple computation falls apart.

To this end, we allocate a heap at the end of every page storing all variable-size fields. This heap is indexed by fixed-size metadata storing the 4-byte offset into the heap and its 4-byte prefix. While the metadata fields are stored in column-major format, the contents of the fields are stored in row-major format in the heap.

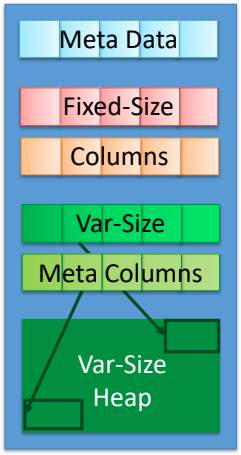


Figure 4: ColumnMap

This has two advantages: First, when materializing records, the variable-size fields are already in row-major format and can simply be copied to the output buffer. Second, by storing the prefix in a fixed-size column-major format, we get a speedup on the commonly used prefix scan queries as we can narrow down the set of candidate tuples without having to look at the heap.

5.3 How to Handle Versions?

Similar to how versioning is implemented in the log-structured approach (see Section 4.3), TellStore-Col stores the creation timestamp in a *valid-from* field as part of the records metadata. Records in the update-log are chained together from newest to oldest using a *previous* pointer. The *newest* pointer stored alongside records in the main and insert-log always points to the newest element in the update-log. To avoid loops, there is no back pointer from the update-log to the main.

Inside a main page, different versions of the same key are stored consecutively from newest to oldest in a column-major format. The *valid-from* timestamp and *newest* pointers are also converted to column-major format and stored as normal attributes in the metadata section. The index always points to the metadata field of the newest element in the ColumnMap or the insert log. Given a snapshot timestamp, the *valid-from* fields are scanned from the newest element until a timestamp is found that is contained in the snapshot.

Storing the *newest* pointer alongside the records instead of in the hash table is required to make records self-contained. Otherwise a scan would be required to perform a lookup in the hash table to determine if a new record was written. Records in both delta-logs only store the timestamp of the transaction that created them and as such are not self-contained. This is a trade-off between scan and garbage collection performance, as discussed in Section 5.4.

5.4 When to do Garbage Collection?

The garbage collection is responsible for regularly merging updates from the two delta-logs back to the compacted and read-optimized main. As such it plays a critical part in guaranteeing the locality requirement for good scan performance. All main pages are immutable and rewritten using a copy-on-write approach. This is necessary in order to not interfere with concurrent access from

get/put and scans, as update-in-place would require a latch on the page (see Section 3.1).

Compared to TellStore-Log, compacting a page is more expensive as it involves converting updates from the row-major format to the column-major format. To this end, the garbage collector runs in a separate thread and not as part of the scan. The dedicated thread periodically scans over the metadata section of every page in the main. As soon as it encounters a page containing records that were either updated (by checking the *newest* field) or are no longer contained in any active snapshot (by checking the *valid-from* field), the page is rewritten.

All versions of the same key are gathered from the main and update-log by following the version chain. Elements with timestamp that are not contained in any active snapshot are discarded, while elements gathered from the update-log are converted to column-major format. The elements are then sorted from newest to oldest and appended to a new main page. After relocating a record, the *newest* field is updated to point to the new record, in order to make concurrent updates aware of the relocation. Finally, the garbage collector scans over all records in the insert-log, gathers all versions of the same key from the update-log and writes them in column-major format to the main page. Afterwards, both delta-logs can be truncated and old main pages are released to the pool of free pages.

By splitting insert from update-log, the collector only has to scan over the insert-log to gather all records with keys not already part of the main. On the downside, this reduces data locality for scans, as updates force the scan to perform random lookups into the update-log when following the version chain. The premise here is that by making garbage collection more efficient, it can run at higher frequency, thus keeping the size of the update-log to a minimum.

Pages are compacted aggressively: A whole page gets rewritten as soon as a single element in it becomes invalid. This can lead to high write-amplification especially under heavy load, which will have a strong impact on disk-based systems but less so on in-memory based systems. An extension to this approach would be to compact pages based on dirtiness, similar to TellStore-Log. Delaying the compaction, on the other hand, will keep a higher portion of the data in the delta-log which, in turn, will impact scan performance.

5.5 Summary

Similar to TellStore-Log, carefully balancing the different requirements for get/put, scan and versioning allows us to design a data structure supporting both transactional and analytical access patterns. While scans profit from the fact that the bulk of the data resides in a compact column-major format, updates benefit from the write-optimized log implementation of the delta. Versioning can be achieved by clustering records of the same key together and treating their timestamp as a regular field in a column-major format. Garbage collection is expensive in the delta-main approach and is, thus, executed in a dedicated thread that periodically garbage collects stale records.

6. IMPLEMENTATION

TellStore is a distributed, in-memory KVS. Clients, such as nodes of a SQL-over-NoSQL system's processing layer, connect to TellStore using Infiniband. Each TellStore instance stores a different subset of the database and supports a get/put, scan, and versioning interface as described in Section 2.

While Sections 4 and 5 covered the overall design of the storage engine, this section describes TellStore as a distributed KVS that can use any of the storage engines described above. More details and a complete description can be found in [36]. Furthermore, TellStore is open source [1].

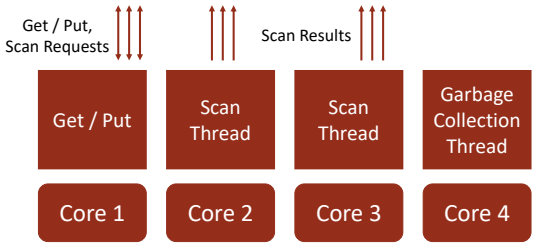


Figure 5: TellStore Thread Model

6.1 Asynchronous Communication

In order to make best use of the available CPU time and network bandwidth, a processing instance should not be idle while waiting for a storage request to complete. This is why TellStore uses an asynchronous communication library, called InfinIO, for its communication with the processing layer. InfinIO, which was built specifically to run on top of Infiniband, employs user-level threads and callback functions through an API similar to the one provided by *Boost.Asio* for Ethernet communication. All requests to TellStore immediately return a *future* object on which the calling user-level thread can wait. InfinIO then transparently batches all requests at the network layer before actually sending them to TellStore. Likewise, responses from the storage are batched together before sending them back to the processing nodes. This batching greatly improves the overall performance as it cuts down the message rate on the Infiniband link, which would otherwise become the performance bottleneck (see Section 7.2.3).

6.2 Thread Model

TellStore runs three different kinds of threads as depicted in Figure 5: get/put threads, scan threads and a garbage collection thread. Whenever a client opens a connection, one of the get/put threads accepts it and takes on the sole responsibility for processing incoming request issued by that particular client. Whenever the client issues a *scan* request, the get/put thread places the request on a queue to be processed by the dedicated scan threads. To guarantee a consistent throughput for scans and get/put operations, TellStore only uses lock-free data structures.

One of the scan threads has the role of the scan coordinator. The scan coordinator consumes all queued scan requests and bundles them into a single, shared scan. The coordinator partitions the storage engine's set of pages and distributes them equally among the scan threads. All the scan threads (including the coordinator) then process their partition in parallel independently. (Partial) scan results are directly written into the client's memory using RDMA. With this threading model, TellStore can flexibly provision resources for *get/put* and *scan* requests depending on the expected workload.

6.3 Data Indexing

Sections 4 and 5 describe how TellStore organizes and processes records. Both approaches use a lock-free hash table to index records stored inside a single node. To locate keys across nodes, TellStore implements a distributed hash table similar to Chord [44]. The choice of using a hash table, however, is orthogonal to the question on how to implement fast scans on a KVS, as is the decision to use a DHT. The same techniques can be used for range partitioning.

For range partitioning, one would typically use a clustered index like a B-tree or a LSM to index data within pages. However, this typically makes get/put operations more expensive. To support range queries, Tell uses a lock-free B-tree that is solely implemented in the processing layer as described in [30].

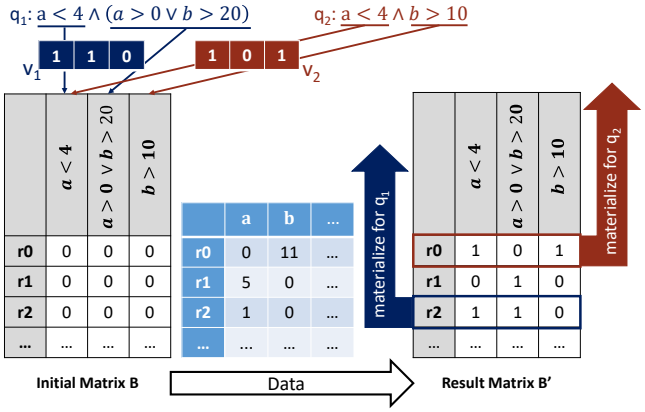


Figure 6: Predicate Evaluation and Result Materialization

6.4 Predicate Pushdown

In order to speed up analytical queries, TellStore allows the client to define selections, projections and simple aggregations (i.e. min, max, sum and count) on individual columns. When transforming a batch of scan requests into a single shared scan, TellStore uses just-in-time compilation with LLVM to get highly-tuned machine code. In essence, TellStore combines the batch-oriented shared scan technique of [22] with LLVM code generation described in [33, 27].

We carefully considered which part of the scan logic to delegate to the LLVM compiler. To this end, TellStore generates optimal LLVM-IR to evaluate the shared selection, projection and aggregation predicates. We made this decision, rather than generating C++ code or simple LLVM-IR and letting LLVM optimize the query code, as LLVM's optimization passes tend to be costly and ended up dominating overall execution time. By generating optimal LLVM-IR upfront, we were able to reduce compilation times from 100ms to less than 20ms for a batch of queries.

The generated code is highly efficient for a number of reasons: First, if queries share a common sub-expression (e.g. $x < 4$), this expression is evaluated only once. Second, for TellStore-Col, vector instructions are generated to evaluate an expression on several records in parallel. Third, the code generation takes the table schema into account and performs memory operations with static offsets, thereby facilitating prefetching and increasing CPU utilization.

TellStore requires all selection predicates of scans to be in conjunctive normal form (CNF). As shown in Figure 6, scanning a page proceeds in two steps: First, a (zero-initialized) bit matrix B is allocated, whose columns represent the unique OR-predicates and whose rows represent the records of the page. The thread then scans the page and evaluates all predicates, thereby setting the bits in the matrix. In the second step, a bit vector v_q is created for every query q involved in the shared scan. Bit i is set if the query involves the i-th predicate. This way, the records that match a particular query q can be computed as the bit-wise expression $r|{\sim}v_q$ with r the row that represents the record in B. When all matching records are identified, they are materialized and written into a local buffer. As soon as this buffer is full, it is copied back into the client's memory using a RDMA write.

Snapshot Isolation can be implemented in a straight-forward way: For each query a predicate is added, matching the *valid-from* and *valid-to* fields associated with each record against the timestamp of the query. These predicates are then processed as part of the normal query expression.

7. RESULTS

This section presents the results of performance experiments conducted with an extension of the YCSB benchmark [11] and an industrial workload from the telecommunications industry. The experiments studied the tradeoffs of the three TellStore variants and several popular KVS.

7.1 Experimental Setup and Methodology

We ran all experiments on a small cluster of 12 machines. Each machine is equipped with two quad core Intel Xeon E5-2609 2.4 GHz processors (two NUMA units), 128 GB DDR3-RAM and a 256 GB Samsung Pro SSD. Each NUMA unit has direct connectivity to half of the main memory. Furthermore the machines are equipped with a 10 GbE Ethernet adapter and a Mellanox Connect X-3 Infiniband card, installed at NUMA region 0.

All the KVS we benchmarked are NUMA-unaware. In order to get best results, we ran every process on only one NUMA unit. Therefore, throughout this section, the term *node* refers to a NUMA unit which can use half of a machine's resources. *Storage nodes* always ran on NUMA region 0 so that they had fast access to the Infiniband card, while *processing nodes* ran on both regions.

The system under test for all experiments is a KVS. Whenever we compare TellStore to other popular KVS, we use the *TellStore-Col* variant for TellStore. In all experiments, we also use Kudu as a baseline. We chose Kudu because it is the only KVS that can provide a reasonable scan performance (Table 1 in the Introduction).

To make sure that the load generation does not become a bottleneck, we used three times as many processing nodes as storage nodes. For all our measurements, we first populated the data and then ran the experiment for seven minutes. We ignored the results from the first and the last minute to factor out warm-up and cool-down effects.

We did a considerable effort to benchmark all KVS at their best configuration. In order to achieve a fair comparison between disk-based and in-memory systems, we used RAM disks and configured the buffer pool so that all data was resident in main-memory for all systems. For Kudu, we collaborated closely with the developers in order to make sure that we had the best possible Kudu configuration. The benchmarks for HBase and Cassandra were implemented in Java using the corresponding client libraries. For *RAMCloud* and *Kudu*, we implemented the benchmarks in C++ using the systems' native libraries. We used *multi-put* and *multi-get* operations in RAMCloud whenever possible and projection and selection push-down in Kudu. TellStore was benchmarked with a shared library that incorporates the native TellStore client library and allowed to execute *get*, *put*, and *scan* requests in a transactional context as well as manage secondary indexes (see [36] for details). For TellStore and Kudu, we batched several get/put request into one single transaction. Kudu does not support ACID transactions but supports *sessions* which have weaker properties. For TellStore, a batch size of 200 proved to be useful, while a good batch size for Kudu sessions was 50. We turned off replication for all KVS, except for HBase where this is not possible, which is why we used three storage nodes (HDFS data nodes) instead of only one for that particular case.

The main result of our experiments is that TellStore is a competitive KVS in delivering a high throughput for get/put requests. It also provides an order of magnitude better scan performance than all other KVS we tested against. Furthermore, TellStore's scan performance does not deteriorate when a moderately-sized get/put workload is executed in parallel. It is not surprising that TellStore-Col provides the lowest scan latencies. It is also able to sustain a high get/put load with a moderate number of update requests. Only for workloads with high update rates, TellStore-Log wins.

7.2 YCSB# Results

7.2.1 Benchmark Description

The Yahoo! Cloud Serving Benchmark (YCSB) [11] is a popular benchmark for KVS and cloud service providers. YCSB, however, does not include analytical queries or any kind of bulk operations. This is why we extended the benchmark in order to test these additional capabilities of a KVS. This new benchmark, called YCSB#, makes the following modifications: First, it extends the schema to include variable-size columns, and second, it introduces three new queries that involve scans of the data.

The new schema consists of an 8-byte key, named P, and tuples that consist of eight fixed-size values, named A to H. We used the following data types: 2-byte, 4-byte, and 8-byte integers as well as 8-byte double-precision float. Each of these types appears twice. The two variable-size fields, I and J, are short strings of a variable length between 12 and 16 characters. The three queries are:

- *Query 1:* A simple aggregation on the first floating point column to calculate the maximum value:
  ```
  SELECT max(B) FROM main_table
  ```
- *Query 2:* The same aggregation as Query 1, but with an additional selection on a second floating point column and selectivity of about 50%:
  ```
  SELECT max(B) FROM main_table
  WHERE H > 0 and H < 0.5
  ```
- *Query 3:* A selection with approximately 10% selectivity:
  ```
  SELECT * FROM main_table
  WHERE F > 0 and F < 26
  ```

The benchmark also defines a scaling factor that dictates the number of tuples in the database. Throughout our experiments, we used a scaling factor of 50, which corresponds to a test set of 50 million tuples. With larger databases, the running times of some KVS (e.g., Cassandra and HBase) for the three queries became prohibitively long.

7.2.2 Exp 1: Get/Put Workload

In the first experiment, we ran two workloads of the traditional YCSB benchmark (no bulk queries). We tested two different workloads: get/put and get-only. While the *get-only* workload solely consists of get requests, the *get/put* workload has 50% update requests, of which one third are inserts, one third updates (puts), and one third deletes. Balancing the number of inserts and deletes ensures that the size of the database stays constant, which is important for experiments that involve scans.

Figure 7 shows the throughput results (number of requests per second) of TellStore-Col and all other KVS. It becomes clear that TellStore is competitive for these traditional KVS workloads. Only RAMCloud outperforms TellStore in this experiment. RAMCloud is a distributed in-memory KVS that was highly tuned and specifically designed for exactly these workloads and can be seen as an *upper bound* for the best possible performance that can be achieved for these workloads. TellStore is within 50% throughput while all other KVS are far behind. The reason is that these systems do not take advantage of the latest, best of breed techniques such as lock-free data structures, batching, and asynchronous communication as described in Section 2.

One interesting result of Figure 7 is that all KVS scale linearly with the number of machines (storage and processing nodes). Only RAMCloud has slight scalability issues in the get/put workload. We believe that this flaw might be caused by a sub-optimal implementation of garbage collection in RAMCloud. Even though we could not measure it, we believe that TellStore would easily scale way beyond 12 machines (24 nodes). In our experience, however, databases that require more than 24 nodes are rare; beyond that point most databases can be sharded to achieve further scalability if the scalability of the KVS system becomes problematic.

Figure 8 shows the performance of the three TellStore variants for this experiment, using Kudu as a baseline. While the two row-based storage engines (TellStore-Log and TellStore-Row) outperform TellStore-Col, the difference is smaller than expected. For update operations, the write-optimized (row-oriented) log in the delta helps TellStore-Col. For get operations, TellStore-Col has higher cost to materialize records from columns, but this extra cost is comparably small to the cost of all other tasks. Thus, TellStore-Col's performance is within 10-20% compared to TellStore-Row and TellStore-Log, even in this worst case for TellStore-Col.

7.2.3 Exp 2: Batching

As explained in Section 6.1, the processing layer uses batching to improve throughput at the cost of increased latency. Whenever a transaction issues a request, this request is buffered within InfinIO and sent to TellStore whenever the buffer is full. While a transaction is waiting for a result, InfinIO schedules other transactions to fill up the buffer. Batching is important, as get/put messages are small and the message rate is limited on any network.

Figure 9 shows the sensitivity of the throughput depending on the batch size (which is determined by the size of the batch buffer) and for the same workload as shown in Figures 7a and 8a. Obviously, the bigger the batch size, the better the throughput for all three variants. However, the effects are not significant which is good news and shows that it is not critical to get this setting right. We used a batch size of 16 in all other experiments because it is a good compromise. The latency for transactions of 200 operations varied from 11.1 msecs (batch size of 1) to 9.29 msecs (batch size of 64 operations) for TellStore-Col.

7.2.4 Exp 3: Scans

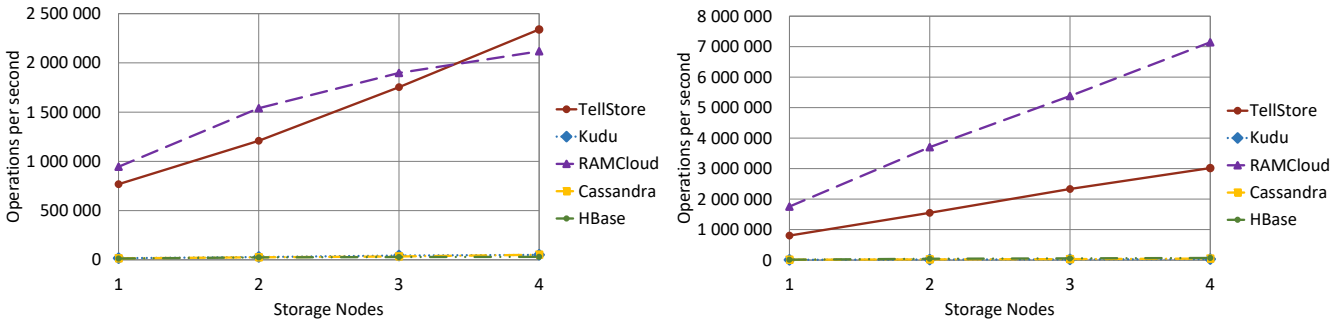
In order to demonstrate the raw scan performance, we ran the YCSB# queries in isolation without a concurrent get/put load. Figure 10 shows the results for Kudu and the three TellStore variants. We do not show the response times for Query 2 as they are nearly identical to Query 1. As shown in the introduction (Table 1), all other KVS (RAMCloud, HBase, Cassandra) are not competitive for queries involving scan operations.

As one would expect, TellStore-Col has the lowest response time for Query 1. For Query 3, however, TellStore-Log is slightly faster. Query 3 has no projections, thereby eliminating the advantages of a columnar layout. Furthermore, this read-only workload is the best case for scans in TellStore-Log as the scan does not have to perform garbage collection and is not affected by data fragmentation.

7.2.5 Exp 4: Mixed Workload

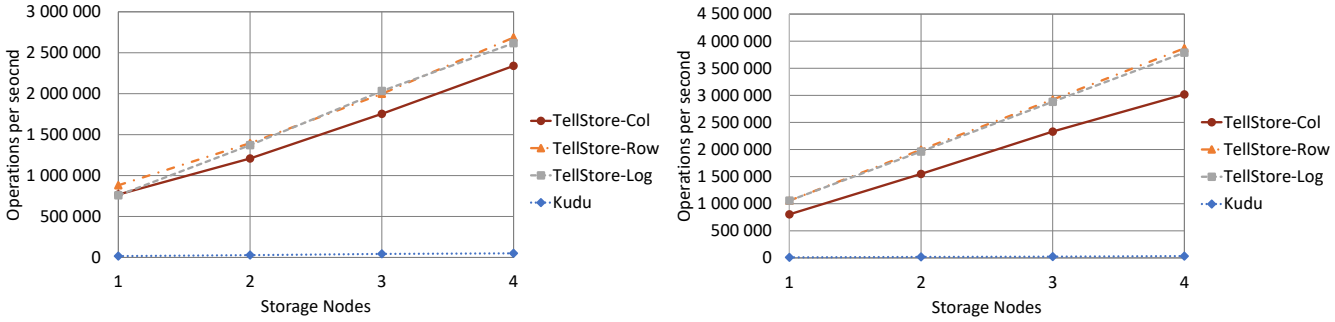
To see how well a scan performs while the system handles get/put requests in parallel, we ran the YCSB# queries concurrently to the get/put workload studied in Section 7.2.2. Figure 11a shows the average response time of Query 1 of the extended YCSB benchmark for a fixed get/put workload of 35,000 operations per second; this is the maximum workload that Kudu could sustain. Figure 11b and 11c show the response times of Query 1 for the three TellStore approaches if we scale the get/put workload beyond these 35,000 operations per second. For brevity, we only show the results for Query 1 as the effects are the same for the other two queries.

Focusing on Figure 11a, all three TellStore variants perform much better than Kudu. Even though Kudu is a column store which



(a) Get/Put Workload (b) Get-only workload

Figure 7: Exp 1, Throughput: YCSB, Various KVS, Vary Storage Nodes



(a) Get/Put Workload (b) Get-only workload

Figure 8: Exp 1, Throughput: YCSB, TellStore Variants and Kudu, Vary Storage Nodes

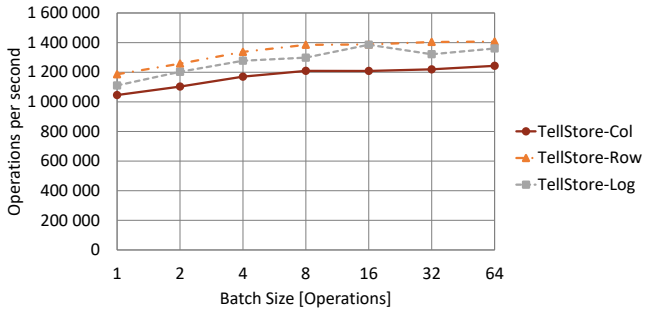


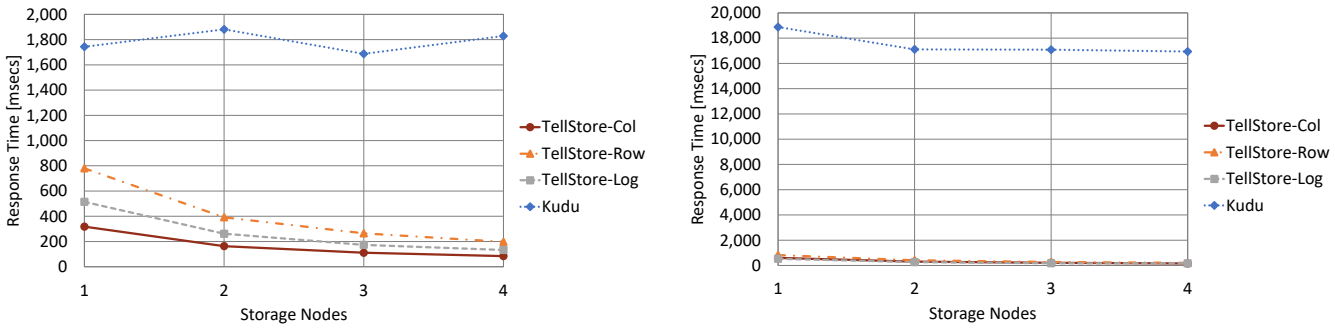
Figure 9: Exp 2, Throughput: YCSB, Vary Batch, 2 Storage Nodes

is particularly favorable for Query 1, its overall performance is not competitive as it makes the wrong compromises to trade off good get/put and scan performance. Of the three TellStore variants, TellStore-Col wins whereas TellStore-Row performs worst. TellStore-Log does fairly well because the update workload is still moderate which allows it to keep locality in the log fairly high.

Figure 11b shows the response times for Query 1, thereby scaling the number of concurrent *get* requests without any updates. In the absence of updates, the response time of Query 1 is constant and independent of the concurrent *get* workload. Scans and get requests are scheduled on different cores in a storage node. Furthermore, the scan performance is not impacted by garbage collection, data stored in the delta, or stale versions of records.

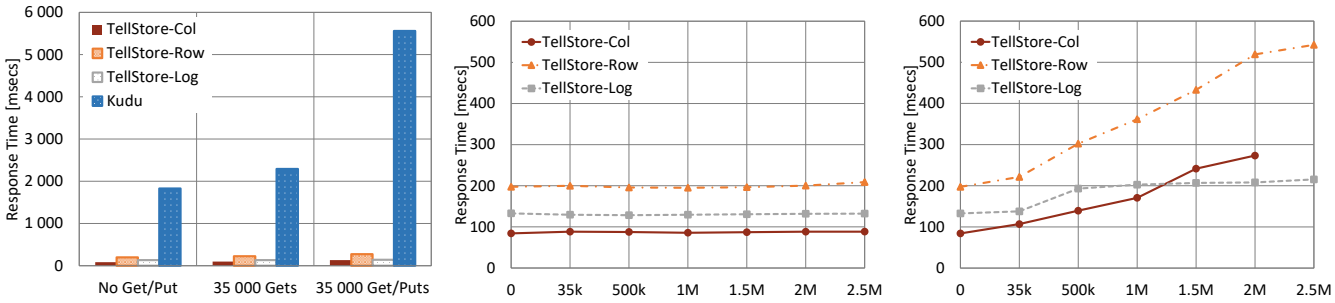
Figure 11c shows the response times for Query 1 with varying get/put workload that involves 50% updates as in Experiment 1. TellStore-Log and TellStore-Row can sustain a load of up to 2.5 Mio. concurrent get/put requests; TellStore-Col of about 2 Mio. get/put requests. In all three approaches, the performance of the scans is affected by a raising get/put workload. TellStore-Log, however, is the most robust approach. In TellStore-Log, the average response time of a scan increases sharply at 500,000 concurrent get/put requests per second. At this point, garbage collection rewrites essentially the entire log with every scan as every page is affected by an update. Note that the updates are distributed uniformly over the entire database in YCSB; with skew, less pages of the log may be affected. The scan performance of TellStore-Col and TellStore-Row is heavily impacted by concurrent updates: With an increasing update load, there is more expensive pointer-chasing to the delta to look up the latest version of a record. With very high update workloads, almost all records are fetched from the row-store delta, so that TellStore-Col has no advantage anymore and is outperformed by TellStore-Log.

Figure 12 contains a box plot that shows the 1%, 25%, 50% (median), 75%, and 99% percentile response times of Query 1 with a concurrent get/put workload of 1 Mio. requests per second. At 1 Mio. get/put requests, TellStore-Col still outperforms TellStore-Log, but Figure 12 confirms that TellStore-Log is more robust. The scan performance of TellStore-Col depends on the size of the delta; shortly *after* garbage collection, the delta is small and the scan is fast. Accordingly, the scan in TellStore-Col is slow shortly *before* garbage collection when the delta is large. The effects were similar for TellStore-Row. For TellStore-Log, the scan with piggy-backed garbage collection always has roughly the same response time.



(a) YCSB# Query 1 (b) YCSB# Query 3

Figure 10: Exp 3, Response Time: YCSB#, Vary Storage Nodes



(a) TellStore vs Kudu (b) TellStore, Scale Get Requests (c) TellStore, Scale Get/Put Requests

Figure 11: Exp 4, Response Time: YCSB# Query 1, 4 Storage Nodes

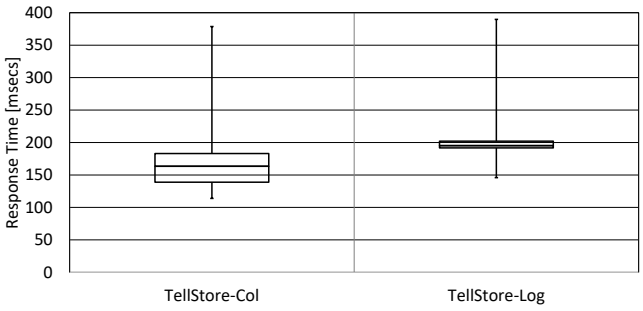


Figure 12: Exp 4, Response Time: YCSB# Query 1
4 Storage Nodes with 1 Mio. concurrent get/put requests

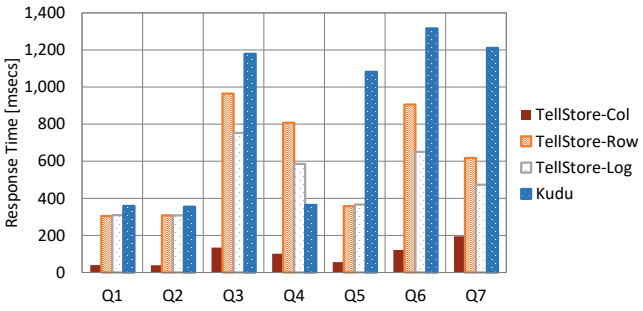


Figure 13: Exp 5, Resp. Time: Huawei-AIM Workload

7.3 Exp 5: Huawei-AIM Benchmark Results

To show that TellStore is able to perform more complex, interactive workloads, we also executed the Huawei-AIM benchmark [10]. This workload defines seven analytical queries with a concurrent get/put workload. In our implementation of this benchmark, we generated a get/put workload of 40,000 operations per second. As this is a low load for TellStore to handle, we did not observe any differences between the query response times in isolation and with concurrent get/put operations. Therefore, we simply show the response times for the workload with concurrent get/put in Figure 13.

For this workload, TellStore-Col outperforms all other implementations significantly. Furthermore, the difference between TellStore-Log and TellStore-Col is larger than in the experiments shown before. There are two reasons for that: First, this workload runs on a table with more than 500 columns. Therefore, the columnar layout has a much larger benefit on query performance if the query involves only a subset of the columns. Second, the queries are much more complex. As mentioned in Section 2, TellStore supports aggregation in the storage node, which turns out to be particularly important for this benchmark. The columnar format enables TellStore-Col to make use of vector instructions to efficiently filter out irrelevant records (Section 6.4). Kudu performs roughly as well as TellStore-Row and TellStore-Log: Kudu is also a column store and, thus, has the same advantage as TellStore-Col for this experiment. However, Kudu has other inefficiencies as shown in the experiments before.

8. RELATED WORK

This section re-iterates the most important lines of work relevant to this paper. Most techniques used in the various TellStore variants are known (Section 3) and the art is to combine them in the best possible way.

There are several systems that adopted the SQL-over-NoSQL architecture. Among them are *FoundationDB* [21], *Hyder* [7], *Tell* [30], and *AIM* [10]. While the first three were not built with analytics in mind, AIM supports read-only analytics and high update rates, but for a very specific use-case that does not require transactions, joins, or variable-size data fields. On the other side of the spectrum, there are the large-scale analytical systems like *Hadoop* [43], *Spark* [48], and *DB2/BLU* [39]. All of these systems have significant trouble with or no support for querying live data that is subject to frequent and fine-grained updates.

There exist systems with good support for mixed workloads on live data, e.g. *HyPer* [26], *HANA* [18], and *Hekaton / Apollo* [13]. What distinguishes TellStore from these systems is its ability to scale out in a distributed system whereas these systems can only scale up on a single machine.

As pointed out already, there exist a number of KVS (e.g. *RAMCloud* [41], *FaRM-KV* [14], *HBase* [20], *Cassandra* [28], *LevelDB* [23], and *RocksDB* [16]) all of which show good get/put performance, but have difficulties to process scans with a competitive performance. Another interesting line of related work are document stores, like *DocumentDB* [42] or *MongoDB* [32]. Like Cassandra, they offer some scans with secondary indexes, specifically tuned to the document-related use-cases. We expect that our work on TellStore is relevant to those systems, too.

9. CONCLUSION

This paper has shown that it is possible to build a KVS that supports efficient scans for analytics and a high get/put throughput for OLTP workloads. Our system, TellStore, has an integrated design that addresses the most important design questions in a holistic way. Furthermore, TellStore makes use of best practices (e.g., lock-free data structures) and makes careful compromises with regard to latency vs. throughput tradeoffs (e.g., batching) and time vs. space tradeoffs (e.g., the TellStore hash table implementation). TellStore uses a number of advanced implementation techniques to help remedy the effects of concurrent updates on scans; e.g., piggy-backing garbage collection on scans or copying old versions to the head of the log in order to improve locality of scans.

This paper studied three particularly promising design variants of TellStore and KVS in general. Of these three designs, TellStore-Col with a columnar layout and a delta-main approach delivered the overall best and most robust performance. However, TellStore-Log with a row-oriented layout and a log-structure approach showed competitive performance, too; in particular, for OLTP workloads. TellStore-Log has more predictable update performance, but it shows lower performance for analytical queries in which a columnar layout is particularly important.

There are several avenues for future work. TellStore currently does not feature high availability with replication. In order to address this requirement, we recently started building a replication feature into TellStore. Our first experiences make us confident that replication is orthogonal to all other aspects of a KVS and that adding replication does not change the main results and observations made in this paper. This observation has been confirmed by other, related work on replication for KVS; e.g., [20, 28, 41].

Running analytical workloads efficiently is a hard problem and having a fast scan in the storage only solves it half-way. In this work, we focused on queries that can efficiently be processed on a single processing instance, taking full advantage of TellStore's scan capacity. However, there exist other analytical queries that can only be efficiently executed in a distributed way. Examples of such distributed query processing systems include Spark [48] and Presto [15]. We implemented TellStore adapters for both of these systems [1] and ran the TPC-H benchmark queries. We found query response times with TellStore to be on par with other storage layer back-ends (Parquet and ORC). Extensive profiling on these platforms revealed that TellStore does not have an advantage because neither Spark nor Presto can make efficient use of TellStore's scan feature (shared scans and pushing down selections, projections, and aggregation into the storage layer). To take advantage of these features requires refactoring these systems. Conceptually, this work is straight-forward, but it involves a great deal of engineering effort. Another avenue for future work is to study less aggressive variants to carry out garbage collection in TellStore-Col and TellStore-Row. In the current implementation, garbage collection can result in high contention of the memory bus and in write amplification if the data does not fit into main memory. So, the performance of these approaches can be even further improved.

10. REFERENCES

[1] Tellstore open-source project. https://github.com/tellproject/tellstore.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 466–475, 2007.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[7] P. Bernstein, C. Reid, and S. Das. Hyder - a transactional record manager for shared flash. CIDR'11, pages 9–20, 2011.

[8] P. Boncz, T. Grust, M. Van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490, 2006.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[10] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 251–264, 2015.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[12] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2016.

[13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[14] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 54–70, 2015.

[15] Facebook. Presto. http://prestodb.io. May. 02, 2016.

[16] Facebook. RocksDB. http://rocksdb.org. May. 02, 2016.

[17] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[18] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[19] A. Foundation. Kudu. http://getkudu.io. May. 02, 2016.

[20] A. Foundation. HBase. http://hbase.apache.org/. May. 27, 2017.

[21] FoundationDB. https://foundationdb.com/. Feb. 07, 2015.

[22] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, Feb. 2012.

[23] Google. LevelDB. http://leveldb.org. May. 02, 2016.

[24] J. Gray and A. Reuter. *Transaction processing*. Morgan Kaufíann Publishers, 1993.

[25] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt. A comparison of c-store and row-store in a common framework. *University of Wisconsin-Madison, Tech. Rep. TR1570*, 2006.

[26] A. Kemper and T. Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[27] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, June 2014.

[28] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[29] J. J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *CIDR*, 2015.

[30] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 663–676, 2015.

[31] MemSQL. http://www.memsql.com/. May. 02, 2016.

[32] MongoDB. http://mongodb.com/. May. 27, 2017.

[33] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, June 2011.

[34] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[35] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[36] M. Pilman. *Tell: An Elastic Database System for mixed Workloads*. PhD thesis, ETH Zürich, Under Submission, 2017.

[37] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *Proceedings of the 19th International Conference on Distributed Computing*, pages 108–121, 2005.

[38] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[39] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[40] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[41] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 1–16, 2014.

[42] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *PVLDB*, 8(12):1668–1679, 2015.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2010.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, 2001.

[45] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564, 2005.

[46] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[47] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–17, 2010.

[49] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.

[50] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 723–734, 2007.