Αποδοτική Αποθηκευτική Διαχείριση Δομημένων Δεδομένων με Πολλαπλές Εκδόσεις Τιμών

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής Εξεταστική Επιτροπή

από τον

Χρήστο Θεοδωράκη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Ιούνιος 2015

Dedication

To those I consider my family...

Acknowledgements

At this point, I would like to thank all these people, who helped for the successful completion of this thesis.

I am mostly grateful to Prof. Stergios Anastasiadis for his systematic supervision and guidance throughout this research, from the early stages of the design, to the last ones of the composition of this thesis. During all these months, through his core knowledge on the field of computer systems, he gave me the opportunity to indulge in this particular area of knowledge.

The deepest gratitude to my parents for everything they have done, from the early stages of my education till this point, and especially for the moral and financial support they provided me, their patience and the tolerance they have shown during all these years. I am really grateful for their continuous encouragement.

My last years where truly amazing so I would like to thank the following people for their contribution to it: Fwteini Karoutsou for all her valuable tolerance during the months that passed; Angelos Chasiotis and Vasilis Filos for everything we have done together since primary school.

Also I would like to thank Alexandros Papadogiannakis and Thanasis Koufoulis for the endless conversations; Nikolaos Papanikos and Andromachi Hatzieleftheriou for the moral and practical support.

Finally, I would like to thank all the people of the Systems Research Group *(SRG)* at the University of Ioannina, who turned the endless hours of study into a joyful experience. Especially Giorgos Margaritis for all the scientific feedback he provided to me; Vasilis Papadopoylos, Giorgos Kappes and Eirini Micheli for all the interesting scientific and not-so-scientific talks we had within these four walls.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Christos V. Theodorakis, MSc, Computer Science Department, University of Ioannina, Greece. July, 2015. Efficient Storage Managment of Multi-version Structured Data

Thesis Supervisor: Stergios V. Anastasiadis.

Large scale distributed storage systems are the basic components of today's cloud computing infrastructures. They play a crucial role in the overall system functionality, as they often become system bottleneck. Hence they affect the overall end user experience. In order to maintain a high throughput ratio, most of these systems are designed to only provide access to the most recent version of each record.

In this thesis we examine the possibility of combining partial persistence with a key-value store based on a write-optimized data structure. We identify the inefficiencies that a multilevel key-value store presents when users should store more than one values for the same key. Therefore we design and implement the Multiversion RangeMerge in a prototype subsystem that we created based on a widely open-source, key-value store. Through extensive experimentation, we study the performance for retrieving a range of older values and we show that our implementation achieves lower latency for several workloads.

Εκτεταμενη Περιληψη στα Ελληνικα

Χρήστος Θεοδωράκης του Βασιλείου και της Αννας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος, 2015. Αποδοτική Αποθηκευτική Διαχείριση Δομημένων Δεδομένων με Πολλαπλές Εκδόσεις Τιμών.

Επιβλέποντας: Στέργιος Αναστασιάδης.

Η ραγδαία ανάπτυξη των μέσων και των συσκευών που μπορούν να παράγουν δεδομένα εχει οδηγήσει στην ανάκγη δημιουργίας μεγάλων κατανεμημέων συστηματων αποθηκευσης. Αυτά τα συστήματα αποτελούν το βασικό κομμάτι της υποδομής που παρέχει το υπολογιστικό νέφος. Τα συστήμτα αυτά θα πρέπει να έιναι σε θέση να παρέχουν την καλύτερη δυνατή εμπειρία στον χρήστη, ακομή και όταν πολλοί χρήστες προσπαθούν ταυτόχρονα να αποθηκεύσουν ή να ανακτήσουν τα δεδομένα τους.

Η αποδοτική αποθήκευση και ανάκτηση δεδομένων είναι ένα κρίσιμο χαρακτηριστικό για τα δομημένα συστήματα αποθήκευσης. Σε αρκετές περιπτώσεις υπάρχει η ανάγκη διατήρησης και ανάκτησης πολλαπλών εκδόσεων τιμών για τα ίδια κλειδιά. Μελετούμε το πρόβλημα της αποθήκευσης και ανάκτησης δομημένων δεδομένων τα οποία ενημερώνονται με το πέρασμα του χρόνου σε συστήματα αποθήκευσης κλειδιού-τιμής. Προκειμένου να ελαχιστοποιήσουμε το χρόνο ανάκτησης παλαιότερων τιμών, σχεδιάσαμε και υλοποιήσαμε μια καινούρια δομή και μέθοδο αποθήκευσης για τη διαχείριση πολλαπλών τιμών του ίδιου κλειδιού.

Η μέθοδος που προτείνουμε βασίζεται στον αλγόριθμο RangeMerge και την ενσωματώσαμε στο σύστημα LevelDB της Google. Με πειραματική μελέτη της πρωτότυπης υλοποίησης δείχνουμε ότι η μέθοδος μας πετυχαίνει μείωση του χρόνου ανάκτησης πολλαπλών εκδόσεων για το ίδιο κλειδί, και ταυτόχρονα διατηρεί τη ρυθμαπόδοση του συστήματος μας πολύ κοντά στο μέγιστο δυνατό επίπεδο. Συνολικά το σύστημά μας μπορεί να προσφέρει γρήγορη και σταθερή απόδοση ανάκτησης για πολλαπλές τιμές που αναφέρονται στο ίδιο κλειδί, ανεξάρτητα από την κατανομή που περιγράφει την εμφάνιση των ζευγών κλειδιού-τιμής.

Chapter 1

Introduction

1.1 Thesis Scope

In the era of BigData, we observe a rapid increase in the usage of cloud computing infrastructures. Applications such as e-commerce, advertising analytics and social networking are used by millions of users who concurrently store or retrieve data. Interaction with those users drives the underling storage infrastructure to archive, process and serve enormous amount of data. This increase has led scalable data management systems to be a critical component of overall cloud infrastructure.

Traditional relational databases used to be the ubiquitous backend storage system. They provide a wide variety of retrieval API calls and the use of B-trees makes them ideal for point insertion and retrieval. Also the choice of B-tree data structure makes relational databases capable of efficiently storing data which have a predefined scheme. On the other hand their performance degrades when large amounts of data should be retrieved. The time needed for a disk seek limits the performance of Relational Database Management Systems (RDBMS). Another drawback arises from the fact that they can not scale linearly.

In contrast to traditional Relational Database Management Systems (RDBMS), NoSQL scalable datastores provide flat data organization, horizontal scalability, simple design and interface. Due to their benefits in scalability, simplicity and performance many of the

largest and commonly used distributed storage systems employ general purpose Key-Value stores as their storage backend. For instance Google uses a variation of LevelDB [20] as the storage backbone for Bigtable [6] and every other system based on it. That makes KV stores one of the most critical component of the system. An efficient key-value datastore should concurrently insert data to the system while at the same time answer point or range queries. Thus, performance for workloads which contain point or batch write requests and point or range queries, has a major impact on the quality of service that the end user will perceive.

In cases such as stock trading, advertising analytics and bank transactions multiple values should be stored for every key. A downside of existing key-value stores is that they do not efficiently handle older values. In most cases they fail to preserve any multiversion semantics and force the developer or the user to provide appropriate key-value pairs. On the other hand, if it is part of a larger system another mechanism should be used to efficiently handle updates for preexisting keys.

In the present thesis, we investigate the performance characteristics of persistent storage in the context of key-value stores and the overhead for retrieving archived values for a key. In order to provide partial persistence and efficiently store the older values, we introduce the *Multiversion RangeMerge* algorithm, which constitutes an enhancement of the original RangeMerge [21] algorithm. In particular the primary idea is to split the data into two layers. The first layer contains only the most recent value for every key inserted to the system. The second layer sorts and stores older values for keys that have been updated. Therefore, we can exploit hard disk's performance for sequential access and efficiently retrieve the requested key ranges. Thus we manage to maintain almost constant latency for queries which retrieve older key-value pairs. At the same time we move the burden of efficiently storing data from the user to the system, because we natively support multiversioning for the storage system. To evaluate our idea we created a system prototype based on RangeDB [23], which implements the original RangeMerge algorithm.

1.2 Thesis Outline

The remainder of this thesis is organizes as follows:

In Chapter 2, we provide an overview of the related research. We review previous research related to methods that have been proposed in order to convert an ephemeral data structure to a persistent one. Also we present proposed persistent data structures based on B-trees and we overview their performance characteristics. Furthermore we

examine the performance characteristics of various key-value stores which are based on the LSM-Trees. Finally we inspect key components of some of the largest distributed storage systems which use key-values as their persistent storage back-end.

In Chapter 3, we describe the structural components of a key-value store. Specifically, we examine internal mechanisms of LevelDB, which provide a persistent storage based on the structure of an LSM-Tree.

In Chapter 4, we define the general design of our system together with the objectives and the decisions which lead to it.

In Chapter 5, we introduce the Multiversion RangeMerge method that we designed and implemented in our prototype system. Our idea is based on the separation of the historical values for each key from the most recent value for that key. We intend to maximize locality between key and time dimension which leads us to minimize latency of retrieval operations.

In Chapter 6, we describe the experimentation environment that we used in our study and present the measurements for various settings. Experimental results are displayed graphically and our conclusions are justified.

In Chapter 7, the conclusions and the future directions of this thesis are outlined.

Chapter 2

Related Research

In this chapter, we describe some data structures that have been proposed in the literature in order to store multiple versions of the same data while keeping time and storage cost bounded. Furthermore, we review the most advanced data storage systems which utilize the Log-Structured Merge Trees (LSM-Trees) as their on-disk storage back-end.

2.1 Multi version data structures

Every time a new insertion is performed the corresponding data structure should either create a new instance or overwrite the one that already existed. We can assume that every time a new insertion is performed data structures change their shape. In most cases, an update overwrites the existing data and makes it irretrievable. Data structures can be divided in two major categories based on the way they handle updates for preexisting data. They can be characterized as either *ephemeral* or *persistent*. The ephemeral data structures store only the most recent version of the data, while they delete the older

versions. Respectively, the *persistent* data structures keep every change in shape and allow access at any version of the stored data. This category is further divided in two classes based on the data that can be modified. The first class consists of the *partially persistent* data structures which allow access to all previous versions of the data and only the latest one to be modified. The second class is composed of the *fully persistent* data structures which store every data version and provide full access to preexisted data contained at any version of the structure.

The most obvious way for storing multiple versions for the inserted data is to keep a copy of the entire structure every time data is renewed. However, this approach has the drawback of requiring space and time proportional to the size of the entire data structure. To avoid that, Driscoll et al. [11] proposed two techniques to convert any ephemeral data structure to persistent. The first and simplest of these techniques, called *fat node*, uses a binary tree in order to store every update efficiently. Every time an update is inserted, the corresponding node is found and an update record is stored alongside with the previous ones. To do so each node keeps an arbitrary number of labels while maintaining the same pointer fields with those in the ephemeral data structure. Also for every update each node keeps a unique record, called versionstamp, indicating when that label was added. That method also needs an auxiliary data structure for storing access pointers to various versions. That structure can be implemented as an array where after the i-th update, pointers to each label with the appropriate versionstamp are placed in the i-th position. That allows systems to access values for a specific version in $O(1)$ time.

The fat node technique consumes $O(1)$ space for every update, but the time for accessing or updating a node needs $O(\log m)$ in time, with m indicating the number of versions. To overcome that drawback the authors proposed the path-copying technique. According to this technique, each node has a fixed number of fields. If updates to these fields lead to an overflow, a new copy for that node which contains only the most recent value for every label is created. Also, the predecessor of that node should be informed for that new copy. In particular when a new node is inserted or updated, a copy of the path that leads to the corresponding leaf node is created. If the inserted data has to update a preexisting label, the data structure just changes the record in that label. In case of inserting new data, the structure creates a new node and appends that node to the end of a newly created path. This means that a new sub-tree is always created which contains all nodes from the root to the one that should be updated. As a result, the path copying access time costs $O(1)$, after searching for the appropriate root node. If there have been m modifications, the search costs $O(\log m)$ time. After finding the appropriate root, the time needed to access each node is $O(1)$. The downside of that technique is that the space and time

needed for modifications is bounded by the number of nodes that must be copied. In the worst case, a modification may need to copy the whole tree, which leads to space and time consumption $O(n)$.

These techniques can convert any ephemeral structure, that can be represented by a linked list, to a partially persistent one. Minor modifications to the proposed ways can also provide full persistence to ephemeral data structures. Also despite its drawbacks the path-copying technique is used on the Copy-on-Write (CoW) B-tree which is key component of modern file systems [7, 4, 28]

2.2 B-trees and Multi-versioning

B-trees are a generalization of binary trees where each node can hold more than two children. They were developed in order to improve the efficiency during reading and writing large blocks of data.

2.2.1 Partially Persistent

Lomet and Salzbert [19] claimed that transaction time databases which are supported by TSB-trees, a variation of original B-trees, can also provide backup functionality in order to protect the database from hardware failures. They support their claim based on the observation that history versions of each transaction resembles these needed for backup. The cost of keeping periodical backups using a TSB-tree is comparable to that of keeping a conventional differential backup. In particular the proposed method splits the database in two components; the first, called *archive*, stores historical data, while the second holds only the most recent values. Access to both components is provided by a single index. The database is partially persistent as archived values can be accessed but not modified. The cost of search and insertion is $O(\log n)$ in time while the storage cost is $O(n)$ in size. It also requires n/B number of I/O operations in case of a range query, in which n is the size of accessed versions and B stands for the size of the block. Moreover it provides locality to time-slices and an integrated index for both databases.

Becker et al. proposed the Multiversion B-tree [3], a variation of original B-tree, which is asymptotically optimal compared to the ephemeral one (space and time are the same at worst case). The proposed Multiversion B-tree is partially persistent because it allows modifications only for the most recent version while at the same time it supports point and range queries for any version. To facilitate version handling, the proposed method first adds an insertion and a deletion time indicator in order to keep track of the lifespan

of each record. Blocks are characterized as live or dead. Live are the blocks that have not been copied yet, while dead are the blocks that their content has been copied during an update or deletion operation. The Multiversion B-tree changes its shape either when a node block overflows or when a deletion leads a block to underflow. In the case of an overflow, the structural modification creates a new copy of that block and removes all updates from it. This modification, called *version split*, is similar to the node copying operation described earlier. On the other hand, an underflow happens only if a deletion is performed for a record in a node that contains the minimum number of entries allowed. That occurs only to a node storing the most recent version for a record because nodes that store older values can not be modified. In MVBT an update costs $O(\log_B^2 n)$ I/O operations,while $O(\log_B n + r/B)$ I/O operations are required to answer a range query. Note that, B is the size of the block, r is the number of returned records and n indicates the number of updates.

Following the same principles the *Multi Version Access Structure* [31] proposed by Varman and Verma achieves the optimal query time for key range and key history search while optimally processing time range queries. In contrast to previous methods MVAS, is based on a multiversion B+-tree. Values are stored in leaf nodes of the B+-tree which are called *data nodes*. Interior nodes of the tree, called *index nodes*, store only pointers which form paths from the root node to each leaf. Both node types consist of records which contain a key label and a set of two variables indicating start and finish version of that record. Each node is either live, if it contains any live record, or dead if there are no live records in it. An insertion or an update of a new record can lead to an overflow. In that case the overflow is handled based on the number of live records that exist in that node.

Three cases are distinguished based on the two parameters which mark the needed occupancy from live records contained in the block that overflew. In the first case, where the live records exceed an upper bound, the overflown node is split equally between two new nodes. In the second case, live records, which are between a lower and the upper bound, are moved from the current block to a new one. In the third case, where live records are beneath the lower bound, the structure finds a sibling node and copies all live records from the overflown node and either all or part of the sibling node to a new one. The deletion operation for a record just populates the end variable using the current version number. If the end field of a record contains a value, this record is considered dead. If a deletion operation reduces the number of the live records below a certain threshold then a mechanism similar to the one used for insertion operations is triggered. This mechanism finds the appropriate sibling node and merges the two nodes. Figure 2.1

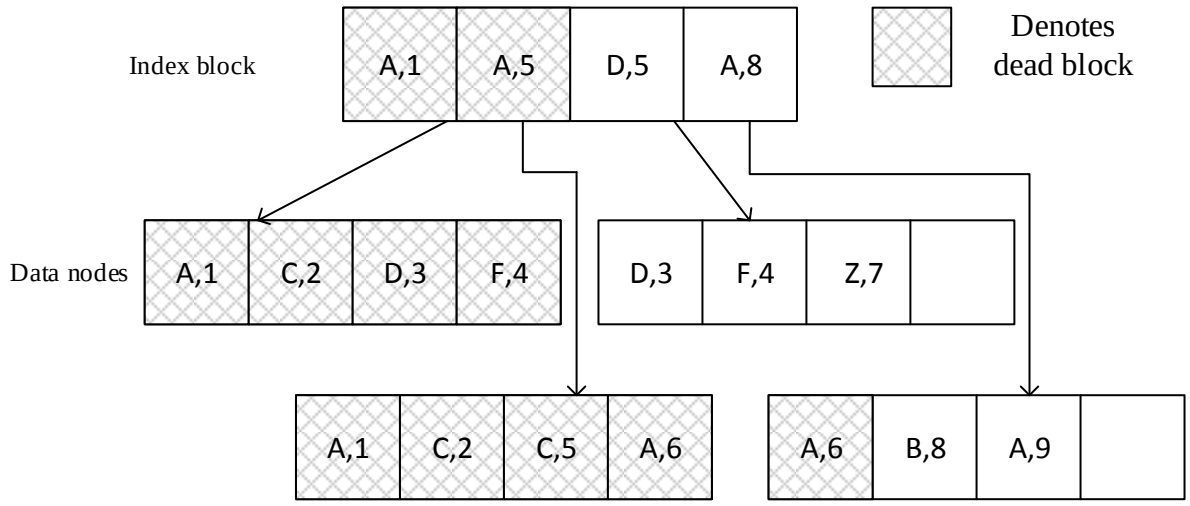


Figure 2.1: Multiversion Access Structure. Each record stores a value and an insertion timestamp. The records of the index node store a pointer to a data block; the value in the index record indicates the smallest key that can be found in a data node and the timestamp shows the creation time of that node. Data nodes contain the actual records, each followed by a timestamp indicating its insertion time. We can observe that records in leaf nodes are not lexicographically ordered but they are sorted based on their arrival order.

depicts the MVAS after the insertion of multiple entries that trigger the creation of new nodes while updating certain values.

MVAS allow users to search either a key, or a range of keys, that existed at time t. At the same time, it efficiently facilitates queries which retrieve all occurrences for a key in a time range. To do so, the proposed method stores each unique record in a C-list which contains pointers for that key in each block. MVAS achieves asymptotically the optimal number of input/output operations for retrieving all keys that exist in a specific time range. Compared to MVBT, MVAS achieves the same asymptotical cost of input/output operations records and range queries as MVBT. However, MVAS provides better performance for the deletion operations as well as a wider variety of range request, which may include more than one dimension.

2.2.2 Fully Persistent

Apart from the partial persistence, B-trees also support full persistence. This is achieved by modifying some key components of the original structure. Stolting et al. [5] present a way to convert an ephemeral B-tree to a fully persistent one. The proposed structure sup-

ports range queries at any version costing $O(\log_B n + t/B)$ input/output operations.The modification of any version of the fully persistent tree is performed in $O(\log_B n + \log_2 B)$ amortized input/output operations. The proposed approach is based on the node splitting method enhanced with new features such as a global version list and a linked list. The latter records all changes for an ephemeral node and creates its *family*.

Another approach to make a B-tree fully persistent was proposed by Twigg et al. [30] Stratified B-trees present a versioned dictionary which modifies nodes in any version, two order of magnitudes faster than the Copy-on-Write B-tree. Also, it reduces the latency for range queries by an order of magnitude compared to Copy-on-Write B-tree. To achieve the aforementioned gain, the proposed approach utilizes a collection of arrays to store sorted tuples (key, version, value) arranged into levels. Each of those arrays is tagged with a subtree of the version tree. Arrays of the same level have disjoint version sets. The supported operations in Stratified B-tree are (i) a key update in some version, (ii) multiple key return in a range for a certain version and (iii) feedback information related to a version.

Fully persistent data structures provide modification to any value at any moment. That further complicates the construction of mechanisms which provide the consistency semantics, especially in cases system serves concurrent transactions. For that reason, in this thesis, we mainly focus on partially persistent data structures.

2.3 Key-Value stores

The Log-Structured Merge Tree (LSM-tree) [26] is a data structure created to minimize the cost of insertion and deletion of randomly distributed data. A configuration of an LSM-tree can be split in two components. The first component is a indexed structure kept in main memory and used to accumulate new entries. The second component consists of multiple modified B-trees that reside on disk. These B-Trees can be organized into multiple levels. In fact, organizing on-disk B-Trees in multiple levels, and having the size of each level greater than the previous one by a constant factor r, minimizes the memory and disk cost. The LSM-tree achieves better insertion performance than the B-tree due to the fact that it batches incoming requests in the main memory of the system and then exploits sequential throughput of the persistent storage medium. Several well known systems use variations of the original LSM-Tree as their storage backend [6, 33, 12].

Dynamo [10], Voldemort [32] and Cassandra [17] are some of the most popular key-value storage systems providing the backend storage to some of the biggest companies in

IT. Indeed, user experience is greatly affected by the performance of the storage backend. In their study Lum et al. designed and implemented SILT [18]. SILT's main goal is to efficiently handle memory in order to serve as many read operations as possible without affecting the insertion time. SILT is designed for flash storage disks and needs only a small portion of memory for each key-value; at the same time it needs to perform almost one read to the storage media. SILT can be divided in three smaller key-value stores, with each one playing a different role in data insertion and retrieval. The first component, the LogStore, is responsible for inserting key-value pairs in flash memory. LogStore also keeps an in-memory index which indicates each key location in the flash log. In order to keep the memory usage as low as possible, the in-memory index stores the output of cuckoo hashing for each key. Each key is inserted at the corresponding position according to a hash function. If a slot is occupied by a previous entry, it is discarded and the new one is placed at its place. Then, the previous entry tries to find the next appropriate position in the index map. If it fails, the system stops the insertion of new keys and converts the LogStore to an immutable HashStore. While inserted to the HashStore, the key-value pairs are re-ordered based on the hash output of each key. At this point the system maintains another map which contains only the hash output of each key for every HashStore, alongside with a filter which provides the same functionality to a Bloom filter. The third component is the SortedStore which stores entries on the storage media, sorted by the key attribute. Except from sorting HashStore key-value pairs, the SortedStore also merges them with the corresponding existing files. To facilitate the efficient look up queries an entropy-coded trie is created. Leaves of that trie are pointing to the location of a particular key on disk. Hence, when the system looks up for a key in the SortedStore, it incrementally reads the trie's representation. The system provides to the lookup function the actual key and its trie representation. Then the system follows a specific path reaching to the leaf which indicates the appropriate offset of data in disk, from where the corresponding key-value pair should be retrieved.

In order to improve the performance of LSM-tree Sears and Ramakrishnan proposed bLSM [29] key-value store. The authors argue that the storage systems should not only achieve high write throughput but they should also provide stable low latency for read operations. To achieve these goals bLSM tries to exploit the read performance of B-trees and, at the same time, to efficiently serve the write requests using an LSM-tree approach. The system is divided in three levels, where each level has an append only B-treelike format. The first of the three levels is kept in memory, while the others are stored on disk. In order to improve the read performance they store a Bloom filter for each tree stored on disk. To limit read amplification for frequently updated data, the system stores

the entire key of every key-value pair instead of keeping just a delta from a previous one. A read operation starts searching from the first tree, which is also the smallest, and stops when it finds the first suitable record. Bloom filters allow system to avoid unnecessary disk seeks for keys that are not stored in a level. On the other hand, Bloom filters do not help in cases of range requests. So the choice of maintaining only tree levels targets to to minimize the impact of the scan operation. Another novelty that bLSM inserted is the spring-and-gear scheduler. The scheduler tries to maintain the size of the in-memory data structure in a certain range. If the first in-memory level approximates the lower end of that range system pauses the merging procedure in lower levels. On the other hand when the upper bound of the range is reached, the system forces application to stall insertion requests. Those mechanisms allow system to adapt its form based on the needs of each workload. Nevertheless that system does not take into consideration older values for the same key.

A most recent approach for constructing an efficient key-value stored based on LSM-tree is presented in LSM-trie [33]. Authors follow the same design principles as in LevelDB (explained in Section 3). The proposed method improves the system's performance by minimizing write amplification. As in most recent key-value stores, the design is based on the fact that the system in addition to the appealing characteristics for write operations should support low latency in get requests. In this direction, LSM-trie modifies the internal structure of each level of LevelDB. More specifically, each level is divided in a number of sub-levels. While the total size of each level is exponentially larger than its previous, the total size of each sub-level is growing linearly. Write amplification is reduced using the above design, because when a sort-merge operation is performed the system chooses only files from the sub-level of a current level that do not overlap. The output of that operation contains one or more files which are stored in the appropriate sub-level of the next level. That method requires the output of the compaction operation to avoid overlapping with any preexisting file in the next level. To achieve this, the LSM-trie changes the way the keys are stored in a file. To find the appropriate file for a key an SSTable-trie is constructed. Each node in that trie is like a container for a number of separate SSTables. Then each key is stored in the file indicated by the result of a predefined cryptographic hash function over that key. That renders the insertion process faster, but the system has the same overhead in the case of read operations as the original LevelDB. To overcome this obstacle, the authors changed the entire format of the SSTable file. They introduced a different file format called HTable. In HTable each data block acts as a bucket for incoming entries. Before a key-value pair is written to an HTable they use a hashing function over the key. The output of this function indicates the bucket

in which key-value pair will be inserted. The use of cryptographic hash-functions ensures that key-value pairs will be uniformly distributed across the buckets of each file. Another key difference is that Bloom filters for buckets from different levels are kept in a single disk block. That minimizes the number of disk seeks needed for a key. LSM-trie is basically designed around Solid State Drives. The extended use of hashing techniques uniformly distributes keys across different files or different blocks in the same file. Hence, it does not take into consideration the locality characteristics of the actual keys. That lack of locality does not allow system to efficiently perform range queries with respect to the key.

2.4 Scalable data storage systems

Large-scale datastores play a crucial role on today's data-center infrastructures. Bigtable [6] is the first large-scale storage system that efficiently handles petabytes of structured data. It is used by a variety of applications, which produced heterogeneous types of workloads. In order to efficiently support the different application Bigtable does not support a fully relational data model. Instead of providing strict relational schemes, clients are allowed to store data based on the locality needs they have. Data indexing is performed using arbitrary strings in a form of a matrix-like structure that contains rows and columns. The flexibility of the dynamic schema allows users to determine if data will be retrieved from memory or from disk files. Bigtable is like a multi-dimensional map spread across many commodity machines. Each value in this map is indexed by a string indicating a row, another one indicating the column and an integer representing a timestamp for every entry. Figure 2.2 provides an abstract view of a row in Bigtable. All row keys are in lexicographical order and each key occupies up to 64KB. Ranges of rows are partitioned dynamically to form what is known as a tablet. Tablet is the smallest unit that can be distributed and is used for load balancing.

Column attributes are also grouped together in order to create the *Column Families*. Each attribute should be part of a column family even if that set contains only one attribute. Column families provide an efficient way to provide access control for data stored either in memory or on disk. Each cell of the map in Bigtable can hold multiple instances for the same key, using a unique timestamp identifier. That timestamp is appended by the system, and represent the real time in microsecond in which the cell has been populated. As an alternative, users can also provide their unique identifier. Bigtable provides a mechanism to remove cells that fulfill a certain criterion provided by the user. Among other important pieces that compose Bigtable, such as GFS [14] and Chubby, data is stored using the Google SSTable file format which stores direct key-value pairs in

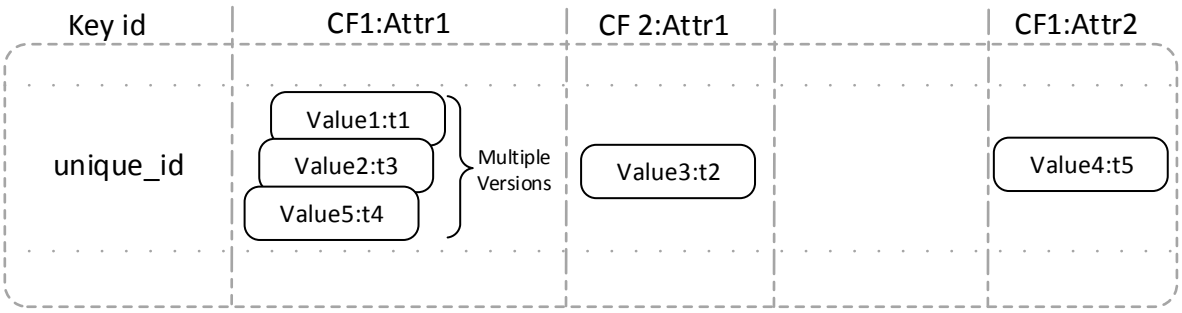


Figure 2.2: Representation of a row in Bigtable [6]. Each row is identified by a unique key, that is associated with values divided in columns, indicated by Attribute. A number of columns can participate in the same Column Family. Bigtable is similar to a sparse matrix and as a result many empty cells exist between two columns. As shown, each cell stores more than one values

raw files. Each SSTable is constructed by blocks of 64KB that contain the actual data and an index which allows efficient lookup operations for a key or a key range. Bigtable compacts ranges of SSTables into a single one in order to keep read latency as low as possible. This keeps the number of files that may be read, bounded by the same number of files which participated in a compaction.

Despite its relatively good performance Bigtable does not support cross-row transactions. To overcome that drawback Peng and Dabek proposed Percolator [27] which incrementally processes updates in large datasets. An idealistic system should be able to efficiently process a large amount of small concurrent updates while tracking which updates have been processed. Many threads may need to concurrently change the content of files that are stored in a repository. To efficiently support that, Percolator provides ACID [15] and snapshot semantics [34] over transactions. That makes it easier for developers to reason about the state of stored data. Every machine participating in Percolator system contains a Percolator worker, a Bigtable tablet server and a GFS chunkserver. Each application communicates with Percolator worker which is responsible for finding any changes that may have happened in tablets stored in Bigtable. Transactions are performed by sending RPCs to the Bigtable table servers. Then each tablet server should communicate with the GFS server. Furthermore two more services are needed. The first is a timestamp oracle which provides monotonically increasing timestamps at any transactions. The second is a lightweight lock service. Percolator uses the same API calls as Bigtable while adding support of multi-row transactions. Multiple versions of data are also stored in Percolator but they are used primevally for providing snapshot isolation. A unique timestamp is appended to each transaction, thus system guarantees that all

committed transactions before a start timestamp will be returned.

The Bigtable's lack of an extended API and the loose consistency model harden the application development. On the other hand, traditional relational database management systems fall sort when the need of serving millions of users appears. To overcome these obstacles Baker et al. proposed Megastore [2]. The basic novelty of Megastore is the combination of the scalability characteristics of a NoSQL system with the functionalities of traditional RDBMS. It provides fully serializable ACID semantics even between replicas that are distributed across a wide area. At the same time it tries to keep latency in levels that real-time applications can benefit from. More specifically, hosts across datacenters are partitioned in entity groups. Each group provide ACID semantics for each operation referring to a node contained in that group. However, the ACID semantics are not guaranteed across different entity groups. The data of every entity group and the replication metadata is stored in scalable NoSQL datastores.

In contrast to Bigtable, Megastore requires a declaration schema for the inserted data. Each schema contains a set of tables consisting of multiple entities. Each entity has a predefined set of properties called values. A set of properties define the primary key of the entity. Furthermore, all entities have a primary key that is unique for each table. As mentioned earlier, Megastore uses Bigtable as its storage back-end. To efficiently store data, keys are chosen in order to collocate entities that will be read together. Each entity is placed in a single row of the Bigtable map and all primary keys are combined in order to produce a unique Bigtable row key. All the other properties of an entity are placed in separate columns. Users are able to retrieve any property from entities by using secondary indexes. Megastore supports local indexes which are used to find keys stored in a local entity. Moreover users can create global indexes that are useful in cases when a field can be found in more than one entities. Scans on global indexes may result into partially consistent output. Index entries are stored as single rows of Bigtable, with the row key constructed by the concatenation of the primary key with the index property. Finally, the underling timestamping mechanism of Bigtable is used by Megastore in order to provide multiversion concurrency control.

Despite its relatively poor performance, Megastore was used by crucial applications such as Gmail, Calendar and Android Market. The semi-relational tables and synchronous replication features makes Megastore a better choice for application development in contrast to Bigtable. Google developed Spanner with the goal to combine the key features that made Megastore attractive with the performance characteristics of Bigtable. Spanner [9] is a scalable multi-version distributed database with synchronous replication. In Spanner each application can create a number of databases, each of them consisting of a

number of tables. In contrast to Megastore, every database in Spanner is split in one or more hierarchies of tables. These tables are similar to those of relational databases which contain rows, columns and versioned values. Each table has one or more primary keys and offers a mapping from a unique primary key to a number of value fields. In Spanner a new mechanism, called TrueTime, is responsible for timestamping each transaction, by representing time as an interval. TrueTime allows Spanner to efficiently support read-only transactions which can retrieve values from the latest completely committed transaction. Also Spanner provides snapshot reads to allow users retrieve the state of the underling database some time in the past. Both the retrieval methods do not need any locking, thus they can be concurrently served.

2.5 Summary

Cloud computing and the growing need for efficient storage has made storage backbone one of the most important parts of a system's infrastructure. Previous research has shown that relational databases lose their efficiency as the insertion rate grows. On the other hand, key-value stores do not provide the same semantics as RDBMSes.

Furthermore, modern key-value stores are mostly oriented in providing very fast insertion rates while compromising the read performance. They achieve this goal by avoiding to write data immediately on disk using structures similar to LSM-Tree. In such structures the read latency is affected by the compaction (sort-merge) algorithm which is used in order to keep the data structured. Moreover, modern key-value stores minimize the amount of stored data by discarding older values for the same key. However, we agree that multiple versions for the same key fulfills a variety of needs.

In this thesis we reconsider the ability of key-value stores to efficiently handle multiple versions of the same key. Towards this end we modify one of the most widely used key-value stores in order to efficiently store and retrieve updates for the same key. We demonstrate that it is possible to achieve nearly constant latency when retrieving data without dramatically affecting performance during insertion.

Chapter 3

Background of LevelDB

The growing demand for managing and storing large amount of data in cloud environments, has resulted in wide deployment of key-value stores over the traditional database systems. In particular, key-value stores provide efficient storage management over structured data. In this chapter we inspect the key components of LevelDB, a key-value store based on the LSM-tree data structure. We start by taking a look at the general design of the system and the features provided to the end user. Then we describe the basic mechanism behind read and write operations, and also the way those mechanisms are implemented in order to provide consistency and durability.

3.1 Overview

LevelDB is a database library which provides the basic functionality for storing key-value pairs on disk. It is developed by Google and distributed freely under the New BSD License [24]. The design of LevelDB follows the same pattern as Bigtable tablet stack, but it was written from scratch in order to avoid dependencies from other non-free libraries

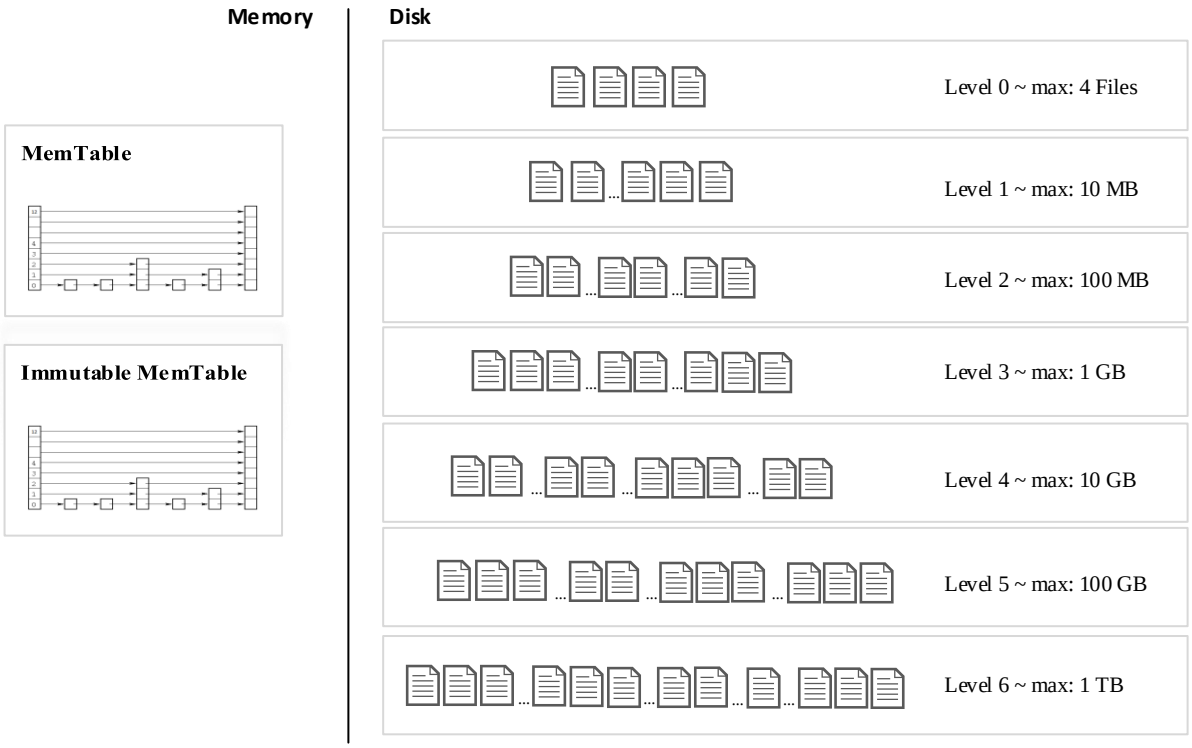


Figure 3.1: General LevelDB overview. After the MemTable is full, it is flushed to Level 0. When the number of files exceeds the predefined threshold all files along with the overlapping files from Level 1 perform a compaction. Compactions may cause overflow at a level $L > 0$ (the total amount of data stored exceeds the max for each level). Then a new compaction is scheduled for one file from level L and all overlapping files in $L + 1$.

which are developed and used by Google. Developers should include the library as part of their application since it neither provides a complete storage server nor supports a command line interface.

3.1.1 Design

The basic system can be split in two major parts shown in Figure 3.1. The first part is associated with the main memory and the logging of most resent insertions. While data is sent from the application to the storage layer, it is inserted to an in-memory buffer called *MemTable*. In order to facilitate efficient insertion and search operations the MemTable is implemented as a skip list. Furthermore, each key-value pair is also appended in a log file.

The second part is composed of multiple levels at which data files exist. In particular, key-value pairs in LevelDB are organized in seven separate levels. Data stored in each level is separated into a number of non-overlapping files, which are similar to an SSTable. Each

level can store a certain amount of data divided in files, each currently holding 2 MB of data. Consistently adding data to a specified level can result in overflow. This eventually triggers a compaction process. During the compaction, LevelDB initially matches each file of the current level with the overlapping files of the net one. Then, those files are merged together in order to produce again a sequence of files, which handle non overlapping key ranges. Newly created files are stored in the latter level and do not overlap with other preexisted files stored there.

3.1.2 Features and API

In order to perform the basic operations of insertion and retrieval of a single key, users can interact with the database through the basic API calls, *put(key, value), get(key) and delete(key)*. Deletion of a key is handled as a special case of write where the same key is placed before the one we want to erase. Additionally, LevelDB provides external iterators which are proportional to range queries. Another interesting feature of LevelDB is the batching of updates, in which a number of put operations can be bundled together and sent as a single write request. In cases of batch writes, the underling system guarantees atomicity, in a way that neither get operations nor iterators are able to see the newly added values until the last one has been written properly.

Moreover the system allows the user to provide a function which is used when keys are getting ordered. Keys and values are stored as sorted pairs of character arrays. By default and if no user function is provided, key-value pairs are sorted via byte-wise comparison between keys. Another functionality that the original system provides is the *Snapshots*, which are a consistent read-only view of the database in the past. Additionally it provides durability guarantees through mechanisms such as logging and block checksums. Finally, a user is able to use the *Snappy* library, in order to minimize the database's on-disk footprint.

3.2 Put pairs

As mentioned in the previous section, there are two ways to modify the database. Users are able to define their own keys which will be associated with each value. After key or keys have been created, users are able to insert pairs either by calling the Put(key, value) API call, or else they can import them as a collection using the write batch abstraction. Deletion of a key, and the value associated with it, are handled as a special case of write. For the rest of this thesis, what holds for insert operations applies also to deletes, unless

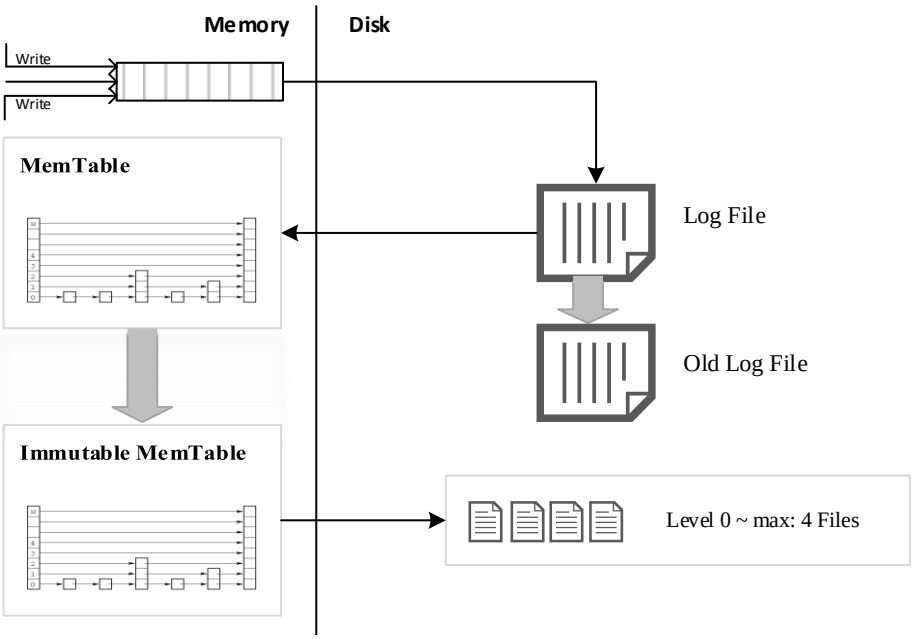


Figure 3.2: Key value pair insertion stages. Write requests are inserted in a queue. In order to avoid congestion, writers queue is protected by a mutual exclusion variable. When a thread is at the head of the queue appends it key-value pair in the log fail and also inserts it in the MemTable

mentioned otherwise.

3.2.1 Write Path

LevelDB uses the MemTable at the first step of data insertion. Key value pairs coming from a write thread are inserted in a lock free skip list. There, the pairs are sorted using either a user-specified function, or a system predefined comparison method. When an update comes for a preexisting key, it is stored just before the first occurrence of that key in skip list. This ensures that the system is able to efficiently fetch the most recent value for a key in the case of get operation. When a MemTable buffer reaches a predefined threshold, it stops accepting new entries. Then, it is converted to an Immutable MemTable and the system allocates memory for a new MemTable The system should only maintain one MemTable and either one or none Immutable MemTable at each time.

Every time a new write should be inserted, the system checks if an Immutable MemTable has been created. If there exists an Immutable MemTable and Level 0 has some available space, Level 0 currently stores up to 4 files, LevelDB flushes the table in that young level. Due to the different insertion rate of the main memory and disk, this procedure can lead the system to a situation in which there is an Immutable MemTable and Level 0 holds

the maximum number of files. In this case the system should stop serving incoming write requests until some space in the first level is free. Then the immutable table is written to that young level and the system is able to continue accepting new write requests.

Reduction, or avoidance, of contention is achieved by slowing down incoming write operations. More specifically, when the number of files in Level 0 surpasses a certain threshold each incoming request is delayed by one millisecond. This is helpful in situations at which the input rate varies. Over time, when the number of write operations keep growing and the MemTable becomes full, the system stops accepting any put operation for one second.

3.2.2 Logging and Recovery

One of the most important features that a database should provide is durability. We can correctly assume that data inserted in LevelDB is durable after it has been written to a file on persistent media. Nevertheless, the modifications of a write operation may remain in memory, either in the Immutable table or in the main MemTable. The system guarantees durability by logging every write request before inserting it to MemTable. When the current in-memory skip list becomes immutable, the logging procedure stops, the current log file is closed, and a new file is created alongside with a new MemTable. In that moment all incoming requests are headed to the new log file. Finally, the Immutable MemTable is written on disk and the associated log file is removed from the system.

After a crash recovery, the system should have the same structure as before. To achieve that, LevelDB holds a file, called the *Manifest*, which keeps information about the files that are stored in each level, the current active log file, the corresponding key range for each file, and other important metadata. When a database is reopened after a crash, the system first reads that manifest file and reconstructs each level. Then it replays the log in order to recreate the key-value pair stored in memory at the time of failure. As a final step, when all keys from the log file are eventually inserted in memory, the system tries to flush the current MemTable to Level 0 even if it has not reached its maximum capacity.

3.2.3 Concurrency

LevelDB architecture is based on the principle of single-writer multiple-readers. Actually more than one threads may request to perform a write operation concurrently in the database. That can lead to an inconsistent log and can also create problems to the in-memory skip list. To avoid those drawbacks, the system serializes incoming write requests by inserting each write thread in a queue. Multiple threads may try to insert themselves

in the queue concurrently, so it is protected by a mutual exclusion variable. In order to insert a key-value pair, a thread should perform the following steps:

1. Lock a mutual exclusion variable
2. Insert itself in writers queue
3. Wait if it is not inserted at the head of the queue
4. Check if MemTable is full
5. Unlock the mutual exclusion variable

Following those steps only the thread at the head of the queue can continue the insertion procedure, after the previous one has finished. Insertion at the log file and the MemTable, is not part of the critical area because at that point only one writer may wake up and continue its writing process. When the insertion is complete, the thread performs the steps described below in order to finalize the procedure of populating the database:

1. Lock a mutual exclusion variable
2. Drop itself out of the head of the queue
3. Check whether there is another thread in the head, in which case it wakes it up
4. Unlock mutual exclusion variable

The previous analysis gives us a more detailed view of the system's behavior from the moment a write request is triggered, until data is written in a persistent file at the first level.

3.3 Compaction

The overall design of LevelDB is based on the LSM-Tree. In order to exploit the advantages of this data structure, the system needs a mechanism to keep data sorted at each level. Also LevelDB should migrate data from one level to the next one when a certain threshold for that level is reached. This mechanism is called *Compaction* and its functionality is to find the file or files that should be sent to the next level, and also detect all the overlapping files at the next one. The compaction process merges all key-value pairs contained in those files and produces a set of files containing the resulting sorted keys.

The compaction procedure starts when the young Level 0 becomes full. At that point the system picks a file from that level and all overlapping files from Level 1 and creates

an iterator over their corresponding entries. In order for the view of the database to be consistent, the iterator creation is protected by the same mutual exclusion variable mentioned in the previous section. The files participating in that compaction can not be removed from the system until the iterator has ended its life cycle. Although the creation of the file iterator is part of a critical area, the actual compaction work is performed while the other threads continue their execution. This allows the concurrent insertion of key-value pairs to the MemTable, as well as read operation. As mentioned earlier, files in Level 0 may contain multiple occurrences of the same key. Those values are dropped during the iteration over the keys stored in Level 0. The output of compaction is a series of non-overlapping files, with each file containing a unique key range. As a final step, the thread responsible for the compaction informs the system about the new files, deletes older files if no other threads read them, and makes those changes persistent by appending them to the Manifest file. This final step is also protected by the mutual exclusion variable. A compaction can result in overflow at the level that new files will be stored at. Then, another compaction is scheduled. The same procedure is followed for the level that overflew. The same routine is applied recursively until either the resulting files fit at a particular level, or the last level has been reached.

Compaction consumes a large amount of disk resources. Indeed, every time a compaction is triggered the system should read one or two files from current level and approximately ten to twelve files from the next one. As a result, concurrent read requests stall and impact the user experience.

3.4 Get Pairs

A user is allowed to retrieve key-value pairs concurrently with storing data. Especially, a user can request either a single pair through the Get API call, or otherwise iterate over a bunch of pairs. In contrast to insert operations, multiple read requests can be performed at the same time without the need for serialization or any other type of synchronization among them.

3.4.1 Concurrency

When the Get API call is used, the system tries to find the corresponding key first in the MemTable. If the key is not found and an Immutable MemTable exists, the system tries to find it there. In case where a key is not present in either of MemTables, the system searches each file at every level. In the previous section, we mentioned that write

operations may trigger a compaction, which could either erase any of the in-memory data structures, or delete files which where part of a finished compaction. That could lead to an indeterminate behavior during execution of get operations. To avoid such circumstances, the system uses reference counters for every structure that can simultaneously participate in a write or get operation. Thus, every time a thread needs to retrieve a key-value pair from the database, the following steps should be followed:

1. Lock of a global mutual exclusion variable
2. Increment of the value of reference counter for MemTable by one
3. If an immutable MemTable exists increment of the reference counter value by one
4. Increment of the reference counter for all files by one
5. Unlock of the global mutual exclusion variable

At that point all structures available to read are referenced so they can not be deleted during information retrieval. This enables the system to perform other tasks concurrently.

Whenever necessary, the system deletes obsolete files as part of the garbage collection mechanism. In order to completely remove a file, or even the in-memory data structures, and free the occupied space, the system checks the corresponding reference counter. If that value equals to zero and these structures have been marked as outdated, no one requested to access them. The key-value pairs stored there have been updated or moved to another file, and at that point it is safe to remove those files from the system. In order for the reference counter to decrease and reach to zero value, the following actions should take place every time a read operation ends:

1. Lock of the global mutual exclusion variable
2. Decrement of the reference counter of MemTable by one
3. If an immutable MemTable exists decrement of its reference counter by one
4. Decrement of the reference counter for all files by one
5. Unlock of the global mutual exclusion variable

The usage of reference counters forms a suitable fine grained solution for the problem of concurrent read and write operations.

3.4.2 Iterators

Iterators follow the same principles with get operations. Actually the get operation is based on low-level internal iterators, which stop their recursion over key-value pairs when they find the first suitable entry. On the other hand iterators allow users to retrieve more than one value, if necessary. To do so, when an iterator constructor is called, the system creates internal iterators for every level that contains data. Then they are merged together in order to give the impression of a single one. Every time a seek for a key is performed, all the iterators internally seek each level in order to find the requested user key or the next one based on the result given by the comparison function.

When a user wants to retrieve a range of keys, one should use the *seek(key)* function provided by the iterator abstraction. This function searches for the first (smallest) key, based on the ways mentioned before and then uses the *next()* function in order to find the next smallest key. When the next() function is called the system moves all iterators to the subsequent entry, which is greater than the current retrieved key. In order to fetch the key which is closer to the current one, internal iterator values are compared and the one with the smallest key is returned to the user.

3.5 Summary

The LevelDB [20] has been created by Google as a lightweight storage back-end. It was written from scratch in order to have no dependencies from any library used internally by Google. Its architecture is based on the storage layer used by Bigtable [6]. The main goal of this system is to achieve fast insertion of random write operations, while keeping the cost of read operations as low as possible.

To achieve low latency in write operations, key-value pairs are first inserted in an in-memory skip-list, called MemTable, in which they are sorted based on the key attribute. A user can specify sorting by providing a function to be used for key comparison. If there is no such function provided, the system performs byte-wise comparison between keys. At this point, more than one value for each key may exist. Data stored in MemTable should be protected against failures that may cause main memory to erase its data. Hence, every key inserted in the system is also appended to a log file. After the MemTable reaches a predefined threshold, it is written as an entity in a file on disk.

The disk is organized in seven different levels. When the MemTable is flushed to disk it is written in the young level (Level 0). The main functionality of that level is to act as a "buffer" which stores a number of files before they create a totally ordered array of

smallest ones. At that young level the content of different files may overlap. Furthermore, when Level 0 fills up, a compaction is triggered. Then, one or more files are chosen along with the overlapping files from the next level. Subsequently, the entries of those files are sorted and multiple versions for a particular key are discarded. Each level L can store up to 10^L megabytes. When the limit for a level is exceeded another compaction is triggered.

Finally, the system allows the user to perform queries in order to retrieve stored data. The end user can request a specific value through the Get API call, similar to point queries. Alternatively a user can read a number of consecutive key-value pairs using an iterator abstraction, similar to range queries. Data consistency is achieved in cases of concurrent read and write operations with the use of mutual exclusion variables and reference counters.

Chapter 4

System Design

In this chapter we initially present the design goals of our study. We also provide the key insights of the architectural components that compose our system. We start by defining some inefficiencies of the current design of LevelDB, which lead to unnecessary overhead during systems normal operation. Then, we propose a more efficient way for storing data with respect to the time that the system needs to answer range queries.

4.1 Design Goals

Most modern datastores are oriented in providing fast and reliable storage for randomly distributed key-value pairs. While they offer relatively low insertion latency they should also provide low latency, when users want to retrieve their data. Except from retrieving only the most recent value for a key-value pair, a number of systems provide the possibility of storing and retrieving older values for a certain key [16, 13, 9].

Large scale systems, such as Bigtable [6],HyperDex [12] and Hbase, have developed internal complex mechanisms in order to keep data sorted based on time dimension. On

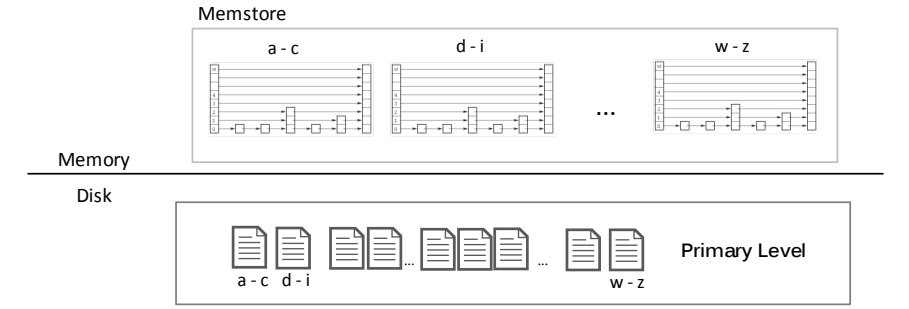
the other hand, smaller scale key-value data-stores, such as the LevelDB, do not use any mechanism that preserves older values for a key. Instead, they transfer the weight of finding an efficient way for key sorting to the user. The most efficient way of data storage should take locality into consideration. Therefore, in order to efficiently retrieve ranges of data, referring to the same dimension from disk, a user should find a way to store them sequentially on disk. Hence the user can get the maximum throughput of the device.

In the case of a multi level LSM-tree, such as LevelDB, we can sort data along one dimension. More specifically, if we should take into consideration the key space and time dimension we should choose to store either continued keys sequentially or keys continually in the order they arrive in our system. That choice could lead to retrieve efficiently either a key and all of its updates in a time range or every key that has been inserted in a time range. Another drawback in multi level LSM-trees has to do with the fact that when a large amount of data has been inserted in the system, and some of them are updates, values for the same key can be located in different levels. So when we perform a query to retrieve values for that key, a disk should do more than one seek (in the worst case one for each file).

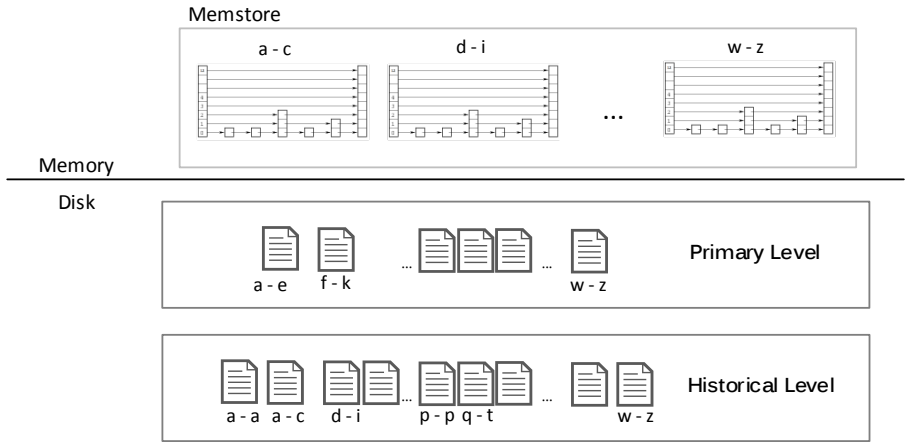
In this study we propose an alternative way for storing data. We focus on efficiently answering queries that request either the most recent value for a range of keys, or the result of every update that has been performed for a single key in a specific time range. The user can take advantage of this functionality through two new API calls. Also, we strive to take the responsibility of key creation and sorting of the user to the system. A system that has its own mechanism to provide versioning to data at the storage layer, it can be used by users who want to access older values. Similarly, large-scale systems can take advantage of our design in order to natively provide features related to multiple versions of a specific key.

4.2 Overview

In our design we want to take advantage of data locality, in order to store data sequentially on disk. Ideally we would like to be able to store data from different dimensions closely on disk. That could lead to an optimal response time for range queries which include different areas of key space. Various solutions for this problem have been proposed (e.g Stratified B-trees [30], HyperDex [12]) especially in cases where data does not expand in many dimensions. In practice those solutions are not widely used because they either do not fully utilize the underlying hardware, or need more recourses.



(a)



(b)

Figure 4.1: In Figure (a) we see the architectural overview of the original RangeMerge. The system keeps in Memstore a number of MemTables, each of them corresponding to a range of keys included in one file at the first layer. In Figure (b) we see the architectural overview of Multiversion RangeMerge where we added the Historical layer. In our approach each MemTable in memory corresponds to a range that exists on the second (Historical) layer.

Therefore we propose a new scheme in which there are two separate layers, shown in Figure 4.1. In the first layer, called *Primary Level* we store only those files which contain the most recent value for a particular key-value pair. In the second layer, called *Historical Level*, all the updates for a specific key are sorted and stored sequentially taking into account the time dimension. Thus, we allow the user to get a set of keys along with their most recent value as fast as possible. Also, we allow the user to retrieve all the values

Algorithm 4.1 Pseudocode indicating KEY-VALUE PAIR INSERTION PROCEDURE

```
1: function INSERT_KV_PAIR(key, value)
2:     Memstore := Container for multiple MemTables
3:     MaxMem := Maximum memory provided by the user
4:     // If Memstore exceeds its maximum capacity
5:     // flush and compact the appropriate MemTable
6:     if (Memstore.size() > Max_Mem) then
7:         MaxTable := Find_Max_MemTable(Memstore)
8:         Flush_And_Compact(MaxTable)
9:     end if
10:    // Find appropriate MemTable in Memostore for the key
11:    ApprTable := Find_Appr_MemTable(key)
12:    Insert_KV_Pair(ApprTable, key, value)
13: end function
```

```
1: function FLASH_AND_COMPACT(MaxTable)
2:     // Find files from Level One that overlap with MaxTable and compact them
3:     // Store dropped values in DropedPairs and compact them with
4:     // overlapping files from history Level
5:     // Create new split in memory based on the files they produced the latter com-
   paction
6:     L1_files := vector storing pointers to all files in Level One
7:     DropedPairs := MemTable to store dropped key value pairs during compactions
8:     over_files := Find_Overlap(MaxTable, L1_files)
9:     Compact(MaxTable, over_files, DropedPairs)
10:    LH_files := vector storing pointers to all files in history Level
11:    over_files := Find_Overlap(DroppedPairs, LH_files)
12:    created_files := Compact(DroppedPairs, over_files)
13:    CreateNewMemSplits(CreatedFiles)
14: end function
```

for a particular key within a time range efficiently by exploiting the maximum available throughput of the device.

4.3 Insertion

The proposed system architecture is based on RangeMerge [21]. The basic idea behind RangeMerge is that instead of having a number of levels that store data, we only have one. Each file in this level is responsible for a different key range. In memory, we keep one MemTable for every key range stored in every file and each new key-value pair is placed in the corresponding table. When that table fills up, it is compacted with the overlapping

files in that first level. As a result, this compaction can produce files with new ranges, which are also mapped in memory. To achieve that the system should split the MemTable from which the compaction began into smaller tables. We benefit from RangeMerge since it performs only small compactions that do not interfere with the retrieval operations.

The scheme described above is very useful when we need to store only the most recent value for every key. When, multiple versions of a value are required, the problem of locality arises. Therefore, we introduce the Multiversion RangeMerge algorithm, which provides maintenance of multiple versions for particular key by adding one more storage layer. That layer stores older values for each key kept sorted. An update request can refer to either a completely new key or a pre-existing one. An abstraction of insertion procedure is described in algorithm 4.1. An incoming key-value pair should be inserted in the appropriate MemTable in main memory. If there is no space left, the memory buffer which occupies the most space is flushed to disk and compacted with the overlapping files. At that point, the original RangeMerge keeps only the most recent record for each key and drops the older ones. Multiversion RangeMerge gathers the dropped records in a temporary MemTable and stores them in the second layer. In order to keep data sorted in the second layer, the Multiversion RangeMerge compacts dropped values with the overlapping files in the second layer. That leads us to change the way that ranges for each MemTable are defined. In our design we split the memory in ranges based on the keys that each file in the second layer holds. Overall, we take the above decision with the goal to minimize the resource consumption during the compaction procedure.

4.4 Retrieval

In the original design of LevelDB, in cases of range queries, the user should provide the first key that should be retrieved and then the system should scan over all following keys until an end criterion is met. For instance, a user may provide the starting key, A, and require from the system to iterate over all values until it finds key F, or until the system has retrieved 10 values. If we assume that keys are stored sequentially based on the time dimension and the user wants to retrieve only the newest value for a range of keys, the system should iterate over keys which are not actually needed. The number of updates for particular key impacts significantly the time spent for iterating over unneeded key-value pairs.

On the other hand our proposed design is capable of retrieving the most recent value of a range of keys by iterating over those that exist in the first layer. The keys at this layer are sorted lexicographically without taking into consideration older values that may

exist. So the system can sequentially read from storage media the keys that satisfy the user's query. Furthermore, the system efficiently uses sequential read disk access when retrieving a number of values for the same key that has been updated within a certain time range. To achieve that, the system finds the file or files which contain values in the time period needed in the second layer that we added. There, the values for each key are sorted based on their insertion time. This approach gives us the advantage of iterating over contiguous records in the file at sequential disk throughput.

4.5 Summary

In the present chapter we highlighted some of the characteristics that make multi-level LSM-trees inadequate to handle more than one versions of a key. In particular we pointed out that storing multiple values for the same key could result in poor locality which further causes higher latency in range queries. We also introduced the Multiversion RangeMerge an improvement over the original RangeMerge in which we added one more layer which is responsible for storing older data. Thus we take advantage of the fact that keys should be sorted along two different dimensions and we introduce a dimension in each level. Subsequently we are able to perform sequential reads which allows the system to minimize latency when answering to queries either in key dimension for a range of keys, or in time dimension for a single key.

Chapter 5

Prototype Implementation

The performance characteristics of LSM-tree make it preferable over all other data structures when it comes to inserting large volumes of data. It exploits the maximum throughput of the storage medium by converting random writes to sequential. However, its design limitations prohibit the storage of multiple versions of the same key, while its compaction algorithm is characterized by a huge impact on the retrieval latency. In this chapter we describe our approach which enables storing of multiple versions for each key and significantly reduces the retrieval latency. Towards this direction, we implemented the *Multiversion* RangeMerge adding a new storage layer in the RangeMerge algorithm [21]. Our prototype is called *Mneme* and it is based on RangeDB [22]. Because both Mneme and RangeDB are based on LevelDB for the rest of this chapter we base our terminology on LevelDB.

5.1 Timestamps

Each key should have a field that keeps a unique number indicating the relative arrival order. In the original system, each record is represented by an array of bytes. In that

array some other auxiliary information except from the actual key and the respective value are also stored. Just after the end of the actual key there are eight bytes that store information about the type of the key and a unique sequence number. Also in the original LevelDB, the system initializes a new counter every time a new database is created. When an insertion request arrives, the system increases this counter by one and appends it just after each key. This indicator is usable only in cases where user takes a snapshot of the database. The system decides not to discard each pair with greater indicator value than the smallest snapshot.

In our system there is no need for snapshots since we store every key that arrives. So we modified the timestamping mechanism by inserting the actual time that each key-value pair is written in the MemTable. Every time a key is inserted in the skip list, we append the time in microseconds since UNIX time. Therefore, we uniquely characterize each key. At the same time that enables us to easily find out which pairs were present at a specific time range in the past.

5.2 Compaction

In the previous chapter we have already mentioned that our approach relies on RangeMerge [22]. According to the RangeMerge algorithm, RangeDB keeps a unique key range every time a new file is created on disk. More specifically RangeDB consists of an in-memory structure called *Memstore*. Memstore contains multiple MemTables. Also it maintains an on disk layer called the Primary Level. Each files in that level contains a disjoint range of keys. Each of this range is associated with one MemTable. Upon the arrival of a new key-value pair RangeDB inserts it in the corresponding MemTable. When the total size of Memstore reaches a predefined threshold, the system picks the MemTable with the largest memory footprint, it flushes it on disk, and compacts it with the overlapping files.

In our prototype, we add a second layer for archived values, the Historical Level, and we modify the compaction procedure. Especially, every time a compaction is performed, we sort and store the older values at the history layer. In particular when the available memory reaches a predefined threshold, the system tries to store a file in Level 0. If a file already exists, it reads the minimum and the maximum keys stored in that file and detects all files from the primary level which contain keys in that range. In Mneme we also find the files in historical level which may be affected by the compaction output. In contrast to the primary level, it is not enough to just check if each file content may overlap with the range between maximum and minimum key. As an example, we can assume that the

files which should be compacted from Level 0 store keys in ranges between A and F, and there can be more than one occurrences of a key. In the historical level we assume that there are ten files storing some values only for key A. After those files there should be one file storing keys between every older value for keys from A to F. And after that there are other ten files storing multiple values of F. If system just reads each file that overlaps with that range, it will end up reading all contents from every one of those twenty one files. In fact the system needs only to read only two of them, the one containing all the updates from A to F, and the last file that stores the last value for key F.

So we change the way that compaction mechanism chooses which files should be fetched in order to retrieve the minimum possible number of them from the historical level. In our system we decided to read either only the file with the same smallest and largest key value but stores the biggest timestamp for that key or the one file that covers the same range as the file from Level 0. As we explain in the next section, facilitating in-memory splits based on ranges of files in the historical level, allows us to read only the minimal amount of files from disk.

5.3 Memory Management

In the original LevelDB design there is only one skip-list kept in memory, which sorts all the incoming key-value pairs based on a user-defined function. When a threshold is reached, the system writes to disk the memory contents. Flushing can cause a number of recursive compactions that can interfere with simultaneous read requests. In contrast, in RangeDB there are as many MemTables as the ranges that comprise primary level. In RangeDB the collection of MemTables are grouped together and stored in Memstore, allowing the system to perform a small immediate compaction between only two files.

In our approach we keep Memstore but we create each MemTable based on ranges stored in the Historical Level. In this case, arises the problem of storing multiple values for the same key in a single file. To make the problem easier to understand we consider the case where we want to store keys in range from A to F. For simplicity we assume that we need to store only one occurrence of A, two occurrences of C and only one occurrence of F. Additionally, each file can store only two keys. Thus, a compaction ends up creating two files, one containing A and C with oldest timestamp, and the other containing the second occurrence of C and F. If we reserve two MemTables with those ranges the next update request for key C will result in breaking the basic assumption of keeping keys sorted in each level. This can happen if an update for C is inserted to the MemTable

Algorithm 5.2 Pseudocode for CREATING RANGE SPLITS

```
1: history_files := File created by compaction for Historical Level
2: range_splits := Array to store new memory splits
3: // Iterate over history_files to find proper ranges
4: for every file i in History_Files do
5:     if file i contains different keys then
6:         if ( max stored key in file i == min stored key in next file) then
7:             // create the range [i.min, i.max)
8:             new_max := First key previous to i.max
9:             insert i.min and new_max in range_splits
10:        else
11:            // create the range [i.min, i.max]
12:            insert i.min and i.max in range_splits
13:        end if
14:    else
15:        // file contains same keys
16:        if ( max stored key in file i != min stored key in next file) then
17:            // that split will contain only keys which are equal to i.max
18:            insert i.min and i.max in range_splits
19:        end if
20:    end if
21: end for
```

which contains the range from A to C. When this table is compacted with the associated file in the historical level, it creates a file storing a newest update for key C than the one in the file containing C and F. In order to avoid this situation, in Mneme we find the appropriate ranges that will be created in memory by using the algorithm shown in 5.2.

At the end of each compaction, we get the ranges of the new files stored in Historical Level. In order to retrieve the appropriate information about files stored in each level we use MetaIndex. MetaIndex is a simple data structure which maintains all metadata information about each file stored in each Level. It was integrated in the original LevelDB and we also use it in Mneme. For each file we check whether it contains a range of keys or the same key. In the first case, if the maximum key for the file is the same as the minimum key of next file, we create a slightly smaller range for this file in memory. Thus the current range will serve every key which is smaller than the maximum key for the corresponding range. At the same time, when the next range is created, its smaller key will be the same as the one we omitted. In the second case, the file stores updates for a single key. This key can also participate in another range. If there is no other file containing this key we create a MemTable which serves incoming updates only for values coming with the specific identifier. If a newer update for that key is found in the next

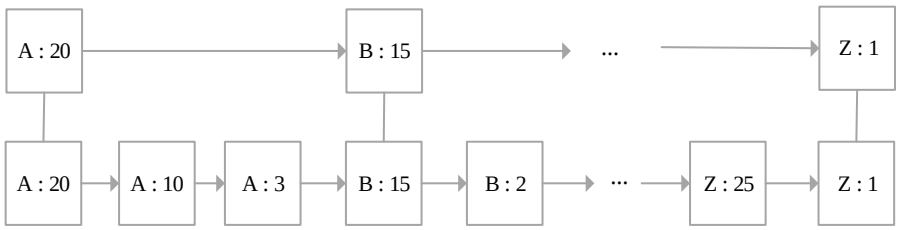


Figure 5.1: Skip list in LevelDB. Every node in skips list indicates a key:timestamp pair. We can observe that at every level of the skip list, the keys are sorted in ascending order based on their key attribute. In contrast updates for each key are sorted in descending order

file, we do nothing because it can be used as the starting point of the next range.

5.3.1 Memory Consumption

Every time a compaction is performed LevelDB flushes skip list in Level 0. As shown in Figure 5.1 while the unique keys are kept in ascending order the updates for each key are sorted from the newest to the oldest. We follow the same principle in our system. This is suitable when we need to store only the newest occurrence of a key. The system is able to find the most recent update for each key easily by just keeping the first occurrence of each different key. When a table is flushed from memory to Level 0, it contains every update that has been applied to each key while in memory. Hence, every time a compaction is performed, the updates for each key are discarded from the most recent to the oldest one. If our system just appended those values at a disk file until the file reached a certain size, we could result in storing files where the minimum key for some of them would be greater than the minimum key of the next ordered file.

Therefore, we reform the keys that are dropped by concatenating the time stamp at the end of each key. Additionally, we insert the reformed keys in a temporary MemTable, which acts as an intermediate buffer. Insertion and sorting of those reformed keys practically does not add any latency in our system, because those operations are performed in memory while a file is read from disk in the background. Disk Input/Output operations are slower by an order of magnitude compared to Input/Output operations performed in main memory. So the latency for sorting those data in memory is amortized over the latency of disk I/O seeks.

Furthermore the temporary MemTable occupies only a very small portion of system's memory. Actually the amount of memory needed is equal to the size of all key-value pairs

that have been updated for the file participating in the compaction. In the worst case scenario the MemTable ends up occupying the same size as an on-disk file. A corner case can occur before any split is created in memory, when the system is possible to end up using up to three times the provided memory, for a short period of time. That corner case could happen only when there is no data in our system and no split has been performed yet. Especially, if all the incoming requests are updates for the same key, the system keeps in main memory (i) a MemTable storing the current incoming requests, (ii) an Immutable MemTable and (iii) almost all values for the ongoing compaction.

During normal operation, the memory consumption of our system is more efficient in comparison to LevelDB. On average, over time our system keeps in memory a Memstore of user-defined size, a small immutable MemTable filled and flushed from Memstore and a small MemTable to store the dropped pairs of ongoing compaction. On the contrary, LevelDB requires the memory for the MemTable and the Immutable Memtable, which is equivalent to two times the size of Memstore.

5.4 Retrieval

As stated earlier our goal is to provide an efficient way of retrieving both every update in a certain time period, and also allow the user to retrieve the most recent values for a particular range of keys. Therefore, we provide two ways for retrieving keys from the underlying database. Both of our methods are based on an Internal Iterator abstraction which is implemented internally in LevelDB. Those retrieval methods are provided to end users through the following API calls that we introduce:

1. GetHistory(key, start , finish, buffer): In order to properly retrieve older updates for a particular key in a range, the user should provide (i) the key to be retrieved, (ii) a starting value which indicates the oldest point in time back to which the key is considered valid, a finish value indicating the newest point in time up to which the key is considered valid, and (iv) a buffer where output is stored.

2. GetRecentKeys(start, finish, buffer): To properly retrieve the newest updates for a range of keys, should only populate start variable with the smallest key and finish with the largest key that is considered valid. Also user should provide an appropriate buffer where retrieved values are stored.

Our introduced API calls takes advantages of the fact that values are stored sequentially on disk. Thus, we can contiguously retrieve multiple values with a single disk seek.

Particularly upon a GetHistory call, the system searches the historical level for the key provided by the user which has a timestamp equal to start value. We can correctly assume that it is difficult for a user to provide the exact point in time that the key was inserted. So if this time value is not found, the system searches for the first key with timestamp attribute following the one provided by the user. Subsequently we retrieve all the values that have a timestamp smaller than the one stored in the finish argument. If the system finds a timestamp for that key that is greater than the finish time, it stops searching and returns the key-value pairs involved to the user. In case that the finish value is greater than the ones stored in the historical level for that key, the system also tries to find the appropriate file in the primary level where the most recent value of that key is stored. If the key has a timestamp within the time range provided by the user, it is added in the buffer and returned to the user.

Moreover the GetRecentKeys() call provides a functionality similar to the iterator. Upon a GetRecentKeys() call, the system tries to find the file containing the requested key in primary Level. Again if the key is not found, the smallest key which is greater than the one requested is retrieved. Then, the system stores all values until it finds a key greater than the one stored in finish. Retrieved values are appended to the provided buffer and returned to user. Finally our system also supports point get queries through the Get API called provided by LevelDB.

5.5 Summary

In the previous sections of this chapter we described the key components of the Mneme system. Our system uses real timestamps in order to determine the order that the key-value pairs are inserted to the system. Also we added the Historical Level which stores dropped key-value pairs sequentially based on the key and the time dimensions. Furthermore we described how Mneme uses the Multiversion RangeMerge algorithm to perform the appropriate compactions and split the Memstore in multi MemTables. Finally we gave an overview of the new API calls that we inserted in order to allow users to retrieve the required key-value pair efficiently.

Chapter 6

Experimental Evaluation

In this section we will present the result of our systems performance. We will start by mentioning the environment in which experiments took place. Afterwards we study the needs of Mneme and we compare its performance with LevelDB when historical data should be stored and retrieved. We graphically present our results.

6.1 Experimental Environment

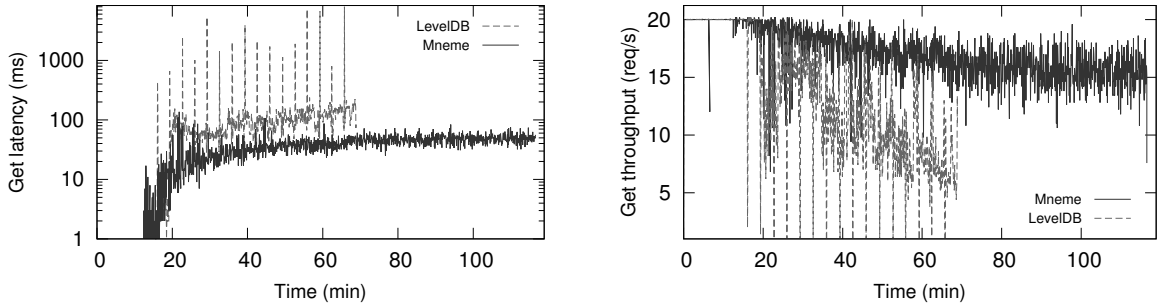
We implemented the Multiversion RangeMerge in Mneme. Our prototype implementation was evaluated using an x86_64 based server running the latest Debian (Jessie) distribution. For our experiments we used nodes equipped with two quad-core 2.33 GHz processors (64-bit x86), one activated gigabit ethernet port, and two 7200RPM SATA2 disks. We configure the server RAM equal to 4GB. Each hard drive has 500GB capacity, 16MB cache buffer size, average seek time 8.5-9.5 ms, and 90 MB sustained data transfer rate. We use the Linux ext4 as the underlying filesystem. In Multiversion RangeMerge we use

rangefiles of size F=32 MB. We also examine the case of LevelDB, which is similar to the storage backend of Bigtable [6] and HBase.

In order to study the characteristics of our system and evaluate our implementation, we performed extensive measurements. Due to lack of public traces [1], we use synthetic datasets with keys that follow the uniform or the Zipfian distribution. To produce the synthetic dataset and to measure our system performance, we use a microbenchmark created to test RangeMerge. We extended that microbenchmark so that it can create updates for key-value pairs that already exist. Specifically in the case of a uniform distribution, we let the system choose either that the new entry will be a new key-value pair, or it will be an update for a preexisting one. In our experiments we modify the number of updates by making it easier for the system to choose between the two states.

Both Mneme and LevelDB, are using a total of M=512 MB for the accumulation of items in memory. In most of our experiments we insert key-value pairs until the total inserted data is equal to 10GB. That size for the datasets fills up 20 times the in-memory buffer and creates interesting compaction activity in both systems. We insert a key whose size varies from 16 to 100 bytes, but we keep the size of the corresponding values stable at 1 KB. Additionally we take measurements every 5 seconds which include average, 95th and 99th percentile of latency for all request in that time period. All requests are range queries. They retrieve either all keys in a time period of 3 seconds. In case of time range queries, retrieved values have different sizes due to the randomness of key creation. On average, both systems retrieve approximately 10 key-value pairs when the distribution of keys is uniform and half of the inserted key-value pairs are unique. In the case of Zipfian distribution systems retrieve on average 78 key-value pairs, but the number of returned values may vary from 1 to 200 entries per request.

Finally, we modify LevelDB in order to keep all the inserted key-value pairs. Before inserting a key in memory, we reform it by appending the timestamp at the end of each key. This is the simplest approach which the user can follow in order to store all key-value pairs in the database. This approach also minimizes the cost for retrieving contiguous entries in cases of time range queries. Furthermore, we facilitate time range queries with the use of iterator abstraction provided by the system to the end user. We concatenate the starting timestamp for the range at the end of the requested key and we request from the system to seek for the first key that is greater or equal to the constructed one. Then we iterate over all keys until we retrieve a key that either has a timestamp greater than the one defined as an end point by the user or if the key has changed.



(a) Avg. Latency

(b) Avg. Throuput

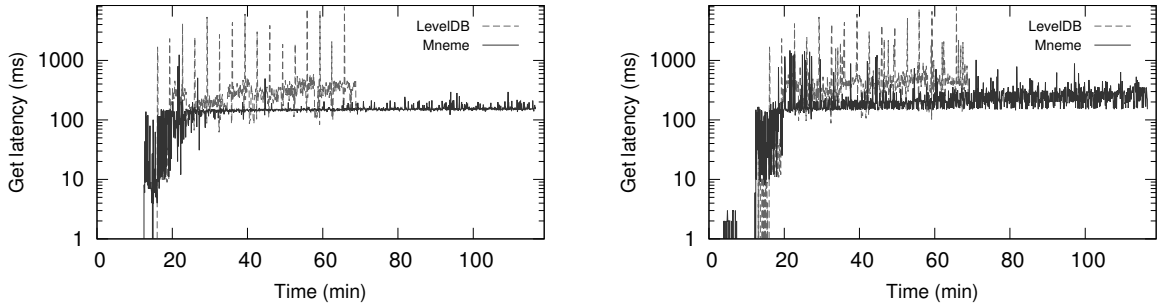
Figure 6.1: In Figure (a) we see the average latency for retrieving all the updates for a key in a timerange. Mneme keeps latency below 100 ms. LevelDB needs more time to retrieve the same amount of data. In Figure (b) we can see the difference in throughput of these systems. Mneme achieves to respond in almost 15 request every second. LevelDB again manages to serve considerably less requests every second

6.2 Query Latency

First we measure the query latency in the case of concurrent puts and gets. On our system a disk seek takes on average 12.9ms allowing max of 79req/s. We configure our system to perform random get with rate 20 req/s. That leads the disk to spend almost 25% of its time to answer the user get requests. Part of the disk's bandwidth is available for insertions and compactions. Due to key randomness we are not able to determine the amount of bandwidth that system may need during the insertion process. Thus, we set the put rate at 2500 requests/s, which is slightly lower than the half of the maximum possible (shown in Figure 6.3(c)). Combining the above settings, the system consumes about two thirds of the total disk bandwidth, allowing the background compactions to consume the rest of the disk bandwidth. The created workload is write-dominated with read/write ratio about 1/125 requests.

When the memory fills up, the put thread is blocked until the system frees space in memory. Hence, the granularity and the duration of a compaction affects system performance. Creating more threads that concurrently request to retrieve data from disk, would lead to an increase of total latency. Yet, this burden is equally affecting both systems so the relative difference between the compared systems remains constant. For simplicity in this thesis, the systems use one read thread.

In Figure 6.1(a) we can observe the relative difference between our system and Lev-



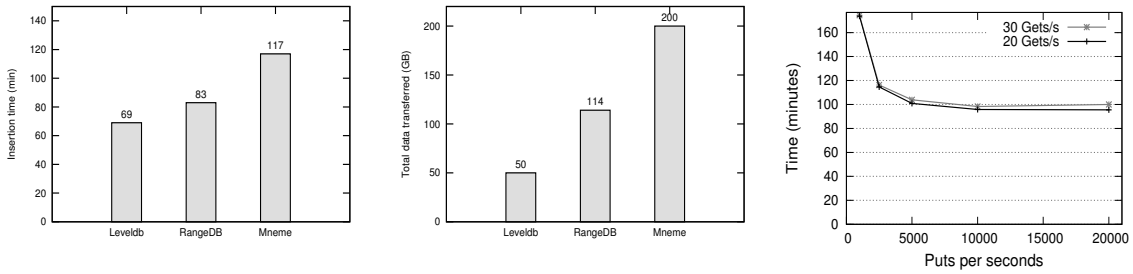
(a) 95th Percentile

(b) 99th Percentile

Figure 6.2: In that Figure we see 95th and 99th percentile of get latency. In subfigure (a) we see that Mneme retrieves 95 percent with almost a constant latency. Our system has similar results for 99th percentile. Spikes in LevelDB indicate that system needes more than 1 second to retrieve the same amount of data

elDB. The experiment runs until a total of 10GB was inserted to the system. We observe that our system almost never surpasses 80 ms for retrieving the appropriate key-value pairs. On the other hand for the same range and the same number of keys, LevelDB requires almost 100 ms to retrieve the appropriate data. Furthermore, we notice that our system shows low variation in the time needed to answer a request. In contrast, the behavior of LevelDB is not stable. That leads to delays in the order of a second. Both systems present similar behavior in the throughput that they achieve every second shown in Figure 6.1(b). Mneme serves more than 15 requests every second, while at the same time LevelDB handles less than 15 requests. Similar unstable behavior leads LevelDB to answer from 10 to 12 requests per second while in cases of background compactions system's throughput falls lower than 5 requests per second.

Inspecting further the time needed to retrieve a range of archived key-value pairs we inspect the 95th and 99th percentile of latency. According to a potential Service Level Agreement [10] users should be able to retrieve the requested data within a predefined time period. In Figure 6.2(a) we observe that 95% of the appropriate key-value pairs are retrieved in just over 100 ms. Again our system achieves a relatively small latency throughout its operation, due to the small period of each compaction. That feature allows Mneme to maintain the same latency upper bound for 99% of all requests (shown in Figure 6.2(b)). On the other hand in both Figures LevelDB needs more time to answer to the same queries. At the same time, it has the same unstable behavior that makes it difficult to maintain an constant upper bound in latency requests.



(a) Total insertion time

(b) Total data transferred

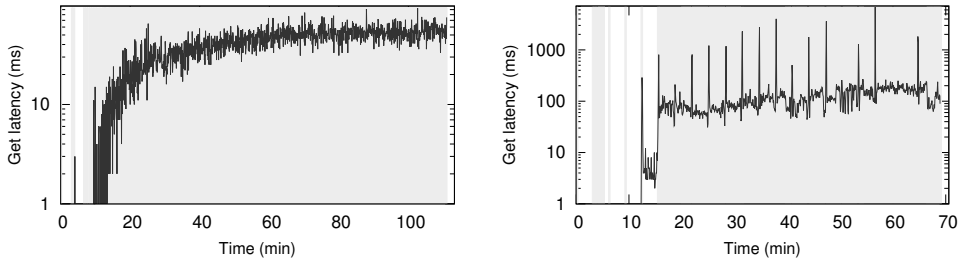
(c) Insertion time for different get/put ratio

Figure 6.3: (a) We present the total amount of time to insert 10GB. There is sufficient difference between Mneme and the original LevelDB. We see also the time that RangeMerge would need to store the same size of unique data. (b) The total amount of transferred data reveals that our system transfers four times the amount of data that LevelDB transfer. (c) The insertion time while keeping the number of request constant and changing the put rate

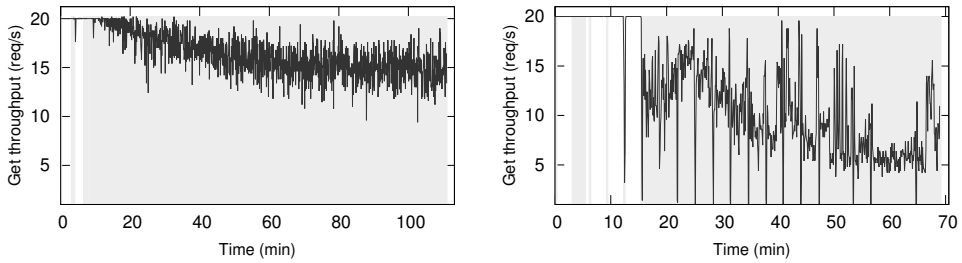
6.3 Insertion Time

In this section we inspect the total time needed to insert 10GB of data to each system. We notice from Figure 6.3(a) that the insertion in our system is significantly slower than the LevelDB. The main reason behind this delay is that our insertion algorithm is designed to firstly provide low latency and stable characteristics when user retrieves data from the system. On the other hand, LevelDB is write optimized, thus it is capable of faster inserting the same amount of data even from the original implementation of RangeMerge.

Furthermore, we see in Figure 6.3(b) that the total size of transferred data is 4 times larger than the corresponding size for LevelDB. Also in comparison with the original algorithm of RangeMerge the total amount of transfer data is almost double. This happens because every time a compaction is triggered, the system should read one or two files from the Primary Level and one file from the Historical Level. Additionally, another drawback comes from the fact that storing all the updates for each key may split the memory in many small pieces. This results in writing only a small part of the total available memory. So, in our system every time a small portion of the total memory is compacted with 2 or 3 larger on disk files. This results in reading and writing the same data while adding only a small part of new key-value pairs. The same behavior may occur in the original RangeMerge algorithm.



(a) Average Latency for Mneme(left) and LevelDB(right)



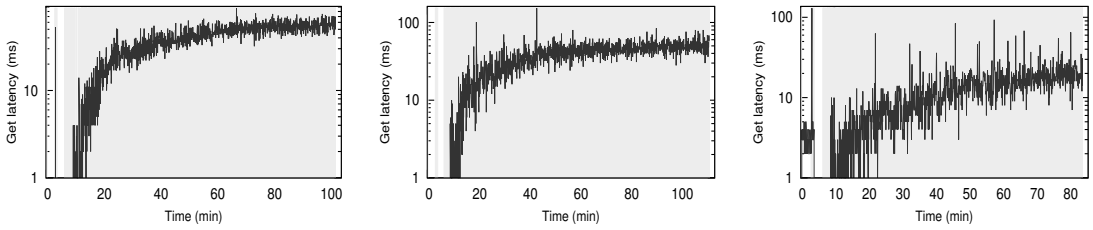
(b) Average Latency for Mneme(left) and LevelDB(right)

Figure 6.4: (a)the relative difference between the system when the a parameter is 0.4. Mneme presents a more stable behavior. (b) Demonstrates the number of requests answered by each system. Again Mneme is able to retrieve more requests every second while also preserving that number almost stable.

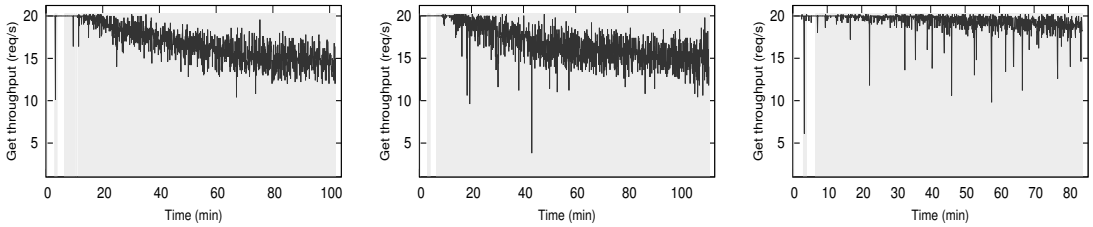
Mneme and LevelDB have a totally different storage structure. That makes it difficult to reason about the amount of effort that a designer would need to make in order to achieve a structure similar to the one presented in Mneme. The two systems are designed to fulfill different needs. So we let as a future work to find a method where insertion of key-value pairs will minimize the total insertion time, while at the same achieves the performance of Mneme.

6.4 Key Distribution

There are many cases where the key indicators follow different distributions. In this section we study the performance of our system compared to LevelDB while inserting key-value pairs that follow either the Zipfian or the Uniform distribution.



(a) Average latency for Mneme and the a parameter alternatively set equal 0, 0.8 and 1.1



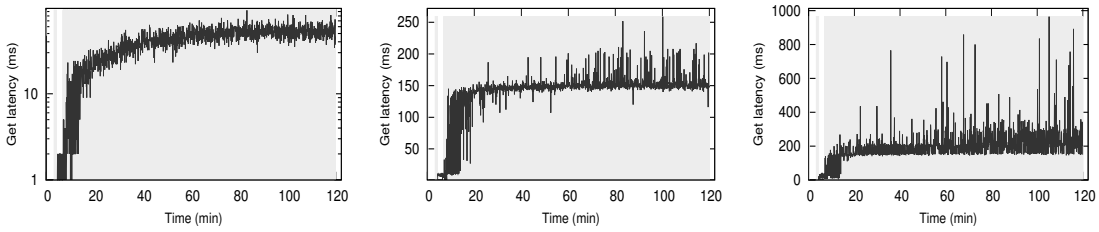
(b) Average throughput for Mneme and the a parameter alternatively set equal to 0, 0.8 and 1.1

Figure 6.5: Our system maintains similar performance for almost each tested value of a parameter of the Zipfian distribution(Figure 6.5(a)). Throughput also has the same behavior for each different value(Figure 6.5(b)).

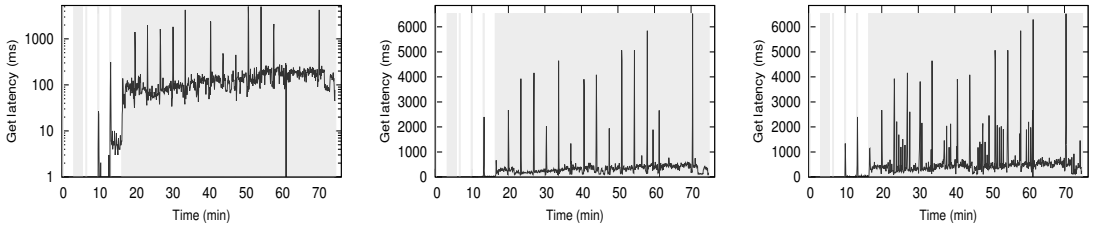
6.4.1 Zipfian Distribution

When the keys follow a Zipfian distribution, a different number of updates is applied to each key. Based on the a parameter of the distribution, every new key-value pair is more or less likely to update a preexisting one. Furthermore, as the value of a increases the total amount of updates becomes larger. Also, few keys receive more updates than other. This leads our system to slit the memory unevenly.

Figure 6.4 presents the relative difference between our system and LevelDB. The total amount inserted is 10GB and the a parameter is set to 0.4, meaning that most recent key-value pairs occupy almost 985MB while most updates for these keys occupy 8.7GB. At Figure 6.4(a) and at all similar figures the gray background indicates that the system performed a compaction while replying to the corresponding requests. Our system performs small compactions, which does not affect the get latency and allow system to serve around 15 requests every second. On the other hand the LevelDB performs large compactions which seems to affect both the latency and the throughput. From the Fig-



(a) Average, 95th and 99th percetile Latency for Mneme respectively



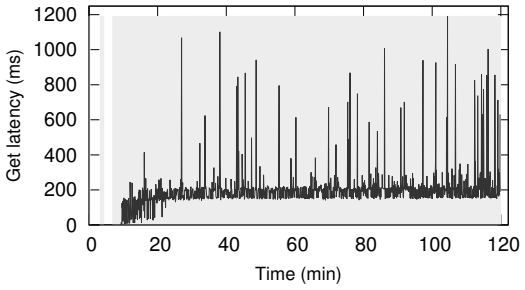
(b) Average, 95th and 99th percetile Latency for LevelDB respectively

Figure 6.6: Both systems remain unaffected from the key distribution. Our system still shows significantly lower latency than LevelDB. Also the 95th and 99th percentiles indicate that our system guarantees an almost stable latency for the majority of requests.

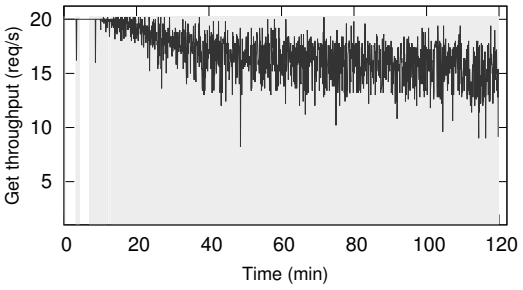
ure 6.4(b) we see that when LevelDB performs heavy compactions, throughput shows a large variation which leads the system to serve almost none request.

To further study the performance characteristics of our system, we imported data using different values for the parameter a. We evaluated the system for parameters of a equal to 0, 0.8, 1.1 which inserts 1.05GB, 950MB and 566MB of different keys respectively. Although the number and the size of inserted keys vary, our system achieves similar latency and throughput for every tested case. Furthermore, the last graph from Figure 6.5(a) illustrates the case where data stored at Primary Level gets smaller. In this case, the system serves the get requests faster and handles larger number of requests per second. This happens due to the fact that more than one MemTables refer to the same files in the Primary Level. These files are cached in system's main memory, minimizing the latency for processing records stored in the respective files.

6.4.2 Uniform Distribution



(a) 99th percetile of Latency for Mneme



(b) Throughout for Mneme

Figure 6.7: (a)99th percentile of Latency when half of the inserted key-value pairs are unique and the other half are updates for preexisting ones. Throughput graph (b) reveals a higher variation due to slightly heavier compactions

Increasing the value of the Zipfian parameter creates a smaller footprint for the different key-value pairs. This leads to unrealistic scenarios where only 1% (or less) of total data create new entries in the Primary Level. To further stress our system we created and inserted keys based on a uniform distribution. More specifically, every time that the benchmark tool inserts a key-value pair it chooses a number in the range [0, 1]. If this number is smaller than a predefined threshold system creates a new key-value pair else it updates a preexisting one. Based on the fact that every number in this range has same probability to be chosen, by tuning the parameter we determine the percentage of unique keys that the tool will create.

We study our system's behavior under two different thresholds while inserting 10GB. First we inspect the case where the amount of different keys is approximately 20% of the total inserted data. This is twice the amount of different keys compared to the Zipfian distribution presented in the previous section. In Figure 6.6 both systems present similar performance for history range requests with the Zipfian distribution. Our system is able to reduce the average latency by almost an order of magnitude in comparison to LevelDB. Furthermore in both systems 95th and 99th percentile appear to have the same form, with the 99th percentile of latency slightly increased for both systems.

In Figure 6.7 we demonstrate the 99th percentile of the latency and the average throughput in the case of a uniform distribution, where half of the dataset consist of unique keys and the rest are updates for the preexisting keys. Latency (Figure 6.7(a)) for most of get requests is slightly more than 100ms which again seems to follow the same pattern as in previous distributions. Additionally our systems maintains its through-

Table 6.1: Mneme slightly serves almost 17 requests per second. Average latency of LevelDB varies significantly due to background compactions

Latency(ms) and Throughput of history range queries

Key size	Average		95th		99th		Throughput	
	Mneme	LvlDB	Mneme	LvlDB	Mneme	LvlDB	Mneme	LvlDB
16	36.67	132.55	127.82	316.20	184.49	445.89	16.92	11.71
25	36.76	129.87	12.03	302.69	180.94	438.38	16.95	12.08
50	37.18	150.317	128.87	334.49	190.44	450.73	16.90	11.20
100	38.30	135.71	130.97	334.82	206.59	453.9	16.75	10.79

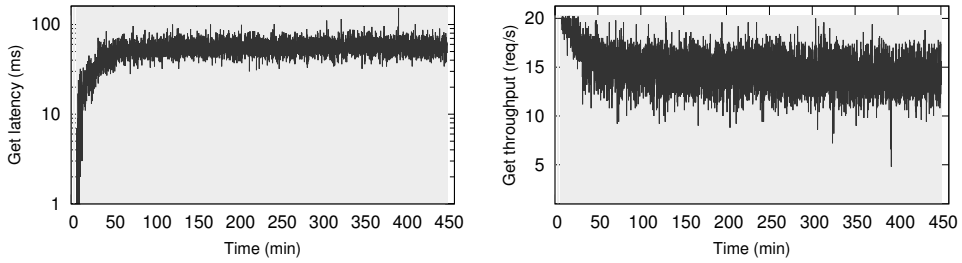
put(Figure 6.7(b)) performance and serves from 13 to 17 requests per second. Compared to Zipfian distribution, the throughput in our system has a higher variation. This happens because the number of files in the Primary Level increases. The consequence is that the files participating in a compaction increases. That leads our system to perform heavier compactions. We can assume that while the number of files for the ratio between the files stored in the Primary Level and the files stored in Historical Level increases, our system performs similar to Remerge [22] sort-merge algorithm.

6.5 Key size and Scalability

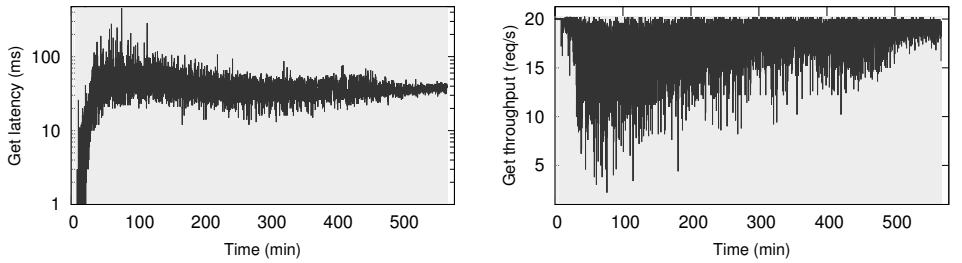
Previous research [33, 25] argues that the allocated size for each key should be 68 (or less) bytes. On the other hand, in Bigtable [6] may occupy up to 64KB, with a typical size between 10 to 100 bytes. Therefore, we insert in both systems records of different key sizes. We tried to insert keys that occupy 16, 25, 50 and 100 bytes, while at the same time each corresponding value needs 1KB. In practice both systems do not have any strict restriction for the size because they hand records as arbitrary strings.

Table 6.1 summarizes the relative difference between the two systems. In section 6.4 we saw that key distribution has a minor impact on system's performance. Nevertheless, the number of update request may cause heavier compactions in our system. The evaluation shown in table 6.1 was tested for the ostensibly worst tested case, where only half of the incoming key-value pairs are updates. Our system increases the average response time only by 1.6 ms as the size of the key increases. Also it manages to answer in almost 17 requests. In contrast LevelDB shows a smaller serving ability. Finally the average time that LevelDB needs to answer a range query is almost 4 time higher than in our system.

The number of the key-value pairs inserted for the 10GB dataset is between 9.7 to 10.5 millions. This is only a small part of the data that a production server handles. Thus,



(a) Average Latency and throughput for 20% of non-updated keys



(b) Average Latency and throughput for 80% of non-updated keys

Figure 6.8: In the case where 20% (top) of the dataset corresponds to different keys Mneme maintains the same latency and throughput as this for the dataset of 10GB. In the case where the vast majority of inserted keys are unique (bottom), the average latency seems to follow the same pattern with previous experiments. In the end our system gains the characteristics of the original RangeMerge

we examine the scalability characteristics of our system by inserting 20GB of data. In order to stress the system, we keep the key size constant at 100 bytes and the value size at 1KB. That correspond at almost 20 million key-value pairs. Additionally we kept the in-memory buffer constant at 512MB (similar to previous experiments). This forces our system to perform more compactions while serving the incoming requests.

In the first set of this experiment (Figure 6.8(a)) we tested the case of inserting approximately 4GB(20%) of the unique key-value pairs which the system distributes uniformly. Average latency has slightly greater values than these presented in the case of 10GB, but our system achieves almost the same latency for all requests. Throughput demonstrates a higher variation due to concurrent compactions but the system is capable of serving more than 10 requests at any moment. In the second case we further pushed our system by forcing it to accumulate a workload which contains 16GB(80%) unique keys. Interestingly

our system starts to increase the latency for every request, which also has an impact to the average throughput, but after the 150 minute it keeps an almost constant latency well below 100ms. That happens due to the fact that Mneme maintains also the characteristics of original RangeMerge. More specifically, at this point only a few files form the Primary Level are related with some files from the Historical Level. Our system has taken that into consideration and has split the memory based on the range of the files in the first level. Thus, the system achieves to minimize the impact of each compaction.

6.6 Summary

To study the performance characteristics of our idea, we modified RangeDB, a key-value store based on Google's LevelDB. We test our system under many different parameters of a synthetic workload and we demonstrate that it achieves to minimize historical range queries latency. At the same time it has a minimum sensitivity to background compactions and has a stable behavior. Furthermore we compared our system with the original LevelDB, which we modify to keep every key-value pair that is inserted to the system. We observe that our system achieves better performance than LevelDB for every range get operation.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The unique demands placed by maintaining multiple versions for the same key indicate that modern key-value pairs lack the design characteristics which allow the system to efficiently handle this kind of workload. A persistent general-purpose storage backend, is one of the most crucial components of todays large scale systems. In our vision, these systems can be used by a variety of applications in the range of monitoring every day activity to efficiently handling transactions. The retrieval of older values in such applications is important, especially when retrieved data should be consistent. Advanced key-value stores use either a general purpose snapshooting mechanism or they do not provide any guarantee for older values. Motivated from the emerging need of efficiently storing and retrieving large number of key-value pairs we thought a new approach for making a key-value store persistent.

We used a well known key-value store, and we demonstrated the inefficiencies that its sort-merge algorithm implies, when a user should store every update that has been inserted to the system. These inefficiencies lead to unnecessary latency and low request throughput. The main reason behind low performance are the compactions executed in the background. Then, we introduced the idea of storing older values based on the RangeMerge compaction algorithm. In order to implement Multi-Version RangeMerge we

create the Mneme system. Our system is separated in two components: the in memory skip-lists, which sorts and temporally stores incoming records, and the two storage layers, which are responsible for storing only separately the most recent and the older values. Also, we tuned the compaction mechanism to take into consideration the updates values that may be overwritten. Thus, we changed the way that system splits the provided memory in smaller ranges. Using a microbenchmark, we evaluated the performance for both our system and the original LevelDB. We found that the performance of our system is not affected neither from the key distribution nor from the size of the key. Furthermore we saw that on average retrieves ranges of updates three to four times faster than the compared system. Also we demonstrated that our system maintains the same performance for larger workloads. Overall our system provides efficient storage and fast retrieval of older values for the same key while sacrificing a part of its insertion performance.

7.2 Future Work

In the future we firstly plan to make our system more efficient for data insertion. Performing better splits in-memory can lead system to efficiently flush larger parts of the in-memory buffer. Choosing the appropriate ranges is not trivial because while the flushed size is getting bigger the compactions are getting heavier.

Another approach that we could follow is to consider a way to handle data in more than two dimensions. In the current work we take into consideration only the key and the time dimension. When storing data that correspond to different dimensions closely on disk, application can benefit from the sequential disk bandwidth. Deciding which data should be stored closely, affects the performance and the type of the queries that can be performed on the system.

Furthermore, we plan to extend the experimental measurements of our prototype implementation, to validate further the contributions of our study and emphasize the offered performance gains. Therefore, initially, we aim to examine the behavior of our system in production systems that use LevelDB as their storage backend. In particular, a real workload with varying number of clients applying concurrent read requests of archived data to the same storage server, will provide a more realistic environment in terms of the ability of Multiversion RangeMerge to serve real-world workloads.

Finally, we intend to examine the behavior of historic RangeMerge under the Yahoo! Cloud Serving Benchmark [8] suite. YSCB provides a framework and many common sets of workloads for evaluating the performance of various key-value stores. It simulates core workloads which fill out the space of performance trade offs made by the larges NoSQl

database management systems. Additionally it provides useful workload generators which can produce synthetic traces. The form of these traces can be defined by the user through the appropriate modification of the provided client.

Bibliography

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[3] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, December 1996.

[4] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. Technical report, 2003.

[5] Gerth StΓ'Elting Brodal, Konstantinos Tsakalidis, Spyros Sioutas, and Kostas Tsichlas. *Fully Persistent B-Trees*, chapter 51, pages 602–614. 2012.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

[7] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, Robert N. Sidebotham, and Transarc Corporation. The Episode file system. pages 43–60, 1992.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st*

ACM Symposium on Cloud Computing, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[11] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86 – 124, 1989.

[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[13] Pascal Felber, Marcelo Pasin, Etienne Rivière, Valerio Schiavoni, Pierre Sutra, Fábio Coelho, Miguel Matos, Rui Oliveira, and Ricardo Vilaça. On the support of versioning in distributed key-value stores. In *33rd IEEE International Symposium on Reliable Distributed Systems - SRDS*, Nara, Japan, October 2014.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[15] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[16] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng

Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[17] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[18] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[19] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 380–390, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[20] Leveldb: A fast and lightweight key/value database library by google. `http://code.google.com/p/leveldb/`, May 2011.

[21] G. Margaritis and S.V. Anastasiadis. Efficient range-based storage management for scalable datastores. *Parallel and Distributed Systems, IEEE Transactions on*, 25(11):2851–2866, Nov 2014.

[22] Giorgos Margaritis. *Efficient Storage Indexing of Strutured and Unstrutured Data.* PhD thesis, University of Ioannina, Ioannina, 2014.

[23] Giorgos Margaritis and Stergios V. Anastasiadis. Range-aware logging for fast durable storage. In *USENIX Symposium on Operating Systems Design and Implementation*, Bloomfield, CO, October 2014. (poster).

[24] The bsd 3-clause licence. `http://opensource.org/licenses/BSD-3-Clause`, 1998.

[25] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[27] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[28] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):1–32, August 2013.

[29] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[30] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[31] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[32] Project voldemort:a distributed database. `http://www.project-voldemort.com/voldemort/`, 2009.

[33] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, July 2015. USENIX Association.

[34] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 155–168, New York, NY, USA, 2012. ACM.

Short Vita

Christos Theodorakis was born in Ioannina, Greece in 1988. He was admitted at the Computer Science Department of the University of Ioannina in 2006. He received his BSc degree in Computer Science in 2012 and he is currently a postgraduate student at the same department. He is a member of the Systems Research Group of the University of Ioannina since 2012. His main research interests lie in the field of large scale storage systems and cryptography.