CaSSanDra: An SSD Boosted Key-Value Store

Prashanth Menon #, Tilmann Rabl #, Mohammad Sadoghi *, Hans-Arno Jacobsen #
# *Middleware Systems Research Group, University of Toronto*
* *IBM T.J. Watson Research Center*

***Abstract*—With the ever growing size and complexity of enterprise systems there is a pressing need for more detailed application performance management. Due to the high data rates, traditional database technology cannot sustain the required performance. Alternatives are the more lightweight and, thus, more performant key-value stores. However, these systems tend to sacrifice read performance in order to obtain the desired write throughput by avoiding random disk access in favor of fast sequential accesses.**

With the advent of SSDs, built upon the philosophy of no moving parts, the boundary between sequential vs. random access is now becoming blurred. This provides a unique opportunity to extend the storage memory hierarchy using SSDs in key-value stores. In this paper, we extensively evaluate the benefits of using SSDs in commercialized key-value stores. In particular, we investigate the performance of hybrid SSD-HDD systems and demonstrate the benefits of our SSD caching and our novel dynamic schema model.

I. Introduction

Many big data challenges are characterized not only by a very large volume of data that has to be processed but also by a high data production rate, i.e., high velocity [1]. In modern monitoring applications, many thousands of sensors will produce a multitude of readings that have to be stored at a high pace but have to also be readily available for continuous query processing. Examples of such applications include traffic monitoring, smart grid applications and application performance management. In application performance management, for example, hundreds to thousands of servers that are connected in an enterprise system will be monitored in order to determine the bottlenecks, sources of errors, and inefficiencies that often stem from highly complex interactions of various services [2]. This requires the collection of monitoring data to enable the tracing of the path that individual transactions take through the system. The common denominator among these applications is the need for write-optimized data storage such as key-value stores. Current storage approaches either try to exploit the speed of memory-based solutions, thus, drastically limiting the amount of data that can be stored or use disk-based approaches and, thus, only allow sampling the required information to cope with high velocity data.

Building on the success of key-value stores, in this paper, new optimized storage approaches for high-velocity data will be explored. The key point is the efficient use of modern hardware, especially, modern storage technology such as solid-state drives (SSDs). These new technologies have highly improved performance in comparison to traditional hardware. However, classical data structures and algorithms can not directly be applied due to the different characteristics of these devices. Also, the high cost of the new technology makes their exclusive use uneconomical in many cases. Therefore, hybrid storage approaches are being explored in both industry and academia, in which modern and traditional technologies are co-allocated to form a storage memory hierarchy (e.g., [3], [4], [5]). In contrast, our proposed techniques are geared towards key-value stores instead of relational database systems.

In this paper, traditional data structures and algorithms in key-value store systems are revisited in the light of new technology, especially, in the light of solid-state drives (SSDs). Log-structured storage techniques – at the core of many key-value stores – that have proven to work well for high insertion rates are re-examined for efficient use with modern hardware such as SSDs. Because of the different characteristics of SSDs, such as fast random access, uneven write and delete speed, high price, limited lifetime, compared to traditional disk, hybrid approaches will be explored that highly improve the performance in comparison to disk-only solutions while leveling off the downsides of SSD technology.

In particular, we enhance the commercialized Cassandra key-value store engine by expanding its storage memory hierarchy. First, we extend Cassandra's row cache with SSDs in order to substantially improve the query performance by serving the host set of data with at most a single SSD I/O, which otherwise requires a number of random HDD I/Os to construct the record by consolidating different versions of records spread across many Cassandra tables (i.e., SSTables). Second, we propose a novel dynamic schema model, also SSD-resident for fast random I/O access, in order to decouple data and meta-data (schema), a decoupling that is well established in relational database, but ignored in key-value store systems. This decoupling avoids redundantly storing meta-data information on disk, which also results in noise when compressing data, while by using SSDs, provides a fast access for persisting and retrieving schema information from SSDs.

The rest of the paper is organized as follows. In Section II, we give an overview of the Cassandra key-value store and the internal I/O processes. In Section III, we give details on the internal architecture of SSDs and explore possibilities of introducing SSDs in the key-value store hierarchy. In Section IV, we introduce our extended row cache technique that extends Cassandra's row cache on SSD. Section V presents the dynamic schema technique, which extracts the repeatedly stored schema information from the rows and stores them in a dynamic dictionary. Section VI presents our extensive performance evaluation. In Section VII, we present related work before concluding with future work in Section VIII.

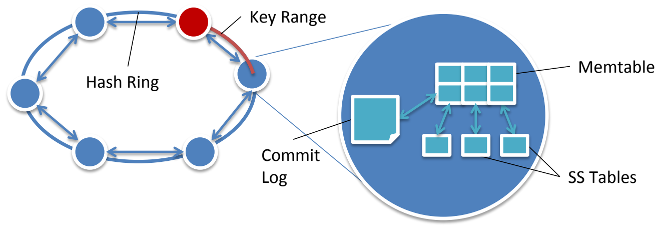


Fig. 1. Cassandra Architecture

II. Cassandra

For this paper, we modify the Apache Cassandra engine since it has proven to be highly performant under different workloads [2]. Cassandra is a distributed key-value store initially developed at Facebook [6]. It was designed to handle large amounts of data spread across many commodity servers. Cassandra provides high availability through a symmetric architecture that contains no single point of failure and replicates data across nodes.

Cassandra's architecture is a combination of Google's BigTable [7] and Amazon's Dynamo [8]. Like in Dynamo's architecture, all Cassandra nodes form a ring that partitions the key space using consistent hashing (see Figure 1). This is know as distributed hash table (DHT). The data model and single node architecture are mainly based on BigTable as is its terminology. Cassandra can be classified as an extensible row store since it can store a variable number of attributes per row [9]. Each row is accessible through a globally unique key. Although columns can differ per row, columns are grouped into more static *column families*. These are treated like tables in a relational database. Each column family will be stored in separate files. In order to allow the level of flexibility of a different schema per row, Cassandra stores metadata with each value. The metadata contains the column name as well as a timestamp for versioning.

Like BigTable, Cassandra has an in-memory storage structure that is called Memtable, one instance per column family. The Memtable acts as a write cache that allows for fast sequential writes to disk. Data on disk is stored in immutable Sorted String Tables (SSTable). SSTables consist of three structures, a key index, a bloom filter and a data file. The key index points to the rows in the SSTable, the bloom filter enables checking for the existence of keys in the table. Due to the limited size of the bloom filter it is also cached in memory. The data file is ordered for faster scanning and merging.

For consistency and fault tolerance, all updates are first written to a sequential log after which they can be confirmed. An overview of the read and write path is given in Figure 2. In addition to the Memtable, Cassandra provides optional row caches and key cache (not shown in the figure). The row cache stores a consolidated, up-to-date version of a row, while the key cache acts as an index to the SSTables. If these are used, write operations have to keep them updated. It has to be noted that only previously accessed rows are cached in Cassandra in both caches. As a result, new rows will only be written to the Memtable but not the cache.

The read path starts by looking up the caches, first the row cache then the key cache. Rows that reside in the row cache can be completely served from memory since they are always complete and up-to-date. Rows are added to the row cache whenever a row is accessed and replaced in *LRU* fashion. For workloads with small sets of hot data that is frequently accessed, the row cache can improve the systems performance considerably. Rows that do not reside in the row cache can be distributed across multiple SSTables and Memtables, thus creating multiple disk accesses for a single read. In fact, all versions of a row have to be considered because of the flexible schema. This is because an older version might have an additional column that does not exist in later versions. Also, deletes are stored as tombstones and have to be read from the SSTables as well.

In Cassandra, updates and inserts are processed using the same operation. Both are written to the log and the Memtable. If an inserted or updated records key exists in the Memtable, the value will be updated. Also the row and key cache will be updated. Whenever a Memtable is full, it is converted and flushed to disk as an SSTable. Rows can have arbitrary schemas and new attributes can be added at any time, which means that parts of a single row can be stored in multiple SSTables and Memtables. To reduce the number of SSTables that have to be considered if a row is retrieved a periodic *compaction* process merges the SSTables. In this process multiple versions of rows are consolidated and rows that are marked as deleted are completely removed.

III. Introducing SSDs in Cassandra

Today's SSDs are mostly built from NAND flash memory [10]. This type of memory has some specific characteristics that have to be considered in order to get the best performance and durability [11]. In the following discussion, we use Flash and NAND interchangeably and will only consider NAND-based SSDs. NAND chips have an asymmetric read and write performance, which is due to the more complex implementation of write operations. In contrast to disk, NAND memory cannot be overwritten, but has to be erased before it can be written again. While read and write operations are performed on 4 KB - 8 KB pages, erase operations are done in so-called erase blocks, which contain up to 256 pages.

NAND chips, especially the cheaper and higher density *multilevel cell* chips (MLC) have a very limited number of write-erase cycles, which is in the order of 3000 cycles per block for MLC flash. At the same time, flash cells loose data over time, which means that data has to be rewritten in order not to get lost.

To overcome these issues, the embedded controller in a flash drive tries to evenly distribute the data across the blocks to level the wearing of the blocks and reduce the amount of erase operations. To overcome the high rate of failures it implements error correction codes. These techniques make the write operations more involved than read operations and can lead to the *write amplification effect*, where a write operation causes a series of erase and write operations effectively reducing throughput to a fraction of the baseline throughput.

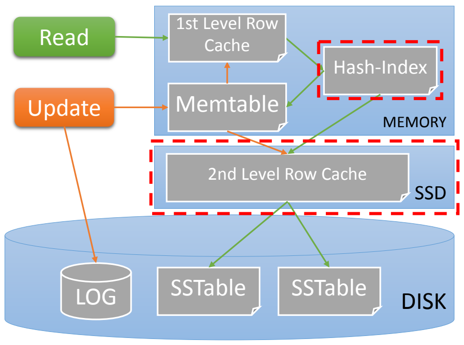


Fig. 2. Row Cache Read and Write Path (Extensions Highlighted)

The controller or *flash transition layer* (FTL) hides all this effort from the OS and the application layer. However, a conscious use of the SSD can ease the pressure on the controller. While random writes increase the frequency of write amplification events, block size-aligned sequential writes yield higher performance and reduce wearing of cells [11].

Due to its scalable architecture, Cassandra's performance can be adapted by adding new nodes [2]. Adding SSDs to nodes or replacing HDDs with SSDs is complementary to this. If throughput is the major concern and the stored data is relatively small, using SSDs will be most efficient due to the smaller unit size. Other than replacing HDD completely by SSD, it is also possible to use it as an additional cache layer [12]. This enables a hybrid architecture, where some data resides on HDD and some on SSD. The benefit of this solution is a more flexible trade-off between performance and cost, and an architecture that can take advantage of the different characteristics of HDD and SSD. In the following, we present two strategies that efficiently make use of SSDs in a hybrid storage architecture.

IV. Extending the Row Cache on SSDs

Cassandra's row cache stores fully compacted rows in memory. This improves performance dramatically, if the workload features a hot data set that fits into memory. Storing only compacted rows is not possible using disk, since it will result in random I/Os for all write requests. If the hot set does not fit in memory the performance benefit of the row cache is quickly consumed by the cost of much slower disk accesses. In order to introduce SSDs into Cassandra's storage hierarchy, we extend the row cache beyond memory onto the SSD.

The extended row cache is modelled as an append-only cache file with an associated hash-based index that is memory resident. The index maps row keys to 64 bit offsets into a logical cache file. A query into the SSD cache is executed by first consulting the index to determine the position of the compacted row in the cache file, followed by a single SSD seek to retrieve the row. An important observation here is that concurrent reads never block one another except inevitably in I/O access to the SSD itself. Append-only operation is achieved by enforcing a sequential order to write requests. We eliminate the potential bottleneck of using a single-writer policy by buffering writes into a 16KB in-memory buffer that is flushed to SSD only when full. Having a write buffer also affords us the opportunity to optimize certain read queries by directly accessing the write-buffer rather than reaching into the SSD. Deletes from the row cache are handled by simply removing the row's key from the hash-index. A background garbage-collection task is executed periodically to remove deleted entries from the SSD cache files. This design takes advantage of the peculiar behaviour of SSDs discussed above.

We now describe the physical implementation in detail. The logical cache file described above is actually composed of a collection of contiguous *segment files*, each of a configurable size and preallocated on creation. The most recently created segment is termed *active* and is the only segment that accepts writes, whereas any segment can service reads. Segment files are created as part of a write if the write operation overflows the currently active segment. A 64-bit offset in this model is analogous to a virtual address; given a 64-bit offset, we use the the higher-order 32 bits to determine which segment file the row's data resides and the lower-order 32 bits as the offset into the segment. We perform this segmentation for practical purposes, but more importantly because it makes the task of garbage-collecting deleted rows a relatively simple one. We track the amount of garbage in each segment file, where garbage is defined as the percentage of the segment file that contains deleted rows. The garbage-collection task uses this metric to select segments to compact that will yield the largest reclamation of space.

The row cache in Cassandra has been modified to chain together the in-memory cache and the SSD cache as shown in Figure 2. Read requests against the row cache flow along the chain, first querying the in-memory cache then the SSD cache. A read request stops along the chain at the first point its request is successful. The in-memory row cache is implemented as a size-bounded LRU cache, where the oldest and largest entries are ejected when a capacity threshold is reached. With the modified row cache chain, ejected entries from the in-memory cache cascade into the SSD cache asynchronously, thereby incurring zero additional latency during cache-writes.

V. Dynamic Schema on SSDs

Every row in the data component of Cassandra's SSTable is made up of a row key, an optional column index, and a series of column name, column value and timestamp tuples. This provides a schema-less data model that is flexible and easy to work with. For many use cases, however, column names rarely differ among rows in a table and are therefore duplicated unnecessarily. This duplication not only increases latency at read-time, since more data needs to be read from disk, but also wastes space on disk. This overhead can be reduced by extracting the schema from the rows and storing them separately in a schema catalogue. If the catalogue is on disk, however, it will introduce random I/Os and, thus, deteriorate performance. Storing the catalogue only in memory is not an option because of the potential size of the catalogue and data loss in cases of failures. Since SSDs provide fast random access and stable storage, it is the ideal medium to store frequently used and updated metadata. We can therefore

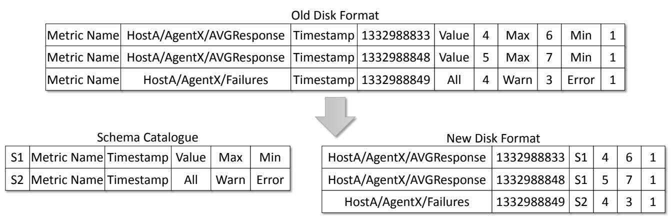

Old Disk Format
Metric Name | HostA/AgentX/AVGResponse | Timestamp | 1332988833 | Value | 4 | Max | 6 | Min | 1
Metric Name | HostA/AgentX/AVGResponse | Timestamp | 1332988848 | Value | 5 | Max | 7 | Min | 1
Metric Name | HostA/AgentX/Failures | Timestamp | 1332988849 | All | 4 | Warn | 3 | Error | 1
Schema Catalogue
S1 | Metric Name | Timestamp | Value | Max | Min
S2 | Metric Name | Timestamp | All | Warn | Error
New Disk Format
HostA/AgentX/AVGResponse | 1332988833 | S1 | 4 | 6 | 1
HostA/AgentX/AVGResponse | 1332988848 | S1 | 5 | 7 | 1
HostA/AgentX/Failures | 1332988849 | S2 | 4 | 3 | 1

Fig. 3. Redundant vs. dynamic schema

maintain the flexibility of a schema-less data model with the benefits of a schema-ed one, along with reduced disk footprint.

The schema catalogue is a global data structure within a single running Cassandra instance. Conceptually, the catalogue maintains a mapping between schema identifiers (IDs) and schema definitions where IDs are monotonically increasing 32-bit unsigned integer values and a schema definition is simply a sorted set of column names each a string value. This is illustrated in Figure 3. The implementation of the schema catalogue resembles what was described for our SSD row cache in the previous section. That is, the physical layout of the schema file consist of a collection of segment files, each storing a contiguous subset of schema ID and schema tuples, along with an associated memory-resident hash-index.

The dynamic schema catalogue exposes operations for insertion and retrieval of schema definitions. When Cassandra is preparing to serialize a row to disk, it first consults the schema catalogue to obtain a schema ID for the row by supplying the sorted set of columns within the row. If the catalogue discovers that the schema definition has already been assigned an ID, this ID is returned immediately. Otherwise, a new schema ID is generated, the schema ID and definition are persisted to SSD and the ID is returned to the caller. Cassandra will serialize the schema ID locally with the row and will exclude serialization of the column index and any of the column names. During a read operation, the schema ID will be deserialized first at which point the schema catalogue will be queried to efficiently retrieve the schema definition from SSD. With the schema definition available prior to deserialization of the columns, Cassandra knows precisely how the column values are laid out on disk and can choose to retrieve either all or a subset of the columns depending on the type of read operation.

If a table's rows are expected to be largely homogeneous, that is rows differ very little in the columns they use, then the proposed schema extraction technique can yield substantial savings in space and reduce the latency of read operations. If, however, the table's data is predicted to be highly variable in the columns that are inserted, extracting the schema from the table can be detrimental. For example, a table with 10 columns can have up to 2^{10} possible unique schemas. It is obvious that multiple high-variability tables all using a schema catalogue can quickly exhaust available schema IDs. Note that the current implementation makes no attempt to reclaim schema IDs if their associated schemas are no longer in use.

Configuration	# of clients	# of threads/client	Location of Data	Location of Commit Log
C1	1	2	RAID	RAID
C2	1	2	RAID	SSD
C3	1	2	SSD	RAID
C4	1	2	SSD	SSD
C5	4	16	RAID	RAID
C6	4	16	SSD	SSD

TABLE I
EXPERIMENT CONFIGURATIONS

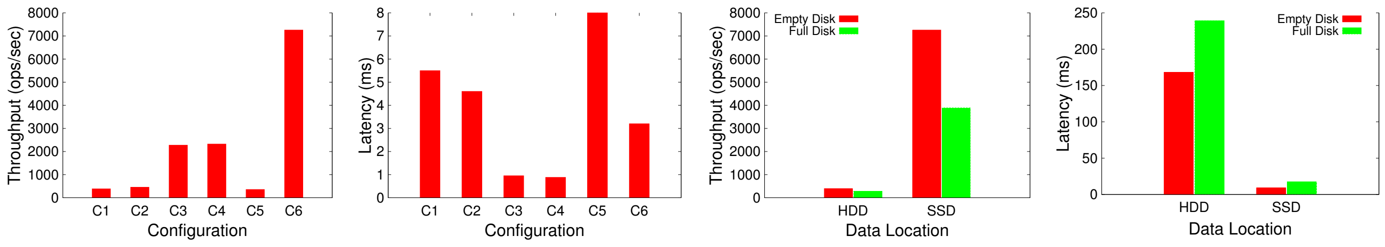
VI. EVALUATION

For our evaluation, we used the Yahoo! Cloud Serving Benchmark (YCSB) suite [13]. YCSB is composed of a data generation component and a workload generation component. For data generation, YCSB functions on records which are a collection of columns where each column has a name and value of fixed size. Each record is indexed by a 25 byte key and the user can configure the number of columns in the record along with the size of each column. The data loading phase will insert a configurable number of records into the data store. In the transactional phase, YCSB provides parameters to control concurrency, maximum execution time, statistical distributions of accessed keys and distributions for operations. YCSB offers standard CRUD (create, read, update delete) operations that makes it amenable to key-value stores.

In this section, we will compare the performance of a Cassandra installation that uses HDD versus an installation with SSD. For these experiments, we used a system that was equipped with an Intel Xeon X5450 eight-core CPU running at 3GHz, 16GB of RAM, four 1TB hard drives configured to create two RAID0 2TB hard drives and two 240GB Intel X520 SSDs. Since Cassandra uses two different basic disk-stored entities (SSTables and the commit log), there are 4 different configurations with HDDs and SSDs. We tested all configurations in a read-mostly environment (95% read, 5% write). The dataset consisted of 100 million rows, totalling 50GB data. We used a distribution where the last inserted row is accessed most frequently and all keys are accessed according to a zipfian distribution; we shall henceforth refer to this type of request distribution as a *latest* distribution.

In Figure 4(a), the throughput of all configurations outlined in Table I can be seen. As expected, putting the data on SSD has a dramatic performance benefit. The same trend can be seen for the latency, as show in Figure 4(b). It is important to note that configurations C1 through C4 use a YCSB client setup that is suboptimal for SSDs. These configurations were chosen because they saturate I/O access to our HDDs. Configurations C5 and C6 are optimized to operate against the SSD by using more clients each running a higher number of threads to achieve a much higher throughput and a latency that is average for the class of SSDs we used.

Figures 4(a) and 4(b) show that placing everything on SSD does not make sense for Cassandra. Storing the commit log to SSD (C2) offers a minimal boost to throughput and latency versus writing the data to SSD (C3). Since not all data is accessed with the same frequency, it is more efficient to selectively store frequently accessed data on SSDs and rely



(a) HDD vs SSD Throughput (b) HDD vs SDD Latency (c) 99% Fill HDD vs SDD Throughput (d) 99% Fill HDD vs SDD Latency

Fig. 4. Throughput/Latency Results for HDD vs SSD and Disk Full vs Disk Empty

on HDD for the bulk of data that is infrequently accessed. Another reason to do this is the fact that SSD performance degrades with higher fill ratios. As seen in Figure 4(c), the performance of a highly filled SSD degrades much worse than the performance of a highly filled disk. It has to be noted that the workload in this case is still read heavy, for write heavy workloads even worse degradations will be experienced.

When evaluating our extended SSD row cache, the size of the data set was 100 million records, where each record had five columns having a size of 75 bytes. The total size of the data on disk after load averaged 50GB. Our evaluation process was broken down into four phases: data loading, data fragmentation, Memtable flush, bufferpool warmup, and transactional workload phases. The fragmentation phase attempts to spread the columns of a row across multiple SSTables to illustrate the effect of read amplification on LSM-based storage systems. In the fragmentation phase, we used a latest request distribution with 10% of operations being read and the remaining 90% of operations updating anywhere between one and all five columns. The warming phase also used a latest request distribution with read operations accounting for 99% of all operations. The warmup phase was run until either the cache was full or stored at most 10% of the total dataset. The transactional phase was run with a latest distribution (a zipfian distribution where the most recently entered keys are favoured). These experiments all used configuration C5 (refer to Table I), the optimal configuration for HDDs to provide a balanced evaluation.

When evaluating our dynamic schema model, we used a dataset consisting of 40 million records where each record consisted of between 5 and 10 columns, of 10 bytes. By default, YCSB does not vary the number of columns in a record during the loading phase. We modified YCSB to create a new varying-size record generator, which we plugged into the default data generator. Each run of the experiment created a different amount of data on disk, but we observed that the average total data size was between 6.5GB and 7GB. In all runs, we varied the read percentage for the experiments between 95%, 50% and 5% using configuration C6.

A. SSD Row Cache

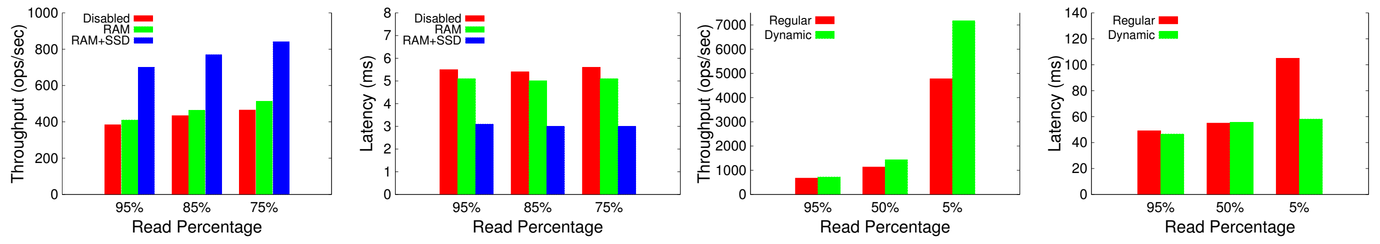
In Figure 5(a), the throughput of the two Cassandra instances can be seen for the three different workloads that were tested. For the 95% read-heavy workload, we see that the SSD-enabled row-cache provides an 85% improvement in throughput growing from 384 reads/sec to 710 reads/sec. This is because a larger portion of the hot data is cached on the SSD; in fact, our configuration enabled storing more than twice the amount of data than when using an in-memory cache alone, achieving a cache-hit ratio of more than 85%. When a read operation reaches the server for a row that does not reside in the off-heap memory cache, only a single SSD seek is required to fulfill the request. In addition, cached data is pre-compacted, meaning that at most one seek is required to fetch the row. We see the same effect in the remaining two workloads despite a lower proportion of reads. Cassandra is a write-optimized system meaning that in write-heavy scenarios, the efficacy of a cache is reduced. This is evidenced by the reduction in the cache-hit ratio from 72% in the workload with 85% reads to 60% in the 75%-read workload.

As seen in Figure 5(b), in the 95% read workload, the SSD-enabled row cache averaged a latency of 3ms while the in-memory cache managed a read latency of 5.6ms, a 46% improvement. As the proportion of reads is reduced from 85% to 75%, the latency when using an SSD for the row-cache remains roughly the same. This is because the latest request distribution gives us a high probability that the reads for the rows can be served directly from Cassandra's Memtable, which effectively acts as a write-back cache.

B. Dynamic Schema

Next, we illustrate that by extracting the metadata (i.e., schema) from the data on-disk we suffer no perceivable performance penalty. The column names in our test were fixed at 5 bytes and the number of columns varied between 5 and 10. This accounts to a minimum saving of 25 bytes from being written on a per-row basis. Cassandra, not uncommon from many commercial databases, performs buffered I/O; all reads and writes are executed in 16 KB pages. In our experiment configuration, one row fits well within a single Cassandra page. This means that reading a row will incur no additional overhead since the total size of a row with a co-located schema is larger than a modified row with the schema extracted out. When we extract out the metadata, we expected no degradation in performance or latency and the results in Figure 5(c) and Figure 5(d) confirm our assertion. Specifically, we conclude that in the 95% and 50%-read workloads, the latency and throughput were comparable with any difference being attributed to the environment.

Throughput and latency are not major motivations for implementing the dynamic schema. Fairly significant space savings can be obtained by extracting redundant schema information



(a) Row Cache (Throughput) (b) Row Cache (Latency) (c) Dynamic Schema (Throughput) (d) Dynamic Schema (Latency)

Fig. 5. Throughput/Latency Results for Row Cache Extension and Dynamic Schema

and we find this to be much more compelling. In normal operation, data sizes averaged 6.8GB compressed after the initial load of 40 million keys. With a modified Cassandra, data sizes averaged at 6.01GB of data, a savings of roughly 10%. This value will grow as the number of columns in the table grow and as column names grow in length.

Another potential benefit for dynamic schema model (omitted in the interest of space), is in executing column-slice queries. When performing a read from Cassandra, it is possible to read a slice of the row by specifying which columns to read. Though Cassandra has an index per-row, it is only a sample; not every column has an appropriate index entry. If we have a schema on hand, we know precisely the layout of the row on disk which we can use to optimize the read process and avoid cache pollution.

Finally, it is important to note that we are not using high-end enterprise PCIe-bus SSDs (e.g., FusionIO), yet we are getting a substantial performance improvement. Therefore, we conclude that even with inexpensive commodity SSDs, a considerable throughput and latency improvement is achieved.

VII. Related Work

There exists a recent move in the database community to exploit key SSD characteristics such fast random reads that is orders of magnitude faster than magnetic physical drives and using SSDs to make updates disk-I/O friendly, e.g., [3], [4], [5]. One way to exploit SSDs is to introduce a storage hierarchy in which SSDs are placed as a cache between main memory and disks. This extends the database bufferpool to span over both main memory and SSDs. A novel temperature-based bufferpool replacement policy was introduced in [4], which substantially improved both transactional and analytical query processing in IBM DB2. In our work, we go beyond a simply extension of the bufferpool with SSDs, instead we develop specialized bufferpool enhancements that targets the slow read path problem (incurring many random I/Os in order to consolidate across many SSTables) of key-value stores in the context of Cassandra. Furthermore, we introduce the concept of dynamic schema (i.e., dynamic catalogue) that decouples the commonly joint meta-data and data on key-value stores (such as Cassandra [6] and BigTable [7]) by maintaining the schema information on SSDs. Lastly, in [14], similar to our framework, the use of SSDs as cache was also explored in a proof-of-concept key-value store prototype. In contrast, we introduce the storage hierarchy and our SSD caching techniques within a commercialized key-value store. Furthermore, we identify new avenues for exploiting the use of SSDs within key-value stores, namely, our dynamic cataloguing technique.

VIII. Conclusion

In this paper, we investigated the performance benefits of SSDs in key-value stores. We benchmarked different configurations of SSD and HDD combinations. We proposed and implemented two specific optimizations for SSD-HDD hybrid systems and showed their effectiveness in detailed benchmarks. Our extended row cache strategy transparently stores hot data on SSD and thus extends the row cache in Cassandra. Our benchmarking results show that this extension can achieve improvements of 85% for realistic workloads. Our second technique for SSD-HDD hybrid systems is a dynamic schema catalogue. It reduces the disk impact of row-level schema models and thus increases the performance of common workloads and data sets.

For future work, we will adapt our methodology so it can be directly run on SSD instead of going through the FTL. This will increase the performance of the SSD operations and allow for SSD optimized data structures and algorithms.

References

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The Next Frontier for Innovation, Competition, and Productivity," McKinsey Global Institute, Tech. Rep., 2011.
[2] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii, "Solving Big Data Challenges for Enterprise Application Performance Management," *PVLDB*, 2012.
[3] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "An object placement advisor for DB2 using solid state storage," *PVLDB*, 2009.
[4] ——, "SSD bufferpool extensions for database systems," *PVLDB*, 2010.
[5] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee, "Making updates disk-I/O friendly using SSDs," *PVLDB'13*.
[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Review*, 2010.
[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006.
[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *SOSP*, 2007.
[9] R. Cartell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, 2010.
[10] M. Cornwell, "Anatomy of a solid-state drive," *Communications of the ACM*, 2012.
[11] L. Bouganim, B. r Jnsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," in *CIDR '09: Fourth Biennial Conference on Innovative Data Systems Research*.
[12] G. Graefe, "The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules ," *Communications of the ACM*, 2009.
[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.
[14] B. Debnath, S. Sengupta, and J. Li, "FlashStore: high throughput persistent key-value store," *PVLDB*, 2010.