UNIVERSITY OF CALIFORNIA
RIVERSIDE

I/O Optimization in Big Data Storage Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Mohiuddin Abdul Qader

September 2018

Acknowledgments

First, I would like to thank my advisor, Professor Vagelis Hristidis. In the past five years, his insightful guidance in research had always pointed me to the new topics and ignited me with novel ideas. He is very responsive whenever I need his input in my research. I appreciate that he put trust on me through my PhD study and gave me enough flexibility on research. I would also like to thank other committee members, Professor Vassilis Tsotras, Professor Eamonn Keogh and Professor Ahmed Eldawy, as well as former committee member Professor Victor Jordan, for their helpful comments on my research.

I thank all my collaborators. Professor Vassilis Tsotras and Mike Carey always brought novel ideas and comments into big data projects. I feel lucky to work with Shiwen Cheng during my first research project. I appreciate his mentorship which trained me to be a professional researcher. I have to thank all my labmates in the Database Lab at UC Riverside. They also contributed to build this collaborative, transparent and enjoyable work environment. Moloud Shahbazi, and Shiwen Cheng instructed me and gave me precious advises on research. We also had many fruitful discussions with Nhat Le, Abhinand Menon, Steven Jacobs and Ildar Absalyamov during our collaborations.

Finally, I would like to thank my family for all the unconditional support. My beautiful wife, Refat Amin, is always supportive and helpful in every aspect of my life. I am forever in her debt. I would like to thanks my parents, without their sacrifices and efforts, I wouldn't reach this success. I would also thank my sister, Fatema Tuz Johra, and my uncle Mahfuz Ahmed, for showing me a path on how to succeed and for their unconditional support.

To my wife and parents for all the support.

ABSTRACT OF THE DISSERTATION

I/O Optimization in Big Data Storage Systems

by

Mohiuddin Abdul Qader

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Vagelis Hristidis, Chairperson

The age of Big data has transformed into the era of Internet of Things (IoT) where massive scale data is generated, stored, and used by a diverse set of physical objects: devices, vehicles, buildings, software, sensors, GPS and networks. It has become an open challenge for researchers in academia and industry to find the best ways to ingest, replicate, manage, read and deliver this massively growing data efficiently to millions of users in real time. Big data storage systems – especially NoSQL databases like LevelDB, Cassandra, BigTable and AsterixDB – have become extremely popular in the last decade for managing large amounts of data that don't require the stringent concurrency or transaction management guarantees. In such settings, NoSQL systems achieve high rates of data writes. My research interests focus on Input/Output (I/O) optimizations of such state-of-the-art big data storage systems. Specially my thesis aims mainly at three aspects of optimization: Indexing, Partitioning and Replication.

a) Indexing of non-key attributes: Current state-of-the-art big data storage systems have limited support for secondary attribute lookup queries or continuous lookup queries. To tackle these limitations, first we introduce and implement five secondary indexes on a NoSQL database.

Specifically, we use the popular LevelDB database, which employs Log-Structured Merge-Tree (LSM) for organizing its data. Our comprehensive experimental study and theoretical evaluation provide empirical guidelines for optimal choice of secondary index, depending on the workload of different applications.

b) Indexing for publish-subscribe systems: We propose and compare several publish/subscribe storage architectures, based on the popular NoSQL LSM storage paradigm, to support high-throughput and highly dynamic continuous lookup queries. Our framework naturally supports subscriptions on both historic and future streaming data, and generates instant notifications.

c) Data partitioning: We create optimization techniques for spatial indexes via intelligent partitioning. Currently NoSQL based databases do not offer any spatial partitioning to achieve faster spatial query response. We propose a level-based organization of disk components and two novel component merge techniques that leverage their spatial properties.

d) Data replication: Another important feature of big storage systems is its availability and reliability, which is achieved through replication. Paxos is a widely used replication policy to ensure the replicas are in sync. We develop an I/O optimized Paxos-based fault-tolerant block storage replication engine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the age of big data, more and more services are required to ingest high volume, velocity and variety data, such as social networking data, smartphone apps usage data and click through data. NoSQL databases were developed as a more scalable and flexible alternative to relational databases. NoSQL databases, such as HBase [39], Cassandra [76], Voldemort [34], MongoDB [59], AsterixDB [7] and LevelDB [55] to name a few, have attracted huge attention from industry and research communities, and are widely used in products. Through the use of Log-Structured Merge-Tree (LSM) [84], NoSQL systems are particularly good at supporting two capabilities: (a) fast write throughput, and (b) fast lookups on the primary key of a data entry (see Section 2.1 for details on LSM storage framework).

Most research on NoSQL databases has focused on how to optimize the core operations: read and write key-value pairs. However, more capabilities are needed to allow more demanding applications to take advantage of NoSQL systems. We study how NoSQL systems can be adapted and extended to support such application needs. We consider the following needs: (a) support non-key queries; (b) support continuous queries in a publish-subscribe setting; (c) support spatial queries; (d) support efficient and safe replication in a distributed environment. We developed algorithms and built prototypes to support these requirements.

(a) Support non-key queries: For instance, if a tweet has attributes such as tweet id, user id and text, then it would be useful to be able to efficiently return all (or the most recent) tweets of an user. However, supporting secondary indexes in NoSQL databases is challenging, because secondary indexing structures must be maintained during writes, while also managing the consistency between secondary indexes and data tables. This significantly slows down writes, and thus hurts the system's capability to handle high write throughput which is one of the most important reasons why NoSQL databases are used. This trade-off indicates a challenging research direction: which secondary indexing techniques are easy to adopt on pure key-value stores and are efficient for different workloads.

(b) Support continuous queries in a publish-subscribe setting: Publish/Subscribe systems usually support real-time continuous queries and are used in many applications, such as social networks, messaging systems, and traffic alerting systems. State-of-the-art publish/subscribe systems are efficient when the subscriptions are relatively static – for instance, the set of followers in Twitter – or can fit in memory. However, now-a-days, many Big Data and IoT based applications follow a highly dynamic query paradigm, where both continuous queries and data entries are in the millions and can arrive and expire rapidly. As a running example application, consider users who are driving and subscribe to nearby traffic or other incidents (accident, crime, roadwork, fire, natural disaster, protest etc.). Every time a user moves to a new location (i.e., a geospatial cell) they need to subscribe to events/publications in the new location for a time duration starting from the near past to the near future – e.g., to know what happened in the last one minute and what will happen in the next one minute until the user moves to another cell. These subscriptions are *highly dynamic* as they come and go every few seconds. At the same time, users are publishing incidents. As millions of users are moving and subscribing to events, these large streams of subscriptions and publications must be stored and managed efficiently. Google's spatial notifications closely follow this moving subscriber example. Here, when subscribers go to a store, Google pushes to them, as a mobile notification, the map of the store, specials deals, and so on. As another application, an airplane continuously queries for data in its path such as turbulence, wind, air pressure, etc. This brings us to another interesting research direction on how to build a storage framework by optimizing NoSQL databases for publish/subscribe systems that can support high-throughput and highly dynamic subscriptions queries on both historic and future streaming data and generate real-time instant notifications.

(c) support spatial queries: Spatial indexes in traditional relational databases supported spatial queries in pre-*big data* era. However, the volume and ingestion rate of spatial data is increasing rapidly in modern applications such as Social Media, Disaster Management, Climate Science, Politics, Urban Traffic, Supply Chain Management, Marketing/Advertising etc. These applications need to ingest and maintain billions of spatial data coming in rapid pace. Many popular big data storage systems begin supporting spatial indexes on top of their NoSQL framework (AsterixDB LSM-Rtree, GeoMesa, STEHIX on Hbase etc). But they all built a separate module on top of existing framework which is optimized for non-spatial indexes. These techniques are usually good for ingestion and also show faster spatial queries than non-spatial indexes. But their low-level storage organization are not optimized for spatial data. For instance, their disk components share spatially overlapped regions. This motivates us to our next research direction: can we spatially organize and partition database components in a optimized way so that their spatial overlapping are minimized. Therefore, we can support significantly faster spatial queries. This will also reduce extra efforts to build separate module on top of an LSM database.

(d) support efficient and safe replication in a distributed environment: Current big data systems become cloud-based now-a-days and they want to be reliable, available and fault-tolerant in a distributed environment. This triggers an important challenge: how can we make database storage available without imposing too much overhead on its reads and writes in an unreliable distributed cloud network. That is why many cloud databases use replicated shared block storage instead of local file storage to store data. Block Storages like AWS-EBS [25], Rackspace [69], Ceph [95] become very popular nowadays as they can provide fixed-sized raw storage capacity. Each storage volume can be treated as an independent disk drive and controlled by an external server operating system [37]. The block storage services are expected to be highly available and fault tolerant in a geo-distributed cloud. Typical block storage volume has a simple mapping layer to export the virtual device blocks to physical blocks. This map and as well as the physical data need to be replicated. For that we need a high performance and I/O efficient fault tolerant system which will act as a stand-alone replication engine maintaining consistency and reliability of the block storage throughout the distributed cloud. Paxos [78] is a distributed consensus protocol which can work in a network of unreliable processors. Although there exists some replicated block store which does not rely on consensus algorithms (e.g. Linux DRBD [31]), but Paxos is theoretically proven as accurate and used where durability is highly required (e.g. to replicate a file or a database). But there has been little work so far on how we can optimize Paxos for block storage replication. This drives to our next research direction: build an I/O optimized and fault-tolerant replication engine for block storage systems on top of Paxos protocol.

1.2 Research Problems and Contributions

In this thesis, I follow three main directions to optimize big data storage systems. First, I studied LSM-based indexes and developed efficient secondary indexes to support both static and real-time continuous queries on secondary attributes. Second, I developed an I/O optimized partitioning technique for spatial big databases. Third, I built an I/O efficient replication engine so that the big data stores can maintain their availability in unreliable cloud network.

Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases

Follow this first direction, I study on how existing different NoSQL databases added support for secondary indexes. I find that these works are fragmented, as each system generally supports one type of secondary index and may be using different names or no name at all to refer to such indexes. As there is no single system that supports all types of secondary indexes, no experimental head-to-head comparison or performance analysis of the various secondary indexing techniques in terms of throughput and space exists. In this thesis, I present a taxonomy of NoSQL secondary indexes, broadly split into two classes: *Embedded Indexes* (i.e. lightweight filters embedded inside the primary table) and *Stand-Alone Indexes* (i.e. separate data structures). To ensure the fairness of my comparative study, we built a system, *LevelDB++*, on top of Google's popular open-source LevelDB key-value store. There, I implemented two Embedded Indexes and three state-of-the-art Stand-Alone indexes, which cover most of the popular NoSQL databases. The comprehensive experimental study and theoretical evaluation show that none of these indexing techniques dominate the others: the embedded indexes offer superior write throughput and are more space efficient, whereas the stand-alone secondary indexes achieve faster query response times. Thus, the optimal choice of secondary index depends on the application workload. This thesis provides an empirical guideline for choosing secondary indexes. We describe the details of this work in Chapter 3.

High-throughput Publish/Subscribe on top of LSM-based Storage

In this problem, I focused on building storage frameworks for big active data and support efficient continuous queries on big data stores. The storage modules of existing Publish/Subscribe systems have several *limitations*, which make them inadequate for modern IoT applications. The goal is to design and build a Publish/Subscribe storage system with the following properties:

1. Scale to millions of dynamically changing subscriptions and publications per minute, that is, both subscriptions and publications arrive and expire at a rapid pace.

2. After a new publication, immediately identify and notify matching subscribers (as in traditional database triggers, discussed in Section 4.2), that is, not follow a periodic check paradigm.

3. Subscriptions or publications may have validity time periods. Subscriptions may request past data in addition to future data.

4. The subscriber should assume that all the matching publications should reach her, that is, there is no data loss, which may occur with periodic check systems.

To support these properties, I proposed and implemented several publish/subscribe storage architectures, based on the popular NoSQL Log-Structured Merge Tree (LSM) storage paradigm, to support high-throughput and highly dynamic publish/subscribe systems. Our framework naturally supports subscriptions on both historic and future streaming data and generates instant notifications. We also extend our framework to efficiently support self-joining subscriptions, where streaming pub/sub records join with past pub/sub entries. Further, we show how hierarchical attributes, such as concept ontologies, can be efficiently supported; for example, a publication's topic is "politics" whereas a subscription's topic is "US politics." We implemented and experimentally evaluated our methods on the popular LSM-based LevelDB system, using real datasets, for simple match and self-joining subscriptions on both flat and hierarchical attributes. Our results show that our approaches achieve significantly higher throughput compared to state-of-the-art baselines. We elaborate the problem and our solutions in Chapter 4.

Spatial-LSM: Efficient Spatial Partitioning in LSM-based databases

Following this direction, I studied on how LSM-trees are optimized for one dimensional indexes. Based on that, I designed techniques that can be applied to optimize LSM-tree for two-dimensional spatial indexes. As the disk components in the LSM-tree are immutable, we need to find a good way to reorganize them during background merge compaction. The goal is not only to minimize the spatial overlap between them so that spatial queries can be faster, but also to minimize overhead on insertion and Compaction and to write amplification factor. Our level based merge compaction techniques effectively partition the disk components such a way so that there is no spatial overlapping between them inside a level. This organization helps to prune more disk components for given spatial queries. This technique closely follows one dimensional index compaction for Leveldb. Our experimental results show that our approaches achieve faster spatial query response time with slightly increasing write amplifications. The study of this problem and the details of our approaches are presented in Chapter 5.

Efficient Paxos-based Block Storage Replication Engine

Paxos based distributed block storage systems are not explained in depth in literature or open source world. The main reason is that it is very hard to implement Paxos where performance heavily depends on implementation. There is some academic work which proves that Paxos state machine replication is the basis of high performance data store (i.e. Gaois [16], Gnothi [94]). But their work is close source and there is not enough technical details on how a logical volume can be efficiently separated from actual physical storage in different replicas. I study on how a Paxos variation Multi-Paxos [85] which replicates a Paxos Log in a distributed cluster can be used to implement a reliable block storage replication engine. Based on our study, we designed and implemented a fault-tolerant block storage system Hydra which performs Log Replication (i.e. State machine replication) using WeChatâĂŹs open-source Paxos library PhxPaxos [63] [98]. Also, we invent a IO optimized write path for block storage in our architecture. Overall, our system is designed in way that ensures availability, strong consistency, atomicity, reliability, IO efficiency in distributed cloud storage. We give details for our replication engine in Chapter 6.

Chapter 2

Foundations

2.1 LSM tree

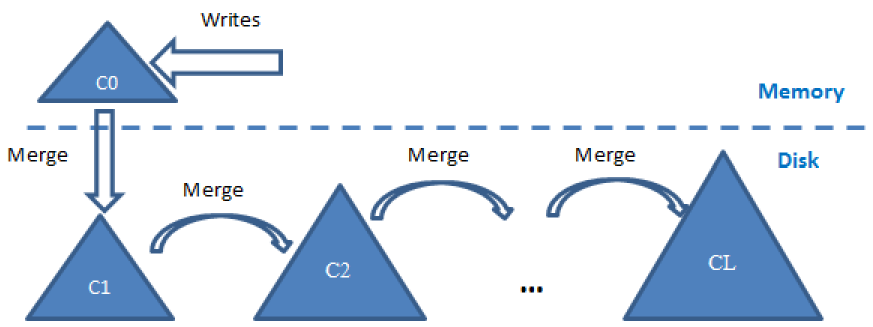


Figure 2.1: LSM tree components.

An LSM tree generally consists of an in-memory component (level in LevelDB terminology) and several immutable on-disk components (levels). Each component consists of several data files and each data file consists of several data blocks. As depicted in Figure 2.1, all writes go to in-memory component (C0) first and then flush into the first disk component once the in-memory data is over the size limit of C0, and so on. The on-disk components normally increase in size as shown in Figure 2.1 from C1 to CL. A background process (compaction) will periodically merge the smaller components to larger components as the data grows. Delete on LSM is achieved by writing a tombstone of the deleted entry to C0, which will later propagate to the component that stores this entry. LSM is highly optimized for writes as a write only needs to update the in-memory component C0. The append-only style updates mean that an LSM tree could contain different versions of the same entry (different key-value pairs with the same key) in different components. A read (GET) on an LSM tree starts from C0 and then goes to C1, C2 and so on until the desired entry is found, which makes sure the newest (valid) version of an entry will be returned. The older versions of an entry (either updated or deleted) are obsolete and will be discarded during the merge (compaction) process. Conventionally, the in-memory component (C0) is referred as MemTable and the on-disk components are referred as SSTable.

2.2 SSTable in LevelDB

Here we present some storage details of LevelDB [55] as background for ease of understanding of the paper. Level-$(i+1)$ is 10 times larger than level-i in LevelDB. Each level (component) consists of a set of disk files (SSTables), each having a size around 2 MBs. The SSTables in the same level may not contain the same key (except level-0[1]). Further, LevelDB partitions its SSTables into data blocks (normally tens of KB in size). The format of an SSTable file in LevelDB is shown in Figure 2.2. The key-value pairs are stored in the *data blocks* sorted by their key. Filter Meta blocks store a bloom filter for each block, computed based on the keys of the entries in that block, to speedup GET operations. Then, there are data index blocks, which we call zone maps for primary table.

[1]which is C1 in Figure 2.1 as LevelDB does not number the memory component

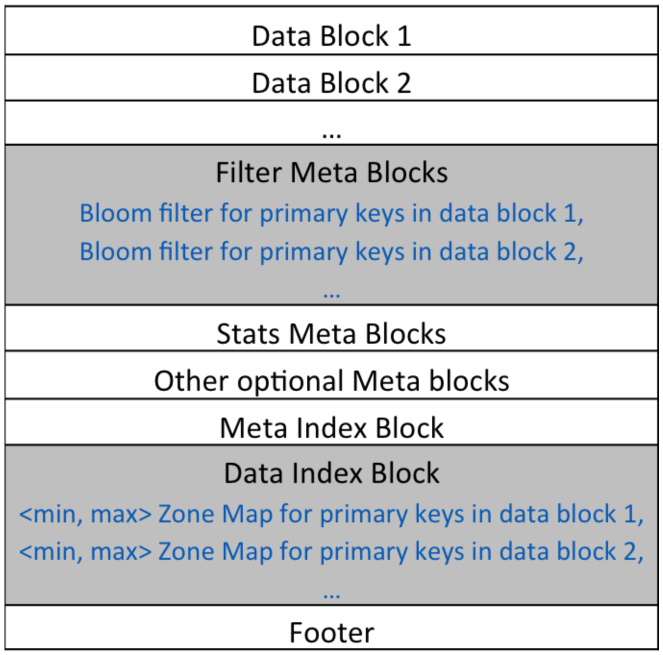

Data Block 1
Data Block 2
...
Filter Meta Blocks
Bloom filter for primary keys in data block 1,
Bloom filter for primary keys in data block 2,
...
Stats Meta Blocks
Other optional Meta blocks
Meta Index Block
Data Index Block
<min, max> Zone Map for primary keys in data block 1,
<min, max> Zone Map for primary keys in data block 2,
...
Footer

Figure 2.2: LevelDB SSTable structure.

2.3 Bloom Filter

Bloom filter [15] is a hashing technique to efficiently test if an item is included in a set of items. Each item in the set is mapped to several bit positions, which are set to 1, by a set of n hash functions. Then, the bloom filter of a set is the OR-ing of all these bitstrings. To check the membership of an item in the set, we compute n bit positions using the same n hash functions and check if all corresponding bits of the bloom filter are set to 1. If no, we return false, else we have to check if this item exists due to possible false positives. The false positive rate depends on the number of hash functions n, the number of items in the set S and the length of the bit arrays m, which can be approximately computed as Equation 2.1. Given S and m, the minimal false positive rate is $2^{-\frac{m}{S}ln2}$ by setting the $n = \frac{m}{S}ln2$.

$$\left(1-\left[1-\frac{1}{m}\right]^{nS}\right)^{n} \approx \left(1-e^{-nS/m}\right)^{n} \tag{2.1}$$

2.3.1 LSM and Stack based merge policies

Below we concentrate on stack-based compaction policies. A component (whether in-memory or on disk) is typically implemented as a B^+-tree. Figure 2.3 illustrates an LSM *flush* operation. Once an *in-memory component* (number 4) reaches capacity, its records are flushed to disk, producing the newest disk component. Then a new (5th) component will be started as the new in-memory component. As the number of disk components increases, sequential groups of

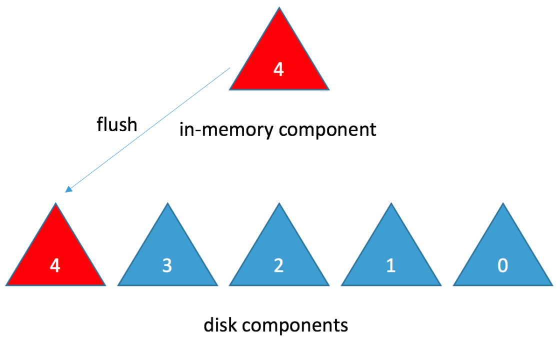


Figure 2.3: Flushing an in-memory component to disk

disk components may be *merged* (Figure 2.4) together based on some *merge policy*. Note that new components always become the top of the stack, so the next in-memory component flushed to disk would be placed after component 4 (and become component 5) in the Figure. This maintains a stack order that is based on freshness of records.

It is important to note here that typical NoSQL systems (including for our experiments) use upsert (insert if new, else replace) semantics. This (combined with the fact that components maintain freshness order) means that there is no need for duplicate-key checking during insertion (which would incur an extra key lookup for every insert). Upserts can be handled simply as new inserts in the in-memory component. Throughout this paper, “insert” will actually use “upsert” se-

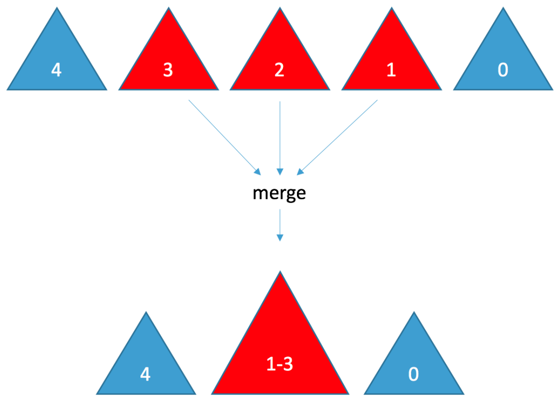


Figure 2.4: Merging a sequence of disk components

mantics. Disk components of an LSM-tree are immutable, meaning that deletes must be handled by ‘anti-matter’ entries into the in-memory component. During merges, older versions of records can be ignored when creating the new disk component, as can older records that are now marked as deleted. During a scan, resolution of deletes and updates can occur by using a heap of sub-cursors sorted on <key, component id> where each sub-cursor operates on a single component. This allows records with duplicate keys to be resolved together (only the version on the newest component is valid). Key lookups can be further optimized. Since components are ordered on freshness, a key lookup simply accesses the components from newest to oldest, until the first instance (the newest version) of a key is found. When compared to B^+-trees, LSM-trees make sacrifices on read performance in order to acheive scalable write throughput [68]. Specifically, the time per read will increase with the size of the component stack. This is mitigated by merges (because they reduce the stack size). Merge (or “compaction”) operations take time proportional to the size of the components being merged. This means that writes incur some amortized merge cost on the system. Any given merge policy will make some trade-off between the total read performance and the total write performance.

2.3.2 Asterixdb storage management

Datasets are managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes[7]. LSM-trees [84] are well-known for providing superior performance for insert-intensive workloads by batching updates into a component of the index that resides in main memory- an in-memory component . When the space occupancy of the in-memory component exceeds a specified threshold, its entries are flushed to disk forming a new component - a disk component . As disk components accumulate on disk, they are periodically merged together subject to a merge policy that decides when and what to merge. The benefit of LSM-trees comes at the cost of possibly sacrificing read efficiency, but, as shown in [90], these inefficiencies can be mostly mitigated. AsterixDB has adopted a framework for converting a class of indexes (including conventional B+ trees, R trees, and inverted in- dexes) into LSM-based secondary indexes, allowing higher data ingestion rates. In fact, for certain index structures such as the LSM R-tree, results have shown[7] that an LSM-based version of an index can be made to significantly outperform its conventional counterpart for both data ingestion and query speed

Chapter 3

Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases

3.1 Introduction

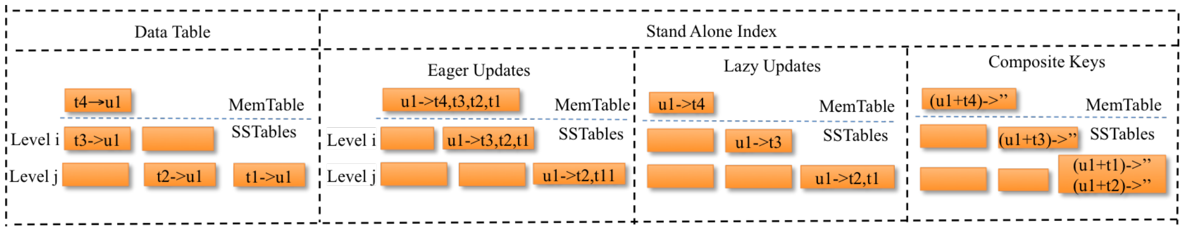
In the age of big data, more and more services are required to ingest high volume, velocity and variety data, such as social networking data, smartphone apps usage data and click through data. NoSQL databases were developed as a more scalable and flexible alternative to relational databases. NoSQL databases, such as HBase [39], Cassandra [76], Voldemort [34], MongoDB [59], AsterixDB [7] and LevelDB [55] to name a few, have attracted huge attention from industry and research communities, and are widely used in products. Through the use of Log-Structured Merge-Tree (LSM) [84], NoSQL systems are particularly good at supporting two capabilities: (a) fast write throughput, and (b) fast lookups on the primary key of a data entry (See Appendix 2.1 for details on LSM storage framework). However, many applications also require queries on non-key attributes which is a functionality commonly supported in RDBMSs. For instance, if a tweet has attributes such as tweet id, user id and text, then it would be useful to be able to return all (or the most recent) tweets of a user. However, supporting secondary indexes in NoSQL databases is challenging, because secondary indexing structures must be maintained during writes, while also managing the consistency between secondary indexes and data tables. This significantly slows down writes, and thus hurts the system's capability to handle high write throughput which is one of the most important reasons why NoSQL databases are used.

Table 4.1 shows the operations that we want to support. The first three operations (GET, PUT and DEL) are already supported by existing NoSQL stores like LevelDB. Note that the secondary attributes and their values are stored inside the value of an entry, which may be in JSON format: $v = \{A_1 : val(A_1), \cdots, A_l : val(A_l)\}$, where $val(A_i)$ is the value for the secondary attribute A_i. For example, the key of a tweet entry could be $k = tweet\ id$, $A_1 = user\ id$ and $A_2 = body\ text$. Key k should not be confused with limit K (top-K), which means K most recent records in terms of insertion time in the database.

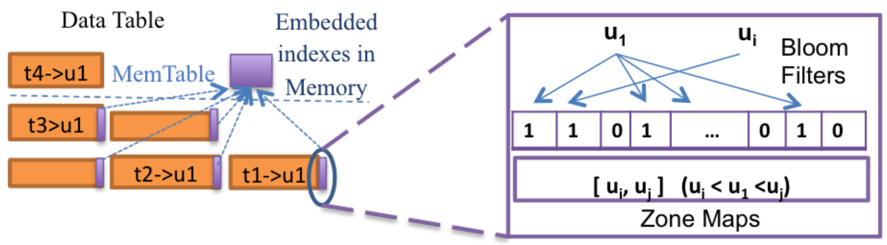
Stand-Alone Secondary Indexes Current NoSQL systems have adopted different strategies to support secondary indexes (e.g., on the user id of a tweet). Most of them maintain a separate Stand-Alone Index table. For instance, MongoDB [59] uses B+-tree for secondary index, which follows the same way of maintaining secondary indexes as traditional RDBMSs. They perform in-place updates (which we refer as "Eager" updates, and we refer to such indexes as Eager Indexes) of the B+ tree secondary index, that is, for each write in the data table the index (e.g., B+ tree) is updated.

Table 3.1: Set of operations in a NoSQL database.

Operation	**Description**
GET (k)	Retrieve value identified by primary key k.
PUT (k, v)	Write a new entry $\langle k, v \rangle$ (or overwrite if primary key k exists).
DEL (k)	Delete the entry identified by primary key k if any.
LOOKUP (A, a, K)	Retrieve the K most recent entries with $val(A) = a$.
RANGELOOKUP (A, a, b, K)	Retrieve the K most recent entries with $a \leq val(A) \leq b$.



(a) Comparison between Eager and Lazy updates, and <secondary+primary> Composite keys in Stand-Alone Indexes



(b) Zone maps and bloom filters in Embedded Index

Figure 3.1: Comparison between various secondary indexes after operations in Example 3.1. We use the notation $key \rightarrow value$.

In the case of some LSM-based NoSQL systems (e.g., BigTable [21]), which store a secondary index as an LSM table (column family), an Eager update is technically not in-place, but a new posting list is written that invalidates the older ones. In contrast to in-place updates, other LSM-based systems (e.g., Cassandra [76]) perform append-only updates on the LSM index table, which we refer as "Lazy" updates, and Lazy Indexes.

Example. Consider the current state of a database right after the following sequence of operations *PUT(t1, {u1, text1}),PUT(t2, {u1, text2}),PUT(t3, {u1, text3},$\cdots$)*, where *ti* is a tweet id, *ui* is a user id, and $texti$ is the text of a tweet. Then, as shown in Figure 3.1(a), to execute *PUT(t4, {u1, text4})* on an Eager Index, we must retrieve the list for $u1$, add $t4$ and save it back, whereas for a Lazy Index, we simply issue a *PUT(u1, {t4})* on the user id index table without retrieving the existing posting list for $u1$. The old postings list of u1 is merged with *(u1, {t4})* later, during the periodic compaction phase. ■

A drawback of the lazy update strategy is that reads on index tables become slower, because they have to merge the posting lists at query time. Earlier versions of Cassandra handled this by first accessing the index table to retrieve the existing posting list of u1, then writing back a merged posting list to the index table. Then the old posting list becomes obsolete. However, this Eager Index degrades the write performance.

In addition to these Stand-Alone indexes which maintain posting lists, several systems (e.g. AsterixDB [7], Spanner [27]) adopt a different approach to maintain the secondary indexes, which we refer as Composite Index. Here, each entry in the secondary indexes is a composite key consisting of (secondary key + primary key). The secondary lookup is a prefix search on secondary key, which can be implemented using regular range search on the index table. Here, writes and compactions are faster than "Lazy," but secondary attribute lookup may be slower as it needs to perform a range scan on the index table. We have implemented Eager, Lazy and Composite Stand-Alone Indexes. All these indexes can naturally support both lookup and range queries.

Embedded Secondary Indexes We also built *Embedded indexes*, which differ in that there is no separate secondary index structure, but secondary attribute information is stored inside the original (primary) data blocks. We built two Embedded Indexes. The first is based on *bloom filters* [15], which are a popular hashing technique for set membership check (See Appendix 2.3 for details on bloom filters). Bloom filters are already being used for primary indexing in many systems such as BigTable[21], LevelDB[55] and AsterxiDB[7]. But surprisingly, to our best knowledge, it has not been used for secondary attribute lookup in any of the existing NoSQL system. A drawback of bloom filter index is that it can only support lookup queries.

In addition to bloom filters, we also implement another Embedded Index, *zone maps*, which typically store the minimum and maximum values of a column (attribute) in a table per disk block. Zone maps are used in Oracle [61], Netezza [57], and AsterixDB [7]. Zone maps can be used for both lookup and range queries. However, in practice, zone maps are only useful when the incoming data is *time-correlated*, otherwise most of the data blocks have to be examined [7]. An attribute is called time-correlated if its value for a record is highly correlated with the record's insertion timestamp. For example, tweet-id is time-correlated because the tweet-id value increases with time. Note that the higher the correlation the stronger pruning we achieve in zone maps. No index's or algorithm's correctness requires that an attribute is time-correlated.

Note that these Embedded Indexes do not create any specialized index structure, but instead attach a memory-resident bloom filter signature and a zone map to each data block for each indexed secondary attribute. As LSM files on disks (called SSTables, see Appendix 2.2) are immutable, these filters do not need to be updated; they are naturally computed when an SSTable is created. *In our experiments, we consider both bloom filters and zone maps together as one index,* *which we refer as Embedded Index.*

Example 1 (cont'd) *Figure 3.1 shows the differences between the five indexing strategies. As in LSM-style storage structures, data are organized as levels, three levels shown in Figure 3.1, where the top row is in-memory and the bottom two on disk. Lower levels are generally larger and store older data. In some systems like LevelDB, lower levels have more SSTables of the same size, and in some like AsterixDB, lower levels have just one but larger SSTable. Each orange rectangle in Figure 3.1 represents a data file (SSTable).*

Table 3.2 shows the various secondary indexing techniques used by existing NoSQL systems, and by our LevelDB++. As we see, each system only implements one or two of these approaches, which does not allow a fair comparison among them, which is a key contribution of this paper. More details on how different systems adopt these techniques are described in Section 4.2.

Summary of Methodology To compare different indexing techniques, we implemented all of them on top of LevelDB [55] framework, which to date does not offer secondary indexing support. We chose LevelDB over other candidates, such as RocksDB [52], because it is a single-threaded pure single-node key value store, so we can easily isolate and explain the performance differences of the various indexing methods. We compare these algorithms using multiple metrics including size, throughput, number of random disk accesses and scalability. We ran experiments using several static workloads (i.e. build the index first and perform reads later) and dynamic (mix of reads, writes, updates) workloads, using real and synthetic datasets. We measured the response time for queries on non-primary key attributes, for varying top-K and selectivity.

Table 3.2: Different Secondary Indexing techniques supported in different NoSQL databases

NoSQL Storage Systems	**Embedded Index**		**Stand Alone Index**		
	Zone Map Index	**Bloom Filter Index**	**Lazy Index**	**Eager Index**	**Composite Index**
LevelDB++	✓	✓	✓	✓	✓
LevelDB [55], RocksDB [52]					
Cassandra [76]			✓		
AsterixDB [7], Spanner [27]	✓				✓
MongoDB [59], CouchDB [3], DynamoDB [50], Riak [13], Aerospike [47]				✓	
Oracle [61], Netezza [57]	✓				

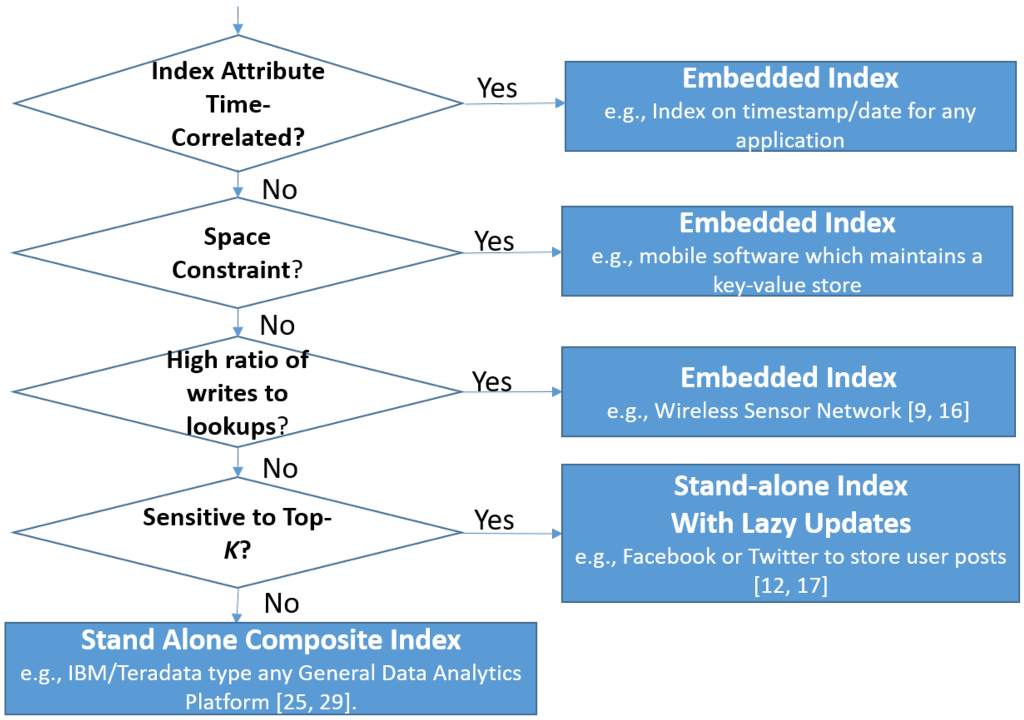

Index Attribute Time-Correlated?
Yes
Embedded Index
e.g., Index on timestamp/date for any application
No
Space Constraint?
Yes
Embedded Index
e.g., mobile software which maintains a key-value store
No
High ratio of writes to lookups?
Yes
Embedded Index
e.g., Wireless Sensor Network [9, 16]
No
Sensitive to Top-K?
Yes
Stand-alone Index With Lazy Updates
e.g., Facebook or Twitter to store user posts [12, 17]
No
Stand Alone Composite Index
e.g., IBM/Teradata type any General Data Analytics Platform [25, 29].

Figure 3.2: Proposed secondary index selection strategy.

Summary of Results As shown in Figure 3.2, the Embedded Index has lower maintenance cost in terms of time and space compared to Stand-Alone Indexes, but Stand-Alone Indexes have faster query (LOOKUP and RANGELOOKUP) times. Embedded Index is a better choice in the cases when index attribute is time-correlated (zone maps perform well) or when space is a concern, (e.g., to create a local key-value store on a mobile device) or if the query workload contains relatively small ($< 5\%$) ratio of LOOKUP/RANGELOOKUP queries over GETs and is heavy on writes ($> 50\%$ of all operations). An example of such an application is wireless sensor networks where a sensor generates data of the form (measurement id, temperature, humidity) and needs support for secondary attribute queries [43, 28]. In terms of implementation, the Embedded Index is also easier to be adopted, as the original basic operations (GET, PUT and DEL) on the primary key remain unchanged. The Embedded Index also simplifies the consistency management during secondary lookups, as we do not have to synchronize separate data structures. Among the Stand-Alone Indexes, Lazy Index achieves overall better performance for smaller top-K LOOKUPs and RANGELOOKUPs. For instances, it is reported that there are much more reads than writes in Facebook and Twitter [4, 1], and hence an ideal index to store user posts which is sensitive to top-K, will be Lazy Index. Eager Index shows exponential write costs and is not suitable for any workloads. Composite Index outperforms Lazy Index when there is no limit on top-K. For example, Composite Index is a good solution for general analytics platforms (e.g. IBM [56], TeraData [64]) where one may group by year or department and so on.

Summary of Contributions We make the following contributions:

- We perform an extensive study on the state-of-the-art secondary indexing techniques for NoSQL databases and present a taxonomy of NoSQL secondary.
- We implement two Embedded Indexes (Bloom filters and Zone maps) on LSM-based stores to support secondary lookups and range queries, and theoretically analyze their cost.
- We implement Lazy, Eager and Composite Stand-Alone Indexes on LSM-based stores and optimize each index for a fair comparison. We also theoretically analyze their cost.
- We conduct extensive experiments on LevelDB++ to study the trade-offs between different indexing techniques on various workloads (Section 4.7).
- We published the LevelDB++ code base as open-source, which includes our secondary indexes implementation on top of LevelDB, along with a Twitter-based data generator [75].

The rest of the paper is organized as follows. Section 4.2 reviews the related work. Sections 3.3 and 3.4 describe the indexing techniques and their theoretical cost models. Section 4.7 presents the experimental evaluation and we conclude in Section 4.8.

3.2 Related Work

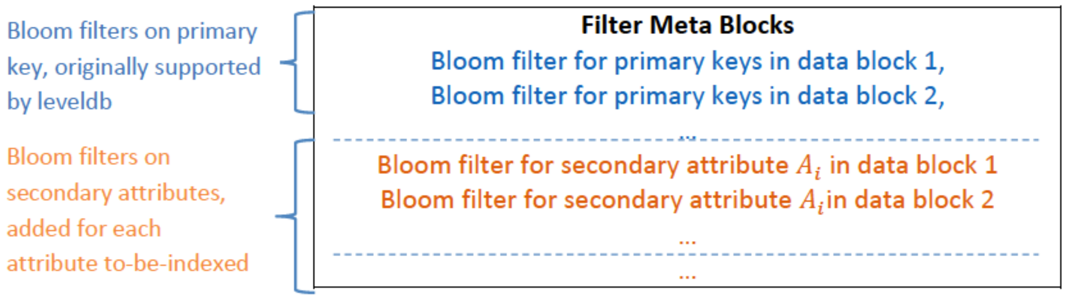
The idea of having the secondary index organized as a table has been studied in the past (e.g. Yahoo PNUTS [6]). Cassandra [76] supports secondary indexes by populating a separate table (column family in Cassandra's terminology) to store the secondary index for each indexed attribute. Their index update closely follows our Lazy Index approach. AsterixDB [7] introduces a framework to convert an in-place update index like B+ tree and R-tree to an LSM-style index, in order to efficiently handle high throughput workloads. Their secondary indexes closely follow our Composite Index strategy. This can be viewed as complementary to our work as we work directly on LSM-friendly indexes (set of keys with flat postings lists). AsterixDB also adopts zone maps to prune disk components (i.e. SSTables) for secondary lookups. But their zone maps are limited as they are only min-max range filters of whole SSTable files whereas our zone maps also maintain filters for all blocks inside an SSTable. Secondary indexes on HBase are discussed in [91], although that paper's focus is on consistency levels. Document-oriented NoSQL databases like CouchDB [3] and MongoDB [59], similarly to RDBMs, employ B+-trees to support secondary indexes. They mostly perform in-place update on the secondary index similarly to our Eager Index. A variation of Lazy Index is implemented in DualDB [87], although it is used as primary index and customized for pub/sub features only. HyperDex [32] proposes a distributed key-value store supporting search on secondary attributes. They partition the data across the cluster by taking the secondary attribute values into consideration. Each attribute is viewed as a dimension (e.g., a tweet may have two dimension tweet id and user id), and each server takes charge of a “subspace” of entries (e.g., any tweet with tweet id in [tid1, tid2] and user id in [uid1, uid2]). Innesto [82] applies a similar idea of partitioning and adds ACID properties in a distributed key-value store. DynamoDB [50], a distributed NoSQL database service, supports both global and local secondary indexes with a separate index table similar to Stand-Alone Indexing. Riak [13], similarly supports secondary indexing by means of Stand-Alone Index table for each partition, and performs an Eager update after each write operation. In this paper, our focus is on a single-machine storage engine for NoSQL databases. The distribution techniques of HyperDex, DynamoDB, Riak and Innesto can be viewed as complementary if we want to move to a distributed setting.

3.3 Embedded Index

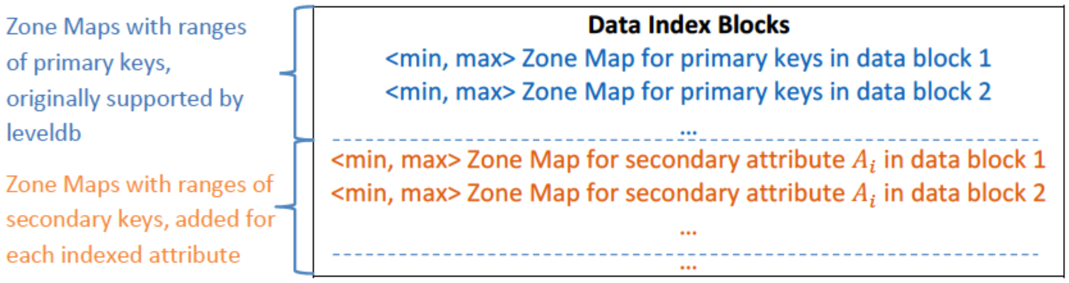
Overview As shown in Figure 3.1(b), in each SSTable file we constructed a block with a set of bloom filters on the secondary attributes and appended the bloom filter block to the end of the SSTable. Specifically, a bloom filter bit-string was stored for each secondary attribute for each block inside the SSTable file (recall that an SSTable file is partitioned into blocks, see Appendix 2.1 and 2.2 for a detailed overview of the LSM storage model and LevelDB storage architecture). In addition, we added zone map indexes for accelerating range lookup on secondary keys. These zone maps store the minimum and maximum range of secondary keys for each data block. We used these zone maps to further accelerate point lookup queries. We generally compute these bloom filters and zone maps for each block when a new SSTable is created (i.e. an LSM memory component MemTable is flushed to disk or a compaction creates new SSTables). Note that these bloom filters and zone maps are loaded in memory (the same approach is used currently with the bloom filters of the primary key in systems like Cassandra and LevelDB). Hence, the scan over the disk files is converted to a scan over in-memory bloom filters/zone maps. We only access the disk blocks, which the bloom filter returns as positive. We refer to these indexes as *Embedded Index*, as no separate data files are created. Instead, the bloom filters and zone maps are embedded inside the main data files.

LevelDB, as other NoSQL databases like Cassandra, already employs bloom filters and zone maps to accelerate reads on primary keys. Several types of meta blocks are already supported in LevelDB (see Appendix 2.2). We modify the filter meta block and the data index block to add secondary attribute bloom filters and zone maps, as shown in Figure 3.3. For lookup in the MemTable, we maintain an in-memory B-tree on the secondary attribute(s). We also store one zone map for each SSTable file, in a global metadata file, allowing us to access the min-max range of secondary keys in the whole file, to avoid searching in block-level zone maps.

LOOKUP(A_i, a, K) We scan one level at a time until K results have been found. A key challenge is that within the same level there may be several entries with the queried secondary attribute value $val(A_i) = a$, and these entries are not stored as ordered by time (but by primary key). Fortunately, LevelDB, as other NoSQL systems, assigns an auto-increment sequence number to each entry at insertion, which we use to perform time ordering within a level. Hence, we must always scan until the end of a level before termination. During this scan, we use the bloom filter and zone map indexes to check each data block in the SSTables. If both index checks return true, we read a block from disk. To efficiently compute the top-K entries, we maintain a min-heap ordered by the sequence number. If we find a match (i.e. we find a record that has secondary attribute value $val(A_i) = a$),


Bloom filters on primary key, originally supported by leveldb
Bloom filters on secondary attributes, added for each attribute to-be-indexed
Filter Meta Blocks
Bloom filter for primary keys in data block 1,
Bloom filter for primary keys in data block 2,
...
Bloom filter for secondary attribute A_i in data block 1
Bloom filter for secondary attribute A_i in data block 2
...
...

(a) Bloom filters


Zone Maps with ranges of primary keys, originally supported by leveldb
Zone Maps with ranges of secondary keys, added for each indexed attribute
Data Index Blocks
<min, max> Zone Map for primary keys in data block 1
<min, max> Zone Map for primary keys in data block 2
...
<min, max> Zone Map for secondary attribute A_i in data block 1
<min, max> Zone Map for secondary attribute A_i in data block 2
...
...

(b) Zone maps

Figure 3.3: Modified LevelDB SSTable for Embedded Index.

then if the heap size is equal to K, we first check whether it is an older record than the root of the min-heap. If it is a newer match or the heap size is less than K, then we check whether it is a valid record (i.e. whether it is invalidated by a new record in the data table). We check this by issuing a special function *GetLite(k, currentlevel)* (Algorithm 10 in Appendix), which is a variation of the *GET* operation. Here we know the *currentlevel* (i.e. the level where the key is placed) of the actual record. GETLite checks the in-memory metadata, index block and bloom filters for primary keys to check which level contains the key. If the key appears in the upper levels (0 to $currentlevel - 1$), that means there is an updated version of the key that is present in the database and this record is considered invalid. Here, we do not need to perform disk I/O, which a regular *GET* operation would do. This simple optimization in Embedded Index significantly reduces disk I/O.

RANGELOOKUP(A_i, a,b, K) We support RANGELOOKUP in Embedded Index via zone maps. Similar to LOOKUPs, here we check SSTables level by level. First, we check the global zone map of an SSTable to check if we can avoid scanning this file. Then we check the zone maps of each block of the SSTable to find overlaps with the query range $[a,b]$ and find out which blocks may contain records with the secondary attribute value a. A min-heap is similarly used here to maintain top-K and we stop scanning after reaching to the end of one level if K matches are found. The pseudocode of LOOKUP (Algorithm 10) and RANGELOOKUP (Algorithm 13) on Embedded Index is presented in Appendix A.

GET, PUT and DEL A key advantage of the Embedded Index is that only small changes are required for all three interfaces: GET(k), PUT(k,v), and DEL(k). Obviously, GET(k) needs no change as it is a read operation and we have only added bloom filter and zone map index to the data files. As SSTables are immutable, PUT and DEL just updates the in-memory B-tree for the MemTable data.

3.3.1 Cost Analysis

Cost of GET/PUT/DEL queries The Embedded Index does not incur much overhead on these operations. MemTable does not have bloom filters or zone maps, so there is no extra cost in maintaining them for each operation. It only adds small in-memory overhead on the compaction process to build bloom filters and zone maps for each indexed attribute per block and small overhead for updating the B-tree for MemTable.

Cost of LOOKUP and RANGELOOKUP queries Let the number of blocks in level-0 be b (and thus level-i has approximately $b \cdot 10^i$ blocks). According to Equation 2.1, the expected minimal false positive rate of the bloom filter is $2^{-\frac{m}{S} ln2}$, denoted as fp (for more details on bloom filters, see Appendix 2.3). Thus the approximate number of block accesses on level-i for LOOKUP due to false positives is $fp \cdot b \cdot 10^i$. Hence, if a LOOKUP query needs to search on L levels, the expected block accesses are $\sum_{i=0}^{L} \left(fp \cdot b \cdot 10^i\right) = \frac{fp \cdot b \cdot \left(10^{L+1}-1\right)}{9}$. Let us assume that matched entries are found in $K + \varepsilon$ blocks, then the total number of block accesses is $K + \varepsilon + \frac{fp \cdot b \cdot \left(10^{L+1}-1\right)}{9}$. Here ε represents the extra cost of searching to the end of a level to find top-K from matched entries. Note that, the CPU cost of computing membership check using in-memory bloom filters cannot be neglected here. For example, if the database size is 100GB and block size is 4KB, that means there exist 25 million bloom filters for one secondary indexes and a LOOKUP query may need to check all of them in the worst case. The CPU cost is generally much smaller for an index on a time-correlated attribute, because file-level zone maps are very effective and prune most files.

For RANGELOOKUP operation, the false positive rate depends on the distribution of the secondary key ranges in different blocks. In the worst case for non time-correlated index, RANGELOOKUP may have to traverse all blocks in the database, same as if there is no index. For time-correlated index, the worst case cost is $K + \varepsilon$ block accesses.

The worst-case number of disk accesses for different operations in Embedded Index is presented in Table 3.3. The Embedded Index will achieve better LOOKUP query performance on low cardinality data (i.e., an attribute has small number of unique attribute values), because fewer levels must be accessed to retrieve K matches.

Table 3.3: Disk accesses for operations in Embedded Index. (** means that CPU cost may not be ignored)

Operation	**Read I/O**	**Write I/O**
GET(k)	1	0
DEL(k), PUT(k,v)	0	1
LOOKUP	$(K+\varepsilon)+\frac{fp\cdot b\cdot(10^{L+1}-1)}{9}$ **	0
RANGELOOKUP	$(K+\varepsilon)$ (Time-correlated index attribute), Same as no index (non time-correlated)	0

3.4 Stand-Alone Index

3.4.1 Stand-Alone Index with Posting Lists

A Stand-Alone secondary index in a NoSQL store can be logically created as follows. For each indexed attribute A_i, we build an index table T_i, which stores the mapping from each attribute value to a list of primary keys, similarly to an inverted index in Information Retrieval. That is, for each key-value pair $\langle k, v\rangle$ in the data table with $val(A_i)$ not null, k is added to the postings list of $val(A_i)$ in T_i. Posting lists can be serialized as a single JSON array. Example 3.4.1 presents an illustration for this.

Example. Assume the current state of a database right after the following sequence of operations *PUT(t1, {u1, text1}),...,PUT(t2, {u1, text2}),...,PUT(t3, {u2, text3}), PUT(t4, {u2, text4})*. Tables 3.4a and 3.4b show the logic presentation of the state of primary data table(UserIndex) and secondary Stand-Alone Index(TweetData) in a key-value store after these four operations. ■

Compared to the Embedded Index, Stand-Alone Index has overall better LOOKUP performance, because it can get all candidate primary keys with one read in the index table in Eager Index

Table 3.4: Stand-Alone UserIndex with Posting lists of tweets

(a) TweetsData - Primary table

Key	**Value**
t1	{“UserID”: u1, “text”: “t1 text”}
t2	{“UserID”: u1, “text”: “t2 text”}
t3	{“UserID”: u2, “text”: “t3 text”}
t4	{“UserID”: u2, “text”: “t4 text”}

(b) UserIndex

Key	**Value**
u1	[t2, t1]
u2	[t4, t3]

(or up to L reads when there are L levels in the case of Lazy Index as we discuss below). However, Stand-Alone Index is more costly due to the maintenance of index table upon writes (PUT(k, v) and DEL(k)) in the data table in order to keep the consistency between them. We present two options to maintain the consistency between data and index tables, Eager updates and Lazy updates, discussed in Sections 3.4.1 and 3.4.1 respectively. We have implemented both options on LevelDB.

Eager Index

PUT($k, \{A_i : a_i\}$)/DEL(k): Upon a PUT, a Stand-Alone Index with Eager updates (i.e. Eager Index) first reads the current postings list of a_i from the index table, adds k to the list and writes back the updated list. *This means that, in contrast to the Lazy Index, only the highest-level posting list of a_i needs to be retrieved, as all the lower ones are obsolete.*

Example. Suppose PUT(t3, {”UserID": u1, “Text”: “t1 text”}) is issued after operations in example 3.4.1, which updates the UserID attribute of t3 from u2 to u1. ■

Figure 3.4 depicts a possible initial and final instance of the LSM tree for Eager Index after PUT operation in Example 3.4.1. The DEL(k) operation in Eager Index follows the same read-update-write process to update the list. Note that the compaction on the index tables works the same

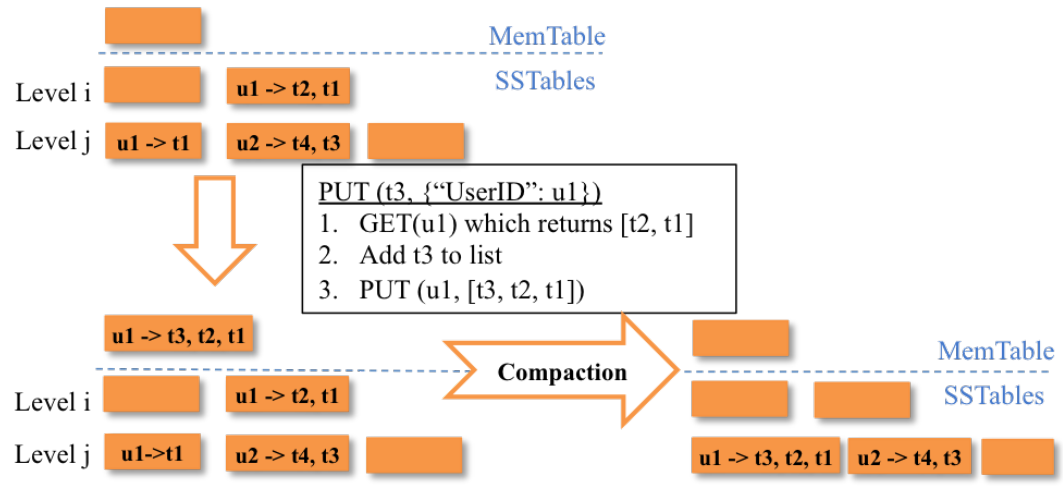


Figure 3.4: Stand-Alone Eager Index and its compaction.

as the compaction on the data tables.

LOOKUP(A_i, a, K): A key advantage of Eager Index is that LOOKUP only needs to read the postings list of a in the highest level in T_i. To support top-K LOOKUPs, we maintain the postings lists ordered by time (sequence number) and then only read a K prefix of them. For each entry k in the list of primary keys, we issue a GET(k) on data table to retrieve the data record. We make sure $val(A_i) = a$ for each entry to check the validity as there could be invalid keys in the postings list of a caused by updates on the data table (i.e. Insertion of an existing key with a different secondary key). Pseudocode of LOOKUP (Algorithm 7) on Eager Index is presented in Appendix A.

RANGELOOKUP(A_i, a, b, K): To support range query (RANGELOOKUP) on secondary attribute in Eager Index, we use range query API in LevelDB on primary key. We issue this range query on our index table for given range $[a,b]$. For each match x, we retrieve K number of most recent primary keys from the posting list which is already sorted by time. To return a top-K among different secondary keys in range $[a,b]$, we need to add associated posting lists' primary keys to the min-heap to get the top-K. To sort mean-heap by timestamps, we attach a sequence number to each entry in the postings list on every write. This sequence number represents entry time for each entry in the posting lists.

Lazy Index

PUT($k, \{A_i : a_i\}$)/DEL(k): The Eager Index still requires a read operation to update the index table for each PUT, which significantly slows down PUTs. In contrast, the Lazy Index works as follows. Upon PUT (on the data table), it just issues a PUT($a_i, [k]$) to the index table T_i but nothing else. Thus, the postings list for a_i will be scattered in different levels. During merge compaction, we merge these fragmented lists. DEL operation similarly issues a PUT(${a_i}^{del}, [k]$), but maintains a deletion marker which is used during merge in compaction to remove the deleted entry. Figure 4.2 depicts the update of index table by the Lazy update strategy upon the PUT in Example 3.4.1.

LOOKUP(A_i, a, K): Since the postings list for a could be scattered in different levels, a query LOOKUP needs to merge them to get a complete list. For this, it checks the MemTable and then the SSTables, and moves down in the storage hierarchy one level at a time. Note that at most one postings list exists per level. Similar to Eager Index, we need to retrieve the data and check validity of a matching record by issuing a GET operation on primary table. Note that, as levels are sorted based on time in the LSM tree, if we already find top-K during a scan in one level, LOOKUP can stop there, and it does not require to go to next level.

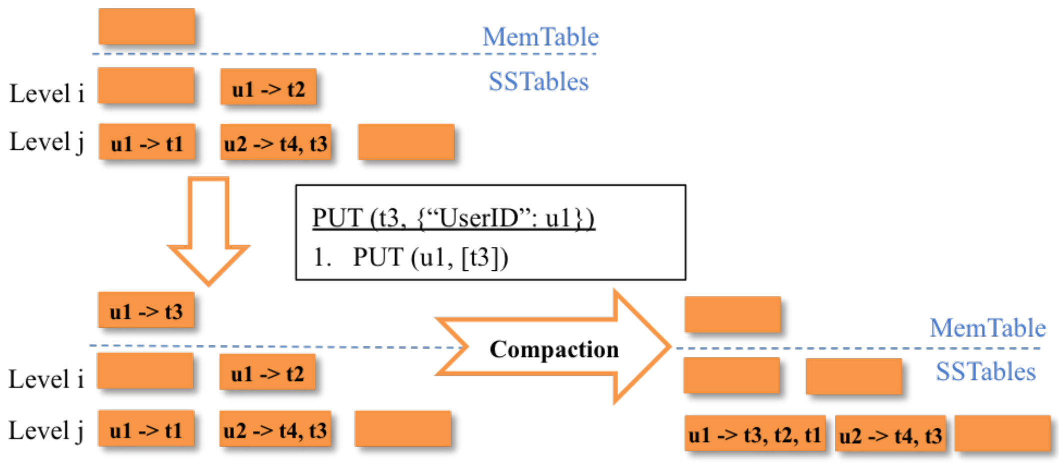


Figure 3.5: Stand-Alone Lazy Index and its compaction.

RANGELOOKUP(A_i, a, b, K): To support range query in Lazy Index, we modified the primary key range query API in LevelDB. The original range iterator iterates through a range and does not scan a key within the range in lower levels if it already exists in an upper level. We force the iterator to scan level by level (same as LOOKUP). For each level it will look for all the secondary keys for a given range [a,b]. As each key in that range can be fragmented into different levels, a range query needs to traverse each level separately. Similarly to Eager Index, for each secondary key, it adds the associated posting lists' primary keys to a min-heap, which is sorted based on the sequence number (which we include in each entry in the list). Pseudocodes of LOOKUP (Algorithm 8) and RANGELOOKUP (Algorithm 11) on Lazy Index are presented in Appendix A.

3.4.2 Stand-Alone Index with Composite Keys

PUT($k, \{A_i : a_i\}$)/DEL(k): Here, the composite key is the concatenation of the secondary and the primary keys, and the value is set to null. Figure 3.6 depicts the update of index table for Composite Index upon PUT in Example 3.4.1. Similarly to Lazy Index, a DEL operation inserts the

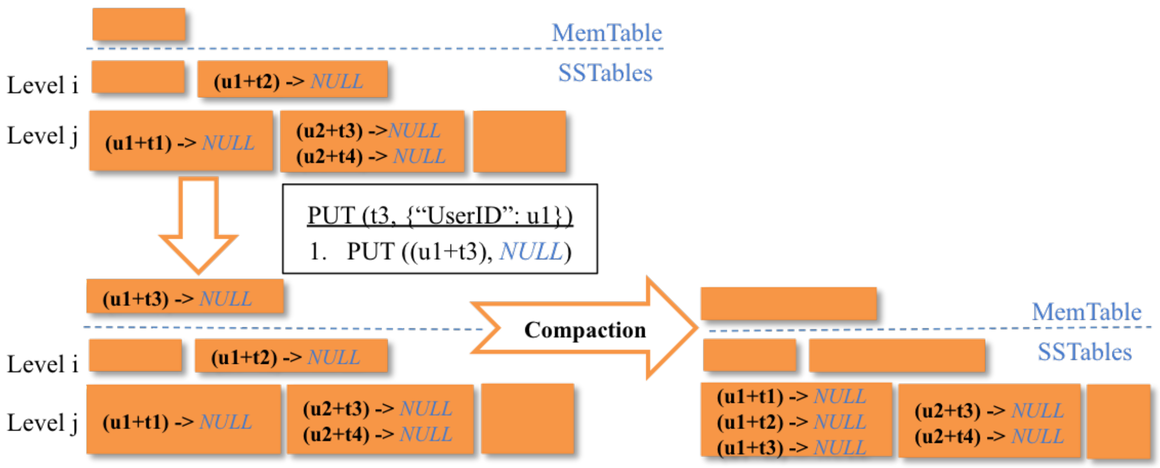


Figure 3.6: Stand-Alone Composite Index and its compaction.

composite key with a deletion marker in index table. During compaction, this marked entry is used to detect and remove the deleted entry.

LOOKUP(A_i, a, K): LOOKUP on secondary key performs a prefix range scan on Stand-Alone Index. This scan returns all primary keys where the secondary key is a prefix of the primary key in the Stand-Alone Index. At each iteration, the iterator breaks the composite key to perform the prefix check. Note that, unlike in Lazy Index, LOOKUP needs to traverse all levels to find top-*K* entries. In LevelDB, a compaction in a level takes place as round-robin basis, based on SSTable's primary key range in that level. We use composite keys as the primary key of index table. These composite keys associated with a secondary key in a level may span into multiple SSTable files where any one of them can participate in compaction and merges with next level. Hence, given a secondary LOOKUP key, we cannot conclude that these composite keys associated with that secondary key in different levels are sorted based on insertion time in primary table.

RANGELOOKUP(A_i, a, b, K): The RANGELOOKUP operation also uses that prefix based range lookup. Here instead of one prefix, it will traverse for all keys in where it's a prefix of an entry within the range [a,b]. Note that for each match in LOOKUP and RANGELOOKUP, Composite Index also performs a GET operation on the data table similarly to Lazy and Eager Index to retrieve the data record, and then checks its validity. The pseudocodes of LOOKUP (Algorithm 9) and RANGELOOKUP (Algorithm 12) on Composite Index are presented in Appendix A.

3.4.3 Cost Analysis

Cost of GET/PUT/DEL queries Stand-Alone Indexes do not incur any overhead on GET queries. However, for a PUT/DEL query on the data table, a Read-Update-Write process is issued on the index table by the Eager Index variant, in contrast to only one write on the index table in the Lazy and Composite Index variant. PUT/DEL overhead is linear to the number of secondary indexes in Stand-Alone Indexes.

However, as Stand-Alone Indexes maintain separate tables compared to Embedded Index, they incur additional compaction cost for each index table which heavily affects write performance. Compaction cost depends on write amplification factor $WAMF$ (i.e. the number of times the same record is written to the disk). Lazy and Composite compaction suffer same write amplification as a leveldb primary table, because they write a simple key value pair on every write to the index table. The cost of WAMF has been shown to be $2 \cdot (N+1)(L-1)$, where N is the ratio of the sizes of two consecutive levels [53, 30]. N is set as 10. Compared to Composite, Lazy Index has some extra CPU overheads (e.g. parsing and merging each json posting list during compaction, handling large lists which do not fit inside a block etc.) during compaction.

Eager Index suffers huge write amplification on index table as it updates the posting list for every write. Let us assume, average size of the posting list in Eager Index is PL_S. That means a record in the list has already been re-written PL_S times by the eager update strategy. Now $WAMF$

for Eager Index becomes $PL_S \cdot 22 \cdot (L-1)$, where $22 = 2 \cdot (10+1)$. Note that $WAMF$ for primary table is the same for all indexing techniques and therefore omitted from the analysis.

Cost of LOOKUP and RANGELOOKUP queries For Eager Index, only one read on the index table is needed to retrieve the postings list[1], as all lower level lists are invalid. In contrast, in Lazy Index, a read on the index table may involves up to L (number of levels) disk accesses (because in the worst case the postings list is scattered across all levels). Then, for each matched key in the top-K list, it issues a GET query on the data table to retrieve the actual record. Hence, if there exists K' matched entries for LOOKUP(A_i, a, K), it takes a total of $K'+1$ and $K'+L$ disk accesses for Eager and Lazy Index respectively. Note that we find our top-K entries from K' matched entries (i.e. $K' \geq K$). For the Composite Index, a read on the index table may involve up to L block accesses. Here we assume that result secondary keys fit in one block for Lazy/Composite Index. However, as discussed in Section 3.4.1 and 3.4.2, Lazy can stop after scanning just one level, if the top-K results are found, whereas Composite Index needs to traverse down to the lowest level to compute the top-K results. This is why Lazy Index performs better for smaller top-K LOOKUP queries than Composite Index. When there is no limit on top-K, both Lazy and Composite have same I/O cost of L, but Lazy has non-negligible CPU cost for maintaining posting lists as discussed earlier. For RANGELOOKUP query, if there exist M blocks in index table that contain a secondary key within the range, in the worst case all the variants need to access all M blocks. Here again for K' matched entries, Stand-Alone Indexes issue K' GET queries on data table. Table 3.5 shows the worst-case number of disk accesses for different operations in Stand-Alone Indexes.

Table 3.5: Disk accesses for operations in Stand-Alone Indexes. Assume l attributes are indexed on the primary table. (** means that CPU cost cannot be neglected, * means that average case is much better)

Operation	**Index Type**	**Data Table I/O**		**Index Table I/O**		
		Read	**Write**	**Read**	**Write**	**WAMF**
GET	All	1	0	0	0	0
PUT, DEL	Eager	0	1	l	l	$l \cdot PL_S \cdot 22 \cdot (L-1)$
	Lazy	0	1	0	l	$l \cdot 22 \cdot (L-1)$ **
	Composite	0	1	0	l	$l \cdot 22 \cdot (L-1)$
LOOKUP	Eager	K'	0	1	0	0
	Lazy	K'	0	$L*$	0	0
	Composite	K'	0	L	0	0
RANGE LOOKUP	All	K'	0	M	0	0

Table 3.6: Notation

top-K or K	Most recent K entries based on database insertion time.
(k,v)	key(k)-value(v) pair.
L	Number of levels in the store.
PL_S	Average length of secondary index posting lists.
K'	Number of matched entries for LOOKUP/RANGELOOKUP.
l	Number of secondary indexed attributes.
M	Number of blocks in Index Table with secondary key within given range of RANGELOOKUP query.
fp	Bloom filter false positive rate.

We summarize the notations used in the algorithms and cost analysis in Table 3.6.

[1]assuming there is no false positive by bloom filters on primary key in the index table

3.5 Experiments

All experiments are conducted on a machine powered by an eight-core Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz CPUs, 16GB memory, 3TB WDC (model WD3000F9YZ-0) hard drive, with CentOS release 6.9. All the algorithms are implemented in C++ and compiled by g++ 4.8. In our experiments, we set the size of the bloom filter to 100 (See Appendix 2.3), which we found to be a suitable number for our datasets, as discussed in Appendix B.1. We use the default compression strategy of LevelDB, Snappy [54], which compresses each block of an SSTable (see Appendix B.2 for experiments with uncompressed blocks). No block cache was used and the $max.open.files$ is set to large number (30000) so that most of the bloom filters and other metadata can reside in memory.

3.5.1 Workload

There are several public workloads for key-value store databases available online such as YCSB [26]. However, as far as we know there is no workload generator which allows fine-grained control of the ratio of queries on primary to secondary attributes. Thus, to evaluate our secondary indexing performance, we created a realistic Twitter-based operations (GET, PUT, LOOKUP, RANGELOOKUP) workload generator [75], which is a able to generate two types of workloads: *Static* and *Mixed*. The Static one first does all the insertions, builds the indexes and then performs queries on the static data. Static workloads can isolate GET, PUT, LOOKUP and RANGELOOKUP performance. In contrast, Mixed has continuous data arrivals, interleaved with queries on primary and secondary attributes simulating real workloads. Next, we show how to build a large synthetic tweets dataset based on a seed one. Then, we show how we generate Static and Mixed operation workloads, given a dataset of tweets.

Synthetic dataset generator Our dataset generator inputs a “seed” set of tweets, a set of secondary (to be indexed) attributes (e.g., UserID, time, place), and a scale parameter, and generates a synthetic dataset, which maintains the attribute value distributions in the seed set. In particular, for each generated tweet, we pick a UserID (or any other attribute, except time which has a special treatment as we discuss below) based on the distribution of UserID’s in the seed set, that is, users with more tweets in the seed set will be assigned to more synthetic tweets. Figure 3.7 shows the rank-frequency distribution of UserID attribute (i.e. number of tweets posted by a user) in our seed dataset. The number of tweets per second is selected based on a uniform distribution with minimum set to 0 and maximum equal to two times the average number of tweets per second in the seed set. We also generate a body (text) attribute with length picked randomly from the seed set, and random characters. The reason we create a body attribute, although it is not used as a secondary key, is to make the experiments more realistic, in terms of number of records that can fit in a primary table block.

Seed dataset We collected 18 million tweets with overall size 10 GB (in JSON format) through Twitter Streaming API, which have been posted and geotagged within New York State. This took about 3 weeks, which motivates why we build a synthetic tweet generator. The average number of tweets per user is 30 and the average number of tweets per second is 35. The average size of a tweet is 550 Bytes, each containing 22 attributes.

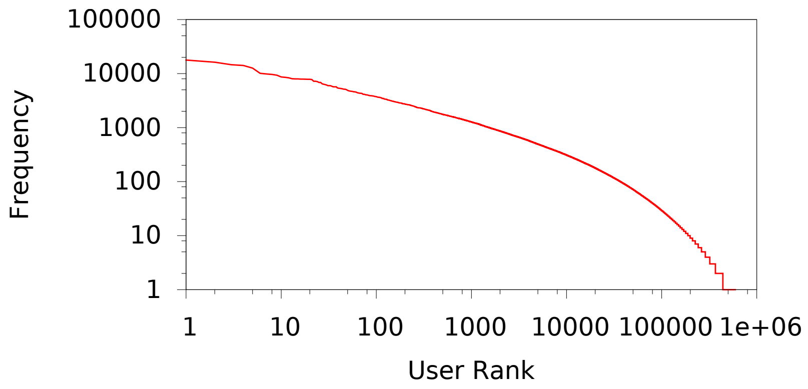


Figure 3.7: Distribution of UserID attribute in Seed Dataset.

Operations workload generator The *Static workload generator* inputs (a) a set of tweets, (b) a number n of GETs (or LOOKUPs or RANGELOOKUPs), and (c) the selectivity (for RANGELOOKUPs only), and generates n query operations of the desired type. All input tweets are first inserted (PUT) and then the query operations are executed. The *Mixed workload generator* inputs (a) a set of tweets, (b) a total number n of operations (PUTs, GETs, or LOOKUPs), (c) the frequency ratios between these three operations, and (d) the ratio of PUTs that insert an existing primary key (TweetID). We refer to the latter as "Updates" below. In both generators, the conditions of the query operations are selected based on the distribution of values in the input tweets dataset.

Workloads used in experiments In our experiments, we set the scale parameter in the dataset generator to 10 and generate a 100GB dataset with 180 million tweets using the workload generator. Using larger datasets would be challenging, given that the experiments took more than one month to run for the 100GB dataset, mainly due to the slow execution of the Eager Index. In the operations workload generator we selected UserID and CreationTime as two secondary attributes; CreationTime is time-correlated (expected to have a good performance for Zone Maps), but UserID is not. Table 3.7 summarizes different parameters and their values used to generate Static and Mixed workloads for our experiments.

Table 3.7: Parameters and values for different workloads.

Number of Op n				**Selectivity**		**Top-**K
PUT	**GET**	**LOOKUP**	**RANGE LOOKUP**	**UserID (no of users)**	**CreationTime (minutes)**	
180 million	100K	15K	50K	10, 100	1, 100	10, 100, No Limit

(a) Static Workload

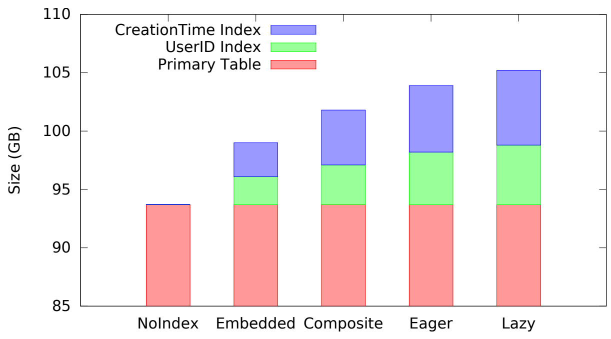
Workload Type	**Operation Frequency Ratios**				n	**Top-**K
	PUT	**GET**	**LOOKUP**	**Update**		
Write heavy	80%	15%	5%	0%	50 million	10
Read heavy	20%	70%	10%	0%		
Update heavy	40%	15%	5%	40%		

(b) Mixed Workloads

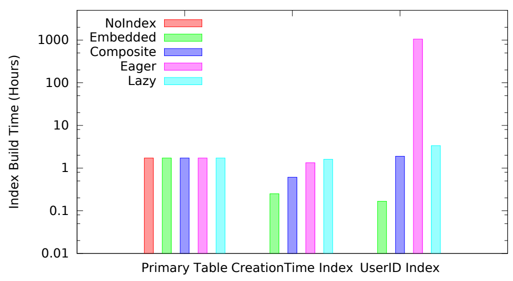
3.5.2 Experimental Evaluation

In this section we evaluate our secondary indexing techniques on LevelDB++.

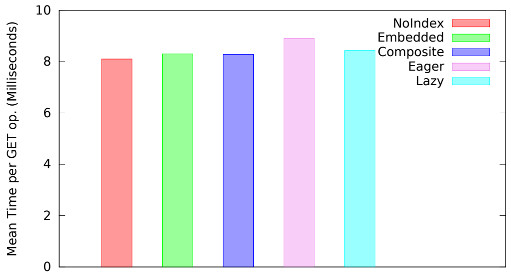
Results on Static workload



(a) Database size



(b) PUT Performance



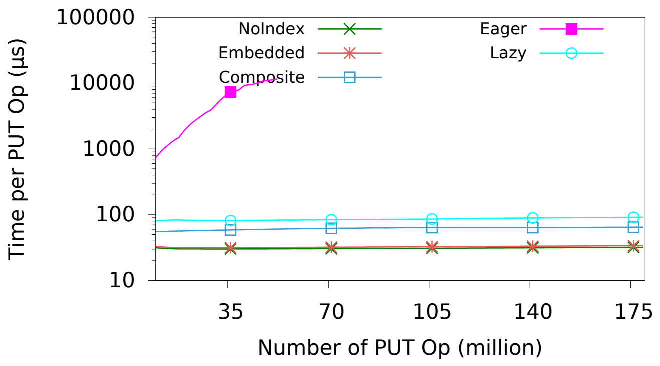
(c) GET Performance

Figure 3.8: Performance of different index variants for basic leveldb operations.

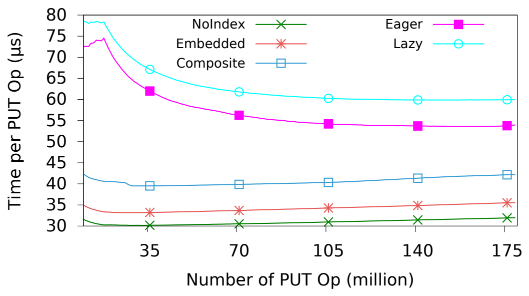
Overhead on Basic LevelDB Operations Figure 3.8 shows how the performance of basic leveldb operations is affected by the various secondary index implementations, in terms of size and mean operation execution time for our Static workload. Figure 3.8a shows that as Embedded Index does not maintain separate index table, it is more space efficient than Stand-Alone Indexes and close to having no index. Eager and Lazy Indexes have JSON overhead to maintain the posting lists, which makes them less space efficient than Composite Index. Lazy has fragmented posting list throughout different levels, whereas Eager Index has more compact list, hence Eager consumes less space than Lazy. Figure 3.8c shows that all the index variants have identical GET performance with negligible difference.

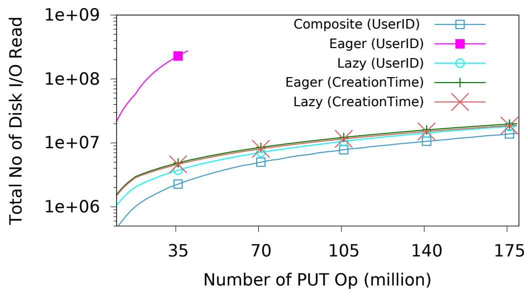
Figure 3.8b shows the PUT performance for different index variants. We isolate the creation time of primary index and two secondary indexes (CreationTime and UserID). That is, the CreationTime Index time shows the difference between the time of PUT when we only have one secondary index (on CreationTime) minus the PUT time when there is no secondary index. If we add the three times, we get the total PUT time for each index. We see that for both secondary indexes, Embedded Index outperforms the Stand-Alone indexes. Composite shows best PUT performance among Stand-Alone Indexes. Eager performs extremely bad for UserID index because of high write amplification and expensive merge compactions.

Figure 3.9 shows how PUT performance varies over time as the database grows. We took reading after each million insertions. PUT performance remains almost identical for both the attribute index as the database grows (Shown in Figure 3.9a and 3.9b) except the Eager Index. Figure 3.9c shows that merge compaction for Eager Index is exponentially costly for UserID index as the database grows. Eager Index shows good performance for the time-correlated CreationTime index, because the posting list is created sequentially and there is no random list update, hence there are fewer compactions.



(a) Mean PUT latency (UserID)





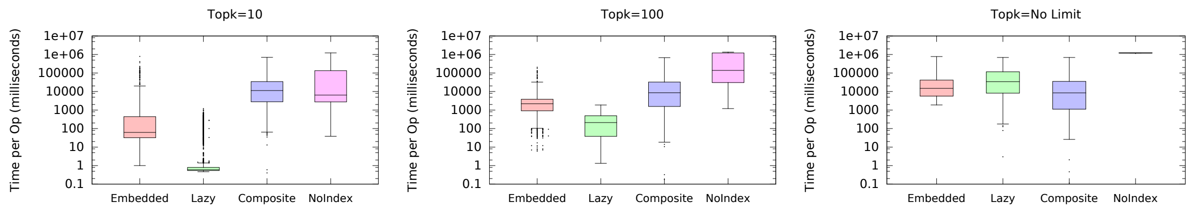
(b) Mean PUT latency (CreationTime)

(c) Cumulative I/O cost for index compaction

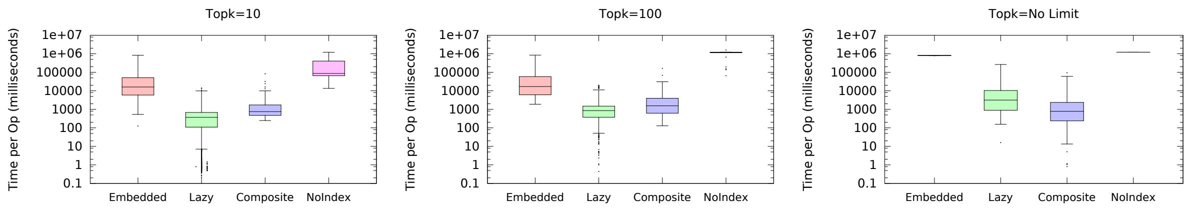
Figure 3.9: Write performance for different index variants over time.

Relationship to theoretical analysis Figures 3.8b and 3.9 support the analysis in Sections 3.3.1 and 3.4.3, which show that Embedded Index incurs negligible overhead on writes (I/O cost is 1), whereas the Stand Alone Indexes need to perform additional disk accesses. Specifically, for Eager Index it is 1+l+l = 5 (l=2), for Lazy and Composite it is 1+l=3. In addition, each Stand-Alone Indexes suffers high Write Amplification ($WAMF$) contributing to extra compaction cost. $PL_S = 30$ for UserID and $PL_S = 35$ for CreationTime index. We compute the $WAMF$ for each indexing techniques $WAMF_{Eager} = 30 \cdot 22(4-1) + 35 \cdot 22(4-1) = 4290$ ($L = 4$ in the index tables). $WAMF_{Composite} = WAMF_{Lazy} = 2 \cdot 22(4-1) = 132$. These numbers are supported by Figure 3.9c.

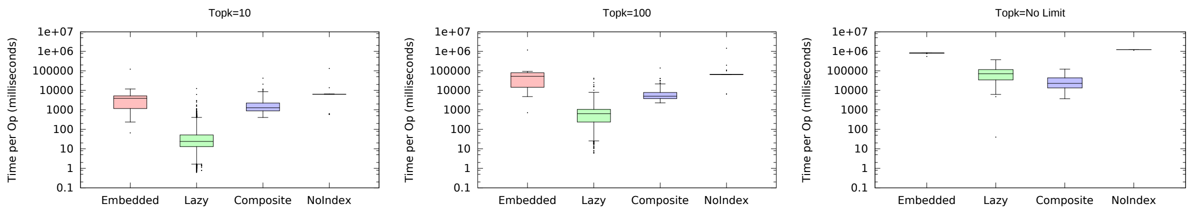
Section 3.4.3 explains and Figure 3.8b confirms that Lazy has higher CPU cost (** in Table 3.5) than Composite Index. Table 3.3 and 3.5 also supports Figure 3.8c, which shows that there is no overhead on GET (reads on primary index). The negligible difference (less than 1 *milliseconds*) between the GET of different approaches may be attributed to OS buffer cache or background compactions, due to the additional storage space required by the secondary indexes.



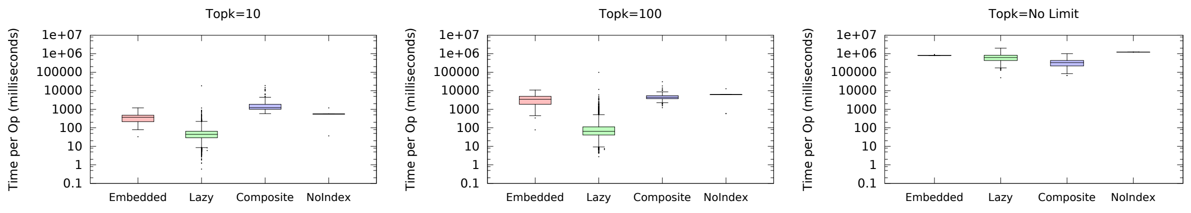
(a) LOOKUP Response Time



(b) RANGELOOKUP Response Time (Selectivity: 10)



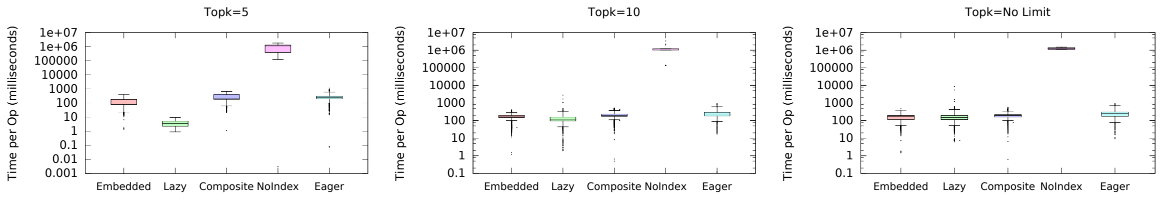
(c) RANGELOOKUP Response Time (Selectivity: 100)



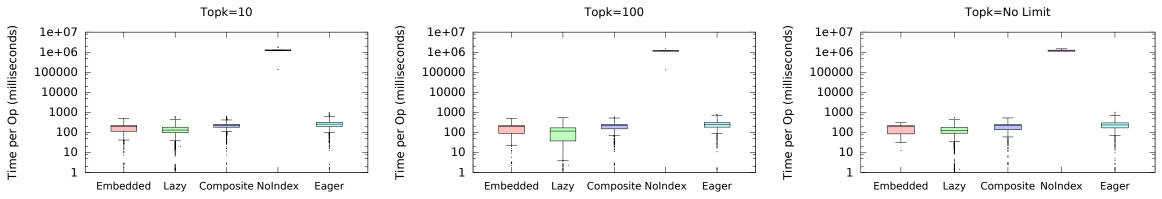
(d) RANGELOOKUP Response Time (Selectivity: 1000)

Figure 3.10: UserID Index performance for different variants varying Selectivity and TopK.

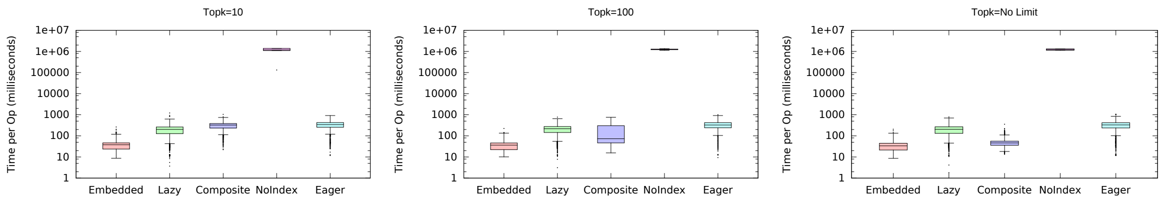
Query (LOOKUP, RANGELOOKUP) Response Time: Figure 3.10 shows the latency of LOOKUP (3.10a) and RANGELOOKUP (3.10b, 3.10d) with different selectivity and top-*K* of different index variants on non time-correlated index attribute UserID. Eager Index is excluded



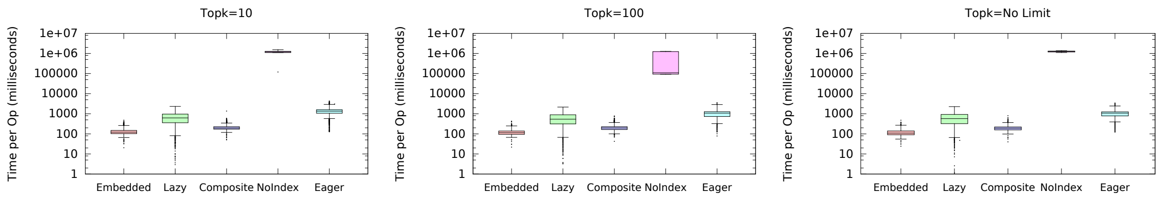
(a) LOOKUP Response Time



(b) RANGELOOKUP Response Time (Selectivity: 1 min)



(c) RangeLookup Response Time (Selectivity: 10 min)



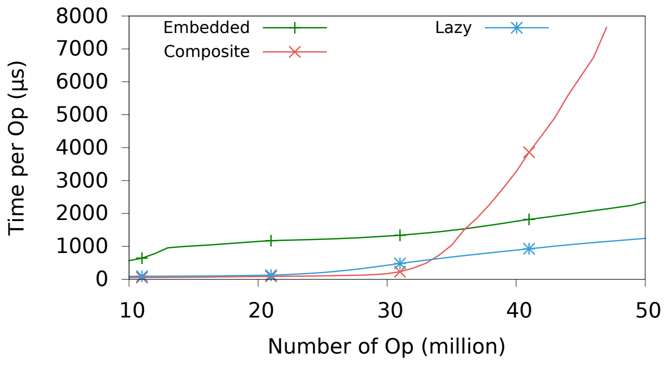
(d) RANGELOOKUP Response Time (Selectivity: 100 min)

Figure 3.11: CreationTime Index Performance for different variants varying TopK and selectivity.

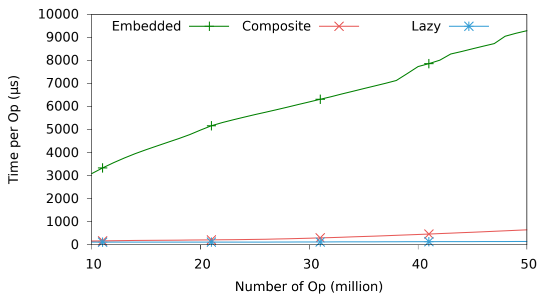
from these experiments because we already found out it is unusable for high write amplification and costly merge compaction. Here, latency for different queries are presented in box-and-whisker plots where quartile boundaries are determined such that 1/4 of the points have a value equal or less than the first quartile boundary, 1/2 of the points have a value equal or less than the second quartile (median) value, etc. A box is drawn around the region between the first and third quartiles, with a horizontal line at the median value. Whiskers extend from the ends of the box to the most distant point whose y value lies within 1.5 times the interquartile range. These figures show that Lazy Index performs best when top-*K* is small for LOOKUP queries. Composite Index performs best when top-*K* is bigger or there is no restriction of top-*K* (i.e. return all results). But when *K* is small Embedded Index also outperforms Composite Index. Stand-Alone Indexes outperform Embedded Index for RANGELOOKUP queries, as zone maps are not useful for non time-correlated attribute. Similar to LOOKUPs, Lazy performs best for RANGELOOKUP queries when top-*K* is smaller, otherwise Composite outperforms Lazy. Embedded Index performs better for higher selectivity and outperforms Composite Index if top-*K* is smaller as Embedded can find top-*K* results in top levels for higher selectivity queries. Figure 3.11 shows query response performance for time-correlated CreationTime index attribute. Similar to UserID index, Lazy outperforms all index for LOOKUP queries (Figure 3.11a). For RANGELOOKUP queries, Embedded Index outperforms Stand-Alone Indexes for all selectivity factors because its zone maps are very most effective on time-correlated index attributes (Figure 3.11b, 3.11d). As the selectivity increases, Composite outperforms Lazy and Eager Index.

Relationship to theoretical analysis The cost analysis in Tables 3.3 and 3.5 supports our experimental results in Figures 3.10a and 3.11a. These figures show that Lazy Index is sensitive to top-*K* for LOOKUP queries and performs better than Composite Index for smaller top-*K* queries on average, whereas Composite Index outperforms Lazy when there is no limit on top-*K*. Our experimental results show that the CPU cost (** in Table 3.3) dominates the Embedded Index LOOKUP performance and it performs worse than Stand-Alone Indexes for non-time-correlated UserID Index. However, for time-correlated index attribute CreationTime, Figure 3.11a supports our analysis that zone maps have strong pruning power for time-correlated indexes and Embedded Index performance is comparable to Stand-Alone Indexes. For RANGELOOKUP queries, the cost analysis in Table 3.3 supports the experimental results in Figures 3.10b and 3.10d, which show that Embedded Index (i.e. Zone Maps) does not perform well for non time-correlated Index and almost perform same as no index. However, Figures 3.11b and 3.11d show that zone maps are powerful for RANGELOOKUP queries on time-correlated index and their better pruning power makes the disk access cost close to K.

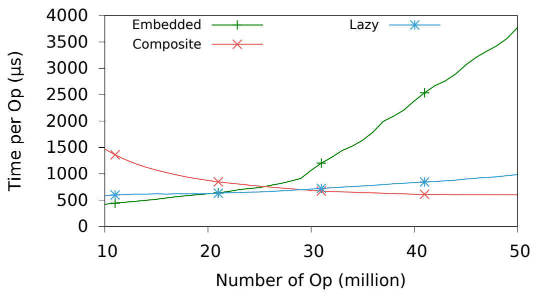
Results on Mixed workloads



(a) Write Heavy Workload

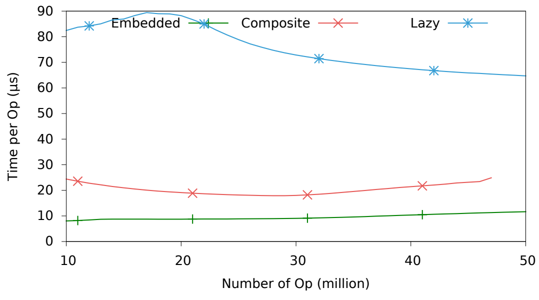


(b) Read Heavy Workload

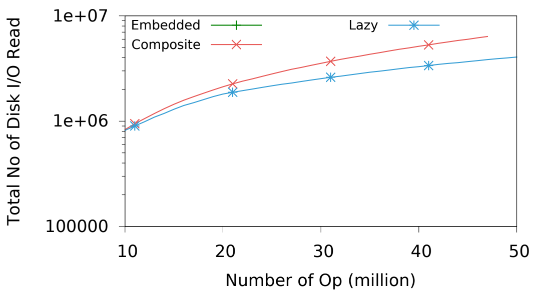


(c) Update Heavy Workload

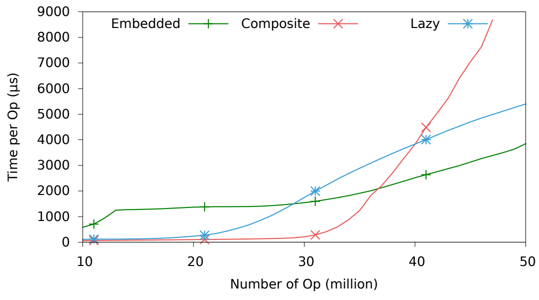
Figure 3.12: Overall performance comparison for Write, Read and Update Heavy workloads



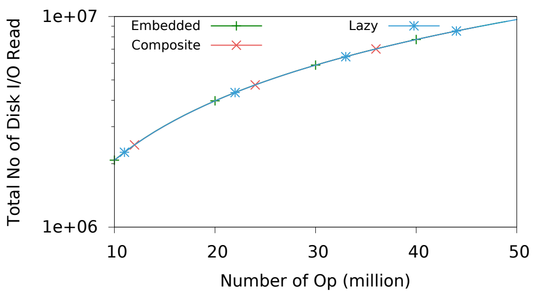
(a) Mean Time per PUT



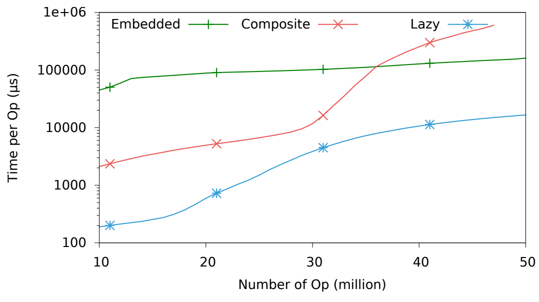
(b) Cumulative Disk I/O cost in Compaction



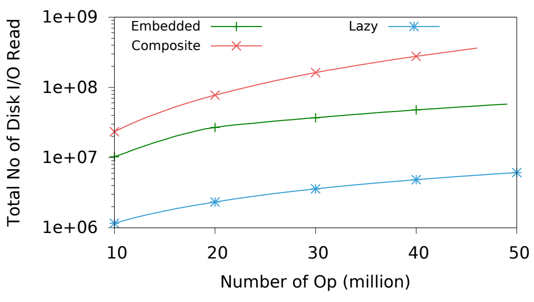
(c) Mean Time per GET



(d) Cumulative Disk I/O cost for GET



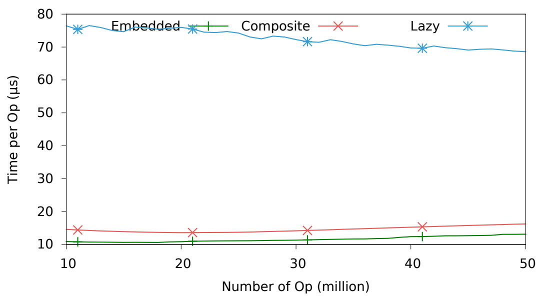
(e) Mean Time per LOOKUP



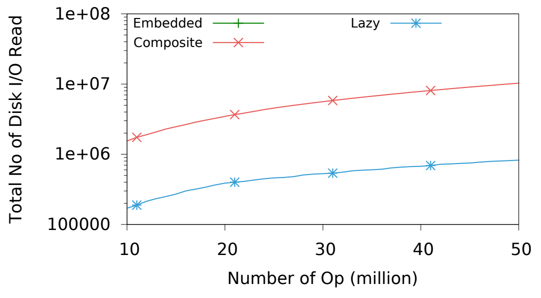
(f) Cumulative Disk I/O cost for LOOKUP

Figure 3.13: Performance in Write Heavy workload for different index variants

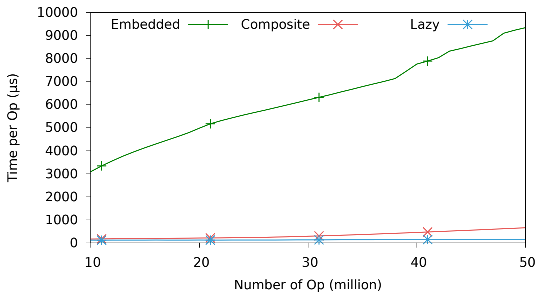
Figures 3.12a, 3.12b and 3.12c show the overall performance of index variants in terms of mean time per operation for different Mixed workloads (write, read and update heavy respectively). Only the UserID attribute is indexed and queried. We record the performance once per 1 million operations. We did not consider Eager Index as it is shown to be unusable in previous discussions. It



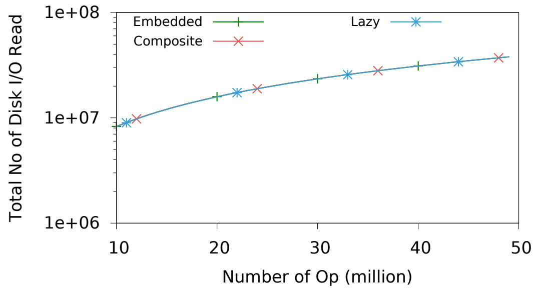
(a) Mean Time per PUT



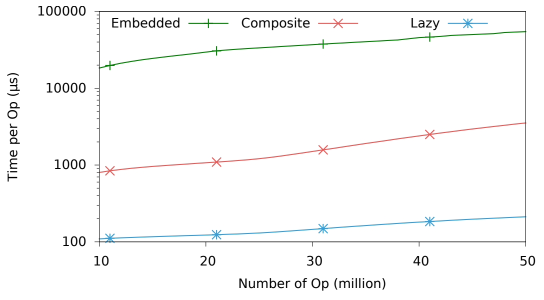
(b) Cumulative Disk I/O cost in Compaction



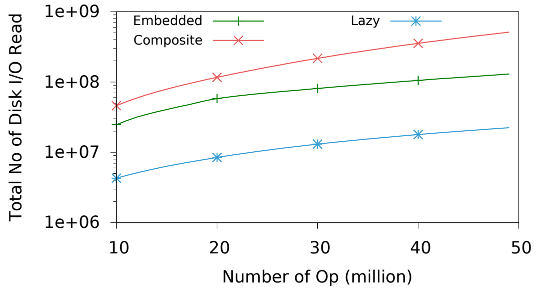
(c) Mean Time per GET



(d) Cumulative Disk I/O cost for GET



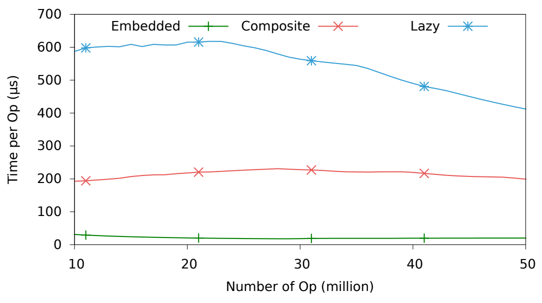
(e) Mean Time per LOOKUP



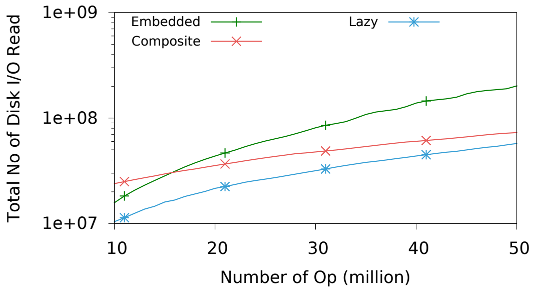
(f) Cumulative Disk I/O cost for LOOKUP

Figure 3.14: Performance in Read Heavy workload for different index variants

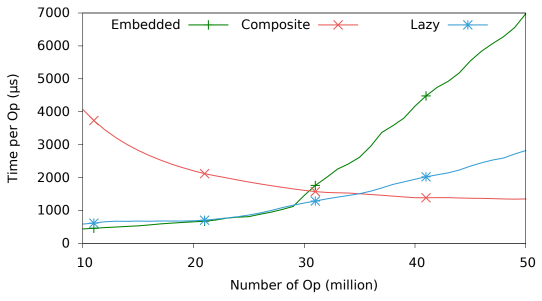
is difficult to isolate the performance of individual operations as one operation might incur overhead on the system and ultimately interfere with another operation. Also, the mean time per operation is not a good metric for performance measurement as we have seen in Figure 3.10. To properly measure the performance of individual operations, we record the cumulative number of disk I/O as



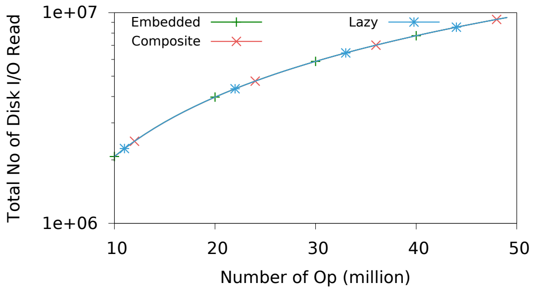
(a) Mean Time per PUT



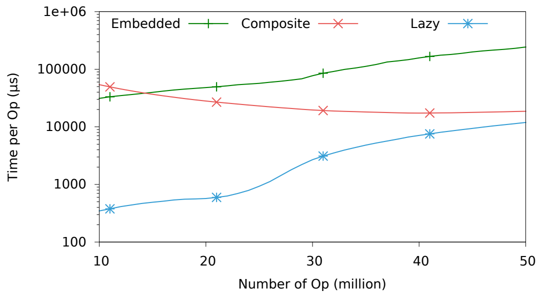
(b) Cumulative Disk I/O cost in Compaction



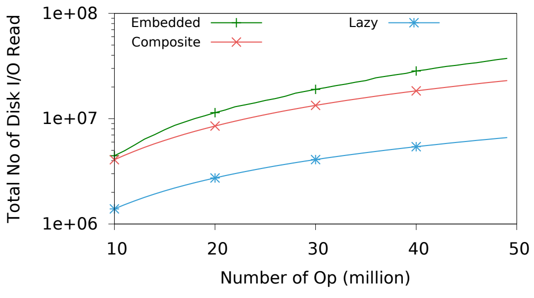
(c) Mean Time per GET



(d) Cumulative Disk I/O cost for GET



(e) Mean Time per LOOKUP



(f) Cumulative Disk I/O cost for LOOKUP

Figure 3.15: Performance in Update Heavy workload for different index variants

the database grows along with the mean time per operation. Figures 3.13, 3.14, and 3.15 represent the individual GET, PUT and LOOKUP performance of index variants in terms cumulative number of disk I/O for write, read and update heavy workloads respectively. We see that for different Mixed workloads, Lazy always perform better than the other indexes in terms of overall mean operation latency. Embedded Index does not perform well for read heavy workloads, as expected for non-time correlated index attribute UserID (Figure 3.12b). There is an unusual jump for Composite Index performance in write heavy workload (Figure 3.12a). The inflection point occurs at 30-35 million operations, which equals about 16GB of data which is the RAM size. At that point, the GET starts being very expensive as OS buffer cache becomes more ineffective. Beyond that point, the relationship between the performance of the various indexes stabilizes, which implies that our observations would likely also hold for bigger datasets. A Composite Index LOOKUP needs to perform many GET operations even for small top-K; these GETs dominate the overall performance. Composite Index shows slightly better performance than Lazy in the update-heavy workload (Figure 3.12c) as the database grows. The updates incur frequent compaction in Index Table, and compaction is heavier in Lazy than Composite Index because of the Json parsing overhead for maintaining lists. We also see that the Embedded Index starts to perform bad once the number of updates grows, as these updates result in extra rounds of compaction. We compare the compaction I/O costs for different workloads in Figures 3.13b, 3.14b and 3.15b. Embedded Index has higher cost only when there are updates on the primary table. Figures 3.13d, 3.14d and 3.15d show the same cost for GET operations as expected for all index variants. For LOOKUP (Figures 3.13f, 3.14f and 3.15f), Lazy always outperforms Composite and Embedded as the index is non-time-correlated and top-K is smaller. LOOKUPs in Embedded Index perform worse in the update-heavy workload. The reason behind unusual jumps in performance for both Embedded and Stand-Alone Indexes in Mixed workloads (Figure 3.12b) is because of an increase in OS block cache miss after a while. LSM-tree relies on frequent compaction operations to merge data into a sorted structure. After a compaction, the original data are reorganized and written to other locations on the disk. As a result, the cached data are invalidated, since their referencing addresses are changed, causing serious performance degradations. Because LOOKUP and RANGELOOKUP read lots of random disk blocks resulting bad cache performance eventually for any system. Various optimizations to overcome this problem are being studied and can be applied to our LevelDB++ [40] [80].

Summary of novel results We next summarize the results that have not been reported or adequately documented in previous work. First, Eager Index shows poor performance during ingestion and does not scale well with large datasets, even though it is currently widely used in several systems for its ease of implementation and fast query response time. Second, the combination of Embedded Indexes (i.e. zone maps and bloom filters) provides a lightweight indexing option, which is effective even for non time-correlated index attributes. However, no existing system, to our knowledge, employs them for secondary indexes. Third, Composite Index performs worse than Lazy for top-*K* LOOKUP queries, which are common in many modern applications (e.g. social networks, messaging apps etc.). However, due to the ease of implementation, Composite indexes are more widely used in such applications. Finally, the effect of buffer cache cannot be neglected during a real system workload, as shown in Figure 3.12.

3.6 Conclusions

We studied five secondary indexing methods for LSM-based NoSQL stores, namely Eager, Lazy and Composite (Stand-Alone Indexes), and Zone maps and Bloom filters (Embedded Indexes). We built a system, LevelDB++, which implements these indexing methods on top of LevelDB, and conducted extensive experiments to examine their performance. The experimental results show the trade-offs between these indexing techniques. We present guidelines to pick the right indexing techniques for different workloads and applications.

Chapter 4

High-throughput Publish/Subscribe on top of LSM-based Storage

4.1 Introduction

In this age of big data and Internet of Things (IoT), large amounts of data are generated, stored, and used by a diverse set of entities – devices, vehicles, buildings, software, and sensors. It is challenging to efficiently ingest, manage, read and deliver the generated data to millions of users or entities in real time. Publish/Subscribe systems are used in many applications, such as social networks, messaging systems, and traffic alerting systems.

As a running example application, consider users who are driving and subscribe to nearby traffic or other incidents (accident, crime, roadwork, fire, natural disaster, protest etc). Every time a user moves to a new location (i.e., a geospatial cell) they need to subscribe to events/publications in the new location for a time duration starting from the near past to the near future – e.g., to know what happened in the last one minute and what will happen in the next one minute until the user moves to another cell. These subscriptions are *highly dynamic* as they come and go every few seconds. At the same time, users are publishing incidents. As millions of users are moving and subscribing to events, these large streams of subscriptions and publications must be stored and managed efficiently. Google's spatial notifications closely follow this moving subscriber example. Here, when subscribers go to a store, Google pushes to them, as a mobile notification, the map of the store, specials deals, and so on. As another application, an airplane continuously queries for data in its path such as turbulence, wind, air pressure, etc.

Challenges and requirements Figure 4.1 shows different layers in a typical pub/sub system. Here

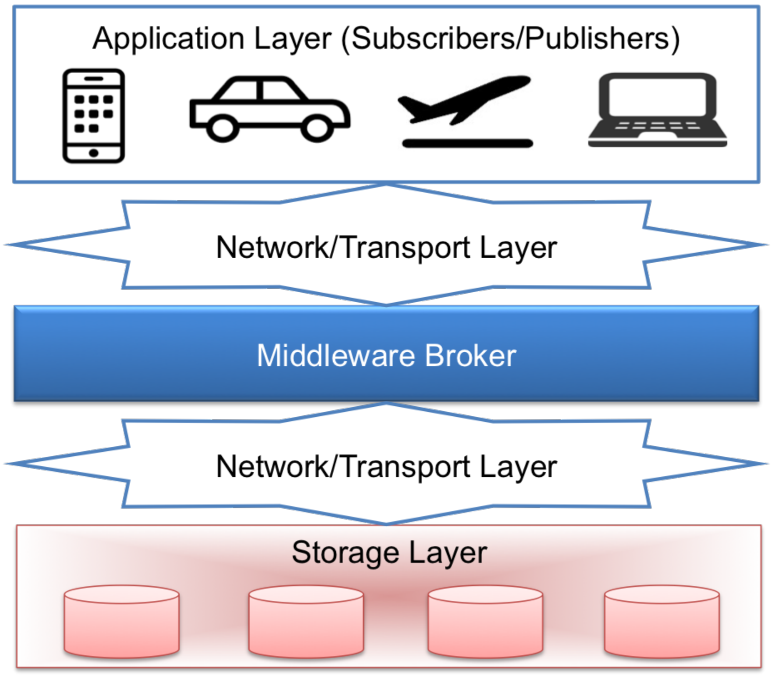


Figure 4.1: Different layers in a typical Pub/Sub system. This paper focuses on the red-highlighted storage layer

we can see that subscriptions and publications can be generated from a variety of devices in application layer. The application communicates through network/transport layer with a middleware broker which notifies subscribers for a matching publication. A key component of a Publish/Subscribe system is its storage layer, which stores both the subscriptions and the publications. The broker talks with the storage layer to store/retrieve these publications/subscriptions. Our paper focuses on the storage layer of a pub/sub system.

As described in detail in Section 4.2, the storage modules of existing Publish/Subscribe systems have several *limitations*, which make them inadequate for modern IoT applications. First, the number of subscriptions (continuous queries) is assumed to be relatively small to fit in main memory, which may not always be true. Second, the subscriptions are viewed as relatively static, for example, a user infrequently modifies her following list in Twitter. This is not the case in applications such as traffic alerting or aviation where vehicles or airplanes subscribe to different cells of interest every few seconds. Third, most of the systems support only queries on future data. In contrast, an airplane may need to know the conditions in an area in the last few minutes and the next few seconds, so a combination of past and future data may be desired. The publications may also have an expiration time; for example, a snow storm warning might have a certain time limit.

A *baseline* solution to realize a Publish/Subscribe system, which we experimentally evaluate, is to maintain a list of subscriptions (queries) and periodically submit them as *repetitive queries* on the publications database. A similar approach is currently used in the AsterixDB BAD system [17], where such subscriptions are referred as repetitive channels. A drawback with this approach is that one cannot get notification at the exact time of an incident. Also, this solution wastes resources as the same query will keep getting submitted even if no new matching publications exist.

Finally, some publications with short expiration time may be completely missed.

To summarize, our goal is to design and build a Publish/Subscribe storage system with the following properties:

1. Scale to millions of dynamically changing subscriptions and publications per minute, that is, both subscriptions and publications arrive and expire at a rapid pace.

2. After a new publication, immediately identify and notify matching subscribers (as in traditional database triggers, discussed in Section 4.2), that is, not follow a periodic check paradigm.

3. Subscriptions or publications may have validity time periods. Subscriptions may request past data in addition to future data.

4. The subscriber should assume that all the matching publications should reach her, that is, there is no data loss, which may occur with periodic check systems.

Contribution To support these properties we present an efficient storage framework on top of LSM-based NoSQL databases. We argue that NoSQL databases – such as Cassandra [76], Voldemort [34], AsterixDB [7], MongoDB [59] and LevelDB [55] – provide the right primitives –fast write throughput and fast lookups on the primary key – on top of which we can design an efficient Publish/Subscribe storage system that satisfies the above properties. In our paper, a database refers to a single LSM-based key-value store, which can be viewed as the equivalent to a table in a relational database. We built our prototypes on top of LevelDB, which is a popular and open-source LSM-based key-value store from Google. These LSM-based key-value stores as well as our databases contain both memory components(i.e. memtable) and disk components (i.e. SSTable).

We first present an approach based on two separate databases (*TwoDB*), one for subscriptions (queries) and one for publications (data records). The advantage of this approach is that it is simple to implement, as it only requires little or no change to current LSM storage modules. We show that despite its simplicity, it is efficient compared to a baseline based on repetitive queries (*RepQueries*) mentioned above. However, this two-database approach suffers from a high rate of random disk accesses when the subscriptions not only match publications but also past subscriptions – e.g., return publications that have not been already returned by a past subscription. To efficiently answer such complex subscriptions, which we refer as *self-joining subscriptions*, we propose a novel *DualDB* approach where both queries and data entries are stored in the same keyspace of the same database, which leads to much fewer random accesses.

We also propose techniques to extend these storage architectures to handle hierarchical attributes. For example, a publication may specify a large affected area (higher level bounding rectangle in a spatial hierarchy), whereas a subscription may only be interested in a smaller spatial cell. Similarly, a publication may specify a specific topic like "soccer", whereas a subscription may specify a more general topic like "sports."

In addition to the repetitive queries baseline, we compare our approaches to a state-of-the-art pub/sub system, Padres [35]. We chose Padres because it is open source and is easy to modify to fit our use-case. Padres supports querying on historic/past data, and uses the PostgreSQL database system. We show that Padres does not scale well with the number of subscriptions. Specifically, the system runs out of memory and crashes very quickly, because it uses an in-memory module to manage the subscriptions.

To summarize, we make the following contributions in this paper:

- We propose and implement two novel approaches built on top of on LSM-based storage systems, *TwoDB* and *DualDB*, to efficiently support high rates of dynamic subscriptions and publications (Section 4.5).

- We implemented our methods on LevelDB and conducted extensive experiments with various workloads with different subscription to publication ratios. For that, we extended LevelDB to handle lists of values per key (Section 4.4). Our experiments show that both approaches perform up to 1000% faster than baseline *RepQueries* (repetitive queries) and up to 3000% faster than a state-of-the-art pub/sub system Padres (Section 4.7).

- In addition to simple lookup subscriptions that return matching publications, we implemented methods to support subscription queries that join streaming publications/subscriptions with existing data/queries, which we refer as *self-joining subscriptions* (Section 4.6). We show that for such subscriptions *DualDB* performs better (up to 20%) compared to the *TwoDB* approach. Interestingly, *DualDB* performs 5% better than *TwoDB* even for simple match subscriptions (Section 4.7).

- To support hierarchical attributes, we propose a Dewey encoded representation of topics/cells and an efficient querying algorithm (Section 4.6). We show that our approach performs significantly better than reasonable baselines that handle multi-granular data (Section 4.7).

- We have published open-source version of our DualDB and TwoDB implementations on top of LevelDB [75].

The rest of the paper is organized as follows. Section 4.2 reviews the related work and background. Then, Section 4.3 presents the framework of the system, Finally, Section 4.8 concludes and presents future directions.

4.2 Related Work and Background

4.2.1 Related Work

Publish/Subscribe Systems

Most of the academic work on pub/sub mainly has studied how to efficiently route the message through the distributed pub/sub network. Instead, in this paper we propose an efficient data storage management system for pub/sub systems. There exist many variations of pub/sub systems supporting content or topic based subscription [33] [72]. Very few works studied the storage architecture of pub/sub systems. Padres [35] is a popular open-source pub/sub system, which allow continuous lookup queries. It supports subscriptions on future and past data, using a PostgreSQL database inside each broker [67]. Pub/Sub systems integrated in relational database systems have also been studied [14]. However, they do not scale with the number of subscriptions as we show in our experiments.

AsterixDB BAD [66] supports complex continuous queries, where queries are executed repetitively on top of the LSM-based AsterixDB database. Due to its repetitive nature, it does not provide instant notification. On the other hand, BAD supports a richer set of subscription query types than this work, where we require that subscriptions and publications match based on a key attribute condition. XML Based pub/sub systems also use a relational database for content based subscriptions [93]. Elaps [41] is a Location aware pub/sub system focusing on efficient processing of continuous moving range queries against dynamic event streams. Their work focuses on the query optimization rather than an efficient storage architecture. Our focus is on millions of dynamically arriving and expiring subscriptions and publications, which requires an efficient specialized storage framework. A preliminary version of our work was recently published as a short paper [88], which does not include the *TwoDB* approach, hierarchical attributes' support or self-joining subscriptions.

Triggers and Continuous Queries

Continuous queries on databases may be implemented using triggers [96] [89]. The TriggerMan project [42] proposes a method for implementing a scalable trigger system based on the assumption that some triggers may share a common structure. It uses a special selection predicate index and an in-memory trigger cache to achieve scalability. However, they are only able to handle a very small number of triggers on a table [23], whereas we want our continuous queries to scale to millions. Further, triggers have a relatively high creation and deletion cost, which makes them inappropriate for dynamically changing subscriptions.

NiagraCQ [23] proposed techniques to group continuous queries with similar structure, to share common computation. TelegraphCQ [20] is another system that handles streams of continuous queries over high-volume, highly-variable data streams. Complex continuous queries on data streams have also been studied [11] [81]. Kafka [38] is a popular stream processing platform. But our storage systems also support subscriptions on historic data where both subscriptions and publications dynamically expire, which data streams cannot support. Gemfire [65] is one of the few NoSQL database systems where continuous queries are supported. They maintain a separate in-memory processing framework, which maintains subscriber channels and communicates with the storage. InfluxDB [58] supports continuous queries on a time series database and thus its application is limited. Microsoft supports efficient spatial continuous queries on SQL Server [44]. However, these works assume that the queries fit in memory and are relatively static, that is, they do not scale to arbitrary numbers of queries nor to rapidly added and expiring queries. Further, these models generally only support "future-only" queries, that is, queries that only return future data items.

Join-Indices

Our work to combine two databases into one dual database is also related to join indices, such as Oracle's Bitmap join index [60], which creates a join index on attributes from two tables to facilitate faster joins. However, these are separate and specialized data structures that are generally expensive to maintain, in contrast to our proposed solution that does not add any redundant indexes and is efficient to maintain.

4.3 Framework

To support pub/sub operations on top of a NoSQL data store, we define a basic API as shown in Table 4.1. To illustrate the functionality of this API, consider the motivating application described in Section 4.1, where Subscriptions S (API call *SUBSCRIBE* (*ID*, *Subscription-JSON*)) are generated by mobile users driving in their vehicles, who are interested to know about recent events, such as accidents or weather changes, close to their current location L_S. "Recent" may refer to events that happened in the time interval $[T_{\min} = T_S - 10$ sec, $T_{\max} = T_S + 10$ sec$]$, where T_S is the query (subscription) time. Note that the time interval associated with each subscription S can include both past and future time ranges. If the intervals only contained future times, no storage would be needed for the publications. Whenever there is a new publication (API call *PUBLISH* (*ID*,

Table 4.1: Set of Operations for Pub/Sub Storage Framework

Operation	**Description**
SUBSCRIBE (*ID*, *Subscription-JSON*, *SelfJoinFlag*)	Store the subscription with *ID* as primary key; if T_{min} is in past ($T_S > T_{min}$), immediately return matching publications (publications with same *ID*), which are published in $[T_{min}, T_{max}]$ time interval ; if *SelfJoinFlag* is true, return matching list of subscriptions (see Section 4.6.1).
PUBLISH (*ID*, *Publication-JSON*, *SelfJoinFlag*)	store publication with *ID* as primary key; return subscribers who subscribed to this ID (i.e. topic/region); if *SelfJoinFlag* is true, also return matching publications .

Publication-JSON)), i.e. an event occurred, we look for all the stored subscriptions to notify them if the ID (cell-id for moving objects or topic-id for topic-based pub/sub systems) of the publication matches that of the subscription. Whenever there is a new subscription (API call *SUBSCRIBE* (*ID*, *Subscription-JSON*)), we store the subscription in the database so that for future publications it can find matching subscriptions. Here, if the SUBSCRIBE operation queries for "past" ($T_S > T_{min}$) publications, it should immediately return all matching and valid publications published during time interval in the past [T_{min} , T_S] (or [T_{min} , T_{max}] , if $T_{min} < T_S$). Note that, for both operations, it should only return "valid" publications/subscriptions. Valid means it should return only those matching results which have not expired at current operation execution time ($T < Tmax$). We emphasize that subscription time intervals span both the past and the future, we need a storage framework to

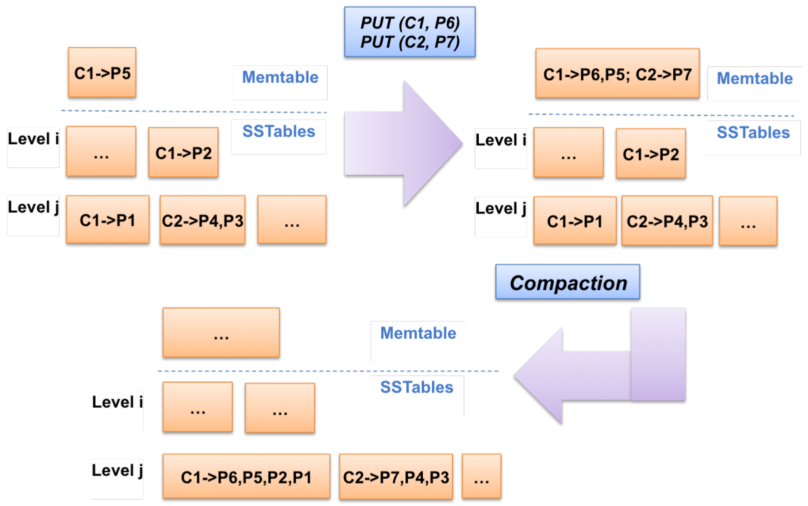


Figure 4.2: Example of handling multiple writes (PUTs) with the same key, and a subsequent compaction.

store the subscriptions as well as the publications.

ID is the key that joins subscriptions and publications. *Subscription-JSON* and *Publication-JSON* are the JSON-formatted values in our Key-Value store. They hold the attributes shown in Table 4.2. In this paper, we only consider topic-based pub/sub queries which match based on one attribute only (e.g. location, topic etc). Note that subscriptions may also specify additional conditions, such as keyword matching, for instance, return data close to me that contain the term "accident." Such conditions can be supported by a postprocessing (filtering) layer on top of the location-constrained or topic-constrained results. Another solution is to implement any of the existing secondary indexing techniques [86] on top of our proposed frameworks. These additional features are out of scope of this paper and we consider them as future works.

Table 4.2: Set of Attributes and Terminologies

Operation	**Description**
ID	cell-id/topic-id/product-id
T_P, T_S	Execution time of a publication and subscription
T_R	Duration between two repetitive queries.
T_{min}, T_{max}	The time interval in a subscription query. T_{max} is expiry time for subscription and publication.
Sid, Pid	Subscription and publication Identifier
Uid	User Identifier (One user can submit multiple subscriptions)
$Desc$	Text Description field in publication
Subscription-JSON	Body of a subscription in JSON $\{T_{min}, T_{max}, T_S, Sid, Uid\}$.
Publication-JSON	Body of a publication in JSON $\{T_P, T_{max}, Pid, Desc\}$

4.4 Enable Value Lists in LevelDB

In this section we discuss how we modify LevelDB to handle lists of values instead of a single value for every key. For example, the list of events for a single cell id is stored as a value list with the cell id as key. LevelDB holds the LSM property shown in Figure 2.1 where data is first inserted in memtable which is flushed to disk components called SSTables later. SSTables like in Figure 2.2 hold records with unique key. We described how SSTables are organized into levels in LevelDB in Section 2.2. STables organized into levels in LevelDB and because of the way the compaction takes place, there can be one unique key in each level.

For our problem, we do not want uniqueness in primary key as we assume that for both *SUBSCRIBE* and *PUBLISH* operation, the location or cell-id is the primary key and we will match queries with events using this key. So, instead of one record associated with a key we will have a posting list of queries or events. So for each insertion with non-unique primary key/cell-id, we add the new record with the current list of records instead of dropping the old record. The naive way to do it will be an in-place update of the list by issuing a read on the current list, appending the new data to the list and writing back to the storage. Background compaction later will discard the old list. But here the main drawback is that each write becomes very expensive, as we need to read a large list. High write throughput is one of the fundamental features of NoSQL databases which we can not compromise. Also we may have a lot of invalid large lists throughout all the levels in SSTables, which will waste a lot space.

To solve this problem we implemented a lazy update strategy on the postings list. When a new Write/Put is issued on a record, we do not look for existing value lists associated with the same key/cell-id in the disk. We simply write them to the memtable. If there is a existing list in the memtable, we perform in-place update on the list in constant time. Memtable is flushed to SSTables when it is full and these SSTables are compacted later to move to lower levels. We modify this compaction strategy appropriately, where instead of discarding old records, we merge lists associated with same key and hence eventually large lists are created in lower levels. Original compaction ensures that there can be at most one value list associated with a key in each level except level-0 files as SSTables in non-zero levels contain non-overlapping keys in LevelDB. Now we have fragmented list associated with a key throughout different levels. When we issue a read on key, instead of returning the record on upper level, it continues to search level by level to collect the fragmented lists.

Example 4.4 illustrates how we maintain value lists in a lazy manner for each key in our key-valuelist storage.

Example. Assume the current state of a Publication database right after the following sequence of publish operations *PUT(C1,P1), ..., PUT(C1,P2), ..., PUT(C2,P3), PUT(C2,P4),..., PUT(C1,P5).* Here *Ci* is the key and *Pi* is the value. Figure 4.2 depicts the state of the storage components (i.e. SSTables and Memtable) after these operations. Note that, the figure only highlights the records affected by these operations, but other records also exist in different components other than C1 and C2. We can see instead of key-value pairs we have value lists for keys C1 and C2, fragmented in components of different levels. Now, two new operations PUT(C1,P6), PUT(C2,P7) are issued. Figure 4.2 shows how it affects the current lists and how these lists are lazily updated during compaction. We first see the P6 is added to the list of C1 with an in-place update and a new record C2->P7 was created inside Memtable. Now after some compaction, these lists are moved to lower levels and we can see they are compacted and merged into larger lists (C1->P6,P5,P2,P1 and C2->P7,P4,P3). Here a read on C1 before the compaction would require to access the Memtable and SSTables in level i through level j. After compaction, a read requires to access an SSTable on level j only. ■

4.4.1 Cleaning up Invalid Records in Compaction

Our problem space explores to the big active data area where millions of subscriptions and publications can arrive and expire at a rapid pace. As they may have an attribute in their value (T_{max}), which denotes their expiration time. The storage management system should efficiently remove the expired items from its storage. Existing LevelDB storage does not support dynamic removal of expiry pub/sub records. Our proposed storage is built on top of LSM-tree which performs compaction in background. During compaction, we pick couple of SSTable files and compact them into a new set of files. During this time we sort and merge the list of publications or subscriptions associated with their ID, and check their expiration time against the current time to discard them if found invalid. This dynamic removal of expiry records during compaction does not allow LSM tree to grow indefinitely which makes our operations lot faster.

4.5 Proposed Approaches

We propose and implement several approaches to efficiently realize the API defined in Section 4.3, while providing instant response times to publications. We first present a reasonable baseline approach, *RepQueries*, which performs queries in a repetitive way and cannot provide instant responses. Next, we present our first novel instant-response approach (*TwoDB*), which maintains two databases for holding subscriptions and data records/publications. Our second approach (*DualDB*) maintains a single database holding both datasets. For all of the approaches we need to modify the standard key-value storage model to support lists of values per key (for details see Section 4.4).

4.5.1 Repetitive Queries Baseline (*RepQueries*)

RepQueries relies on a repetitive query approach and hence cannot guarantee instance delivery of a new publication. This approach is used in many pub-sub systems that use a broker management as a middle-ware between the database and the users. Here the user/broker is responsible for submitting the query repetitively to the publication database to find out the matching events/topics. Note that the user/broker is responsible to detect duplicate data if consecutive time ranges overlap. Also as the events are highly dynamic, some of them might expire in-between the repetitive queries and user can potentially face missing data.

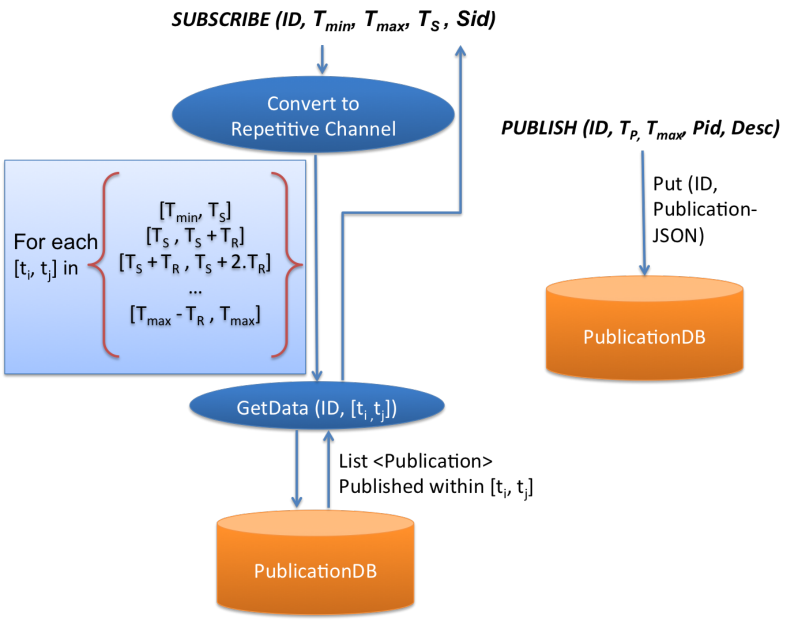
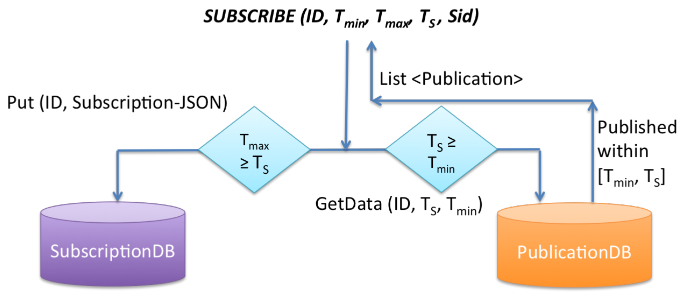

SUBSCRIBE (ID, T_min, T_max, T_S, Sid)
Convert to Repetitive Channel
For each [t_i, t_j] in
[T_min, T_S]
[T_S, T_S + T_R]
[T_S + T_R, T_S + 2.T_R]
...
[T_max - T_R, T_max]
GetData (ID, [t_i, t_j])
List <Publication> Published within [t_i, t_j]
PublicationDB
PUBLISH (ID, T_P, T_max, Pid, Desc)
Put (ID, Publication-JSON)
PublicationDB

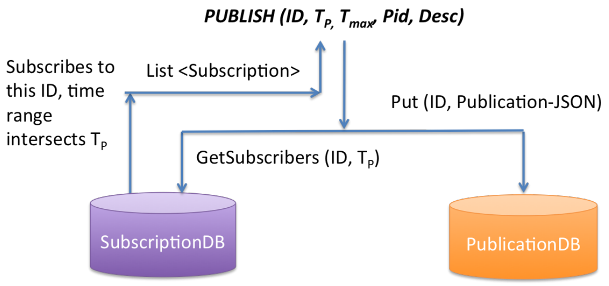
Figure 4.3: **RepQueries Approach**: Processing of *SUBSCRIBE* and *PUBLISH* Operation

This approach only requires a database of publications. As periodic queries are issued from client application, and there is no need for instant response, we do not need to store the queries for future publications. A subscription is converted to a sequence of queries, one every T_R time units as shown in Figure 4.3. Specifically, it first issues a historic query on past data with time interval $[T_{min}, T_S]$, and then it repetitively issues $GetData$ operation after every T_R time units. Here the PUBLISH operation does not issue any read, it only writes the new publication to the database. So PUBLISH will be very fast and SUBSCRIBE will be slow.

4.5.2 Two-Database Approach (TwoDB)


SUBSCRIBE (ID, T_min, T_max, T_S, Sid)
List <Publication>
Put (ID, Subscription-JSON)
T_max ≥ T_S
T_S ≥ T_min
Published within [T_min, T_S]
GetData (ID, T_S, T_min)
SubscriptionDB
PublicationDB

(a) Processing of new *SUBSCRIBE* Operation.


PUBLISH (ID, T_P, T_max, Pid, Desc)
Subscribes to this ID, time range intersects T_P
List <Subscription>
Put (ID, Publication-JSON)
GetSubscribers (ID, T_P)
SubscriptionDB
PublicationDB

(b) Processing of new *PUBLISH* Operation.

Figure 4.4: TwoDB Approach

As discussed in Section 4.2, previous works assume that the subscriptions are stored in memory, which is not realistic in our scenario. Figures 4.4a and 4.4b show the data flow for a new subscription (*SUBSCRIBE*) and a new publication of events/topics (*PUBLISH*), respectively, for a two-database algorithm, where separate databases (column families in Cassandra's terminology) are maintained for, respectively, the queries and the published events/topics.

For every new continuous query, we must, in parallel, insert to the subscription database *SubscriptionDB* and query the data storage *PublicationDB* for matching events/topics. The former is needed when the $T_{\max}$ of the subscription is greater than current query time T_S while the latter is only needed if the $T_{\min}$ of the query is less than T_S. The query to the publication database $GetData\,(cell-id, T_S, T_{min})$ returns the list of events which is published within time interval $[T_{min}, T_S]$ and are valid (i.e.did not expire) on that time. The procedure of *GetData* operation is presented in Algorithm 1. Since the postings list could be scattered in different levels, *GetData* needs to merge them to get a complete list. For this, it checks the Memtable and then the SSTables, and moves down in the storage hierarchy one level at a time.

For each new publication of events/topics, we must, in parallel, check if an active subscription query matches it, and also insert it into the publication database. The matching function $GetSubscribers\,(ID, T_P)$ returns a list of subscribers who subscribed to some events/topics on same ID and the query is still valid and if the publication time T_P is within time interval $[T_{min}, T_{max}]$ of the query. The procedure of *GetSubscribers* operation presented in Algorithm 2 is similar to *GetData* operation moving down level by level in storage to look for matching subscriptions.

Pruning search in SSTables

To answer a subscription that has a T_{min} in the past, we take advantage of an important characteristic of level-based LSM systems. In LevelDB, each level is ordered by time and older records go into lower levels. In our storage, each level can contain at most one list associated with

Algorithm 1 GetData(ID, T_S, T_{min}) operation

```
1:  $ListPB \leftarrow \emptyset$
2:  {Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3:  for $j$ from 0 to $L$ do
4:      if $ID$ is not in $Cj$ then
5:          NEXT
6:      end if
7:      List of Publications $P \leftarrow$ the value of key $ID$ in $Cj$
8:      for $Publication$ in $P$ do
9:          $T \leftarrow Publication(T_P)$
10:         $T_{exp} \leftarrow Publication(T_{max})$
11:         if $T < T_S$ and $T > T_{min}$ and $T < T_{exp}$ then
12:             ListPB.add($Publication$)
13:         end if
14:         if $T < T_{min}$ then $ListPB$
15:         end if
16:     end for
17: end for$ListPB$
```

a key. That means the fragmented lists associated with a key in different levels from top to bottom are ordered by time. Here, the publications in each lists are also ordered by execution time T_P. So for each SUBSCRIBE operation, when we perform *GetData* operation to look for all matching publications from past (T_{min}), we check for associated lists going level by level. For each matching list, we parse the list to check each publication record one by one. If we find one of them is older than

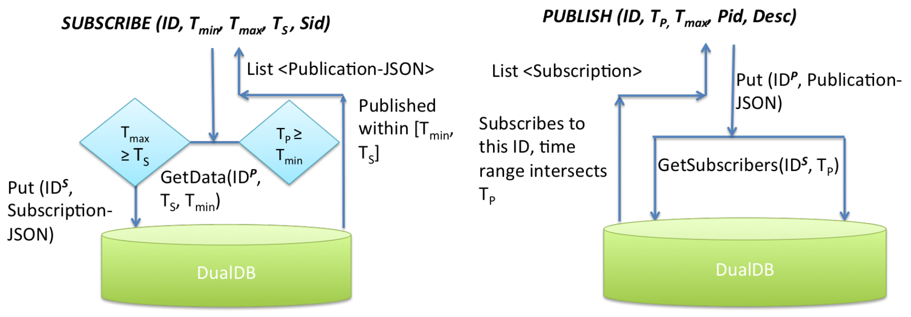
Algorithm 2 GetSubscribers(ID, T_P) operation

```
1:  $ListSB \leftarrow \emptyset$
2:  {Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3:  for $j$ from 0 to $L$ do
4:      if $ID$ is not in $Cj$ then
5:          NEXT
6:      end if
7:      List of Subscriptions $S \leftarrow$ the value of key $ID$ in $Cj$
8:      for $Subscription$ in $S$ do
9:          $T_{exp} \leftarrow Subscription(T_{max})$
10:         $T_{min} \leftarrow Subscription(T_{min})$
11:         if $T_P < T_{exp}$ and $T_P > T_{min}$ then
12:             ListSB.add($Subscription$)
13:         end if
14:     end for
15: end for$ListSB$
```

T_{min}, we can safely assume that we neither need to go any further down the level nor further right to this current list, and hence we stop the search. Lines 11-12 in algorithm 1 show this optimization.

4.5.3 Single Database Approach (DualDB)

A key observation is that *if continuous subscription queries and data publications could be stored in the same key space, then a single database (in LevelDB terminology) could store both of them*. Then query and data insertions would only need to access a single database, which could reduce the number of disk accesses and more importantly improve the caching efficiency. Further, the compaction cost might be decreased, as we only have to compact a single database.

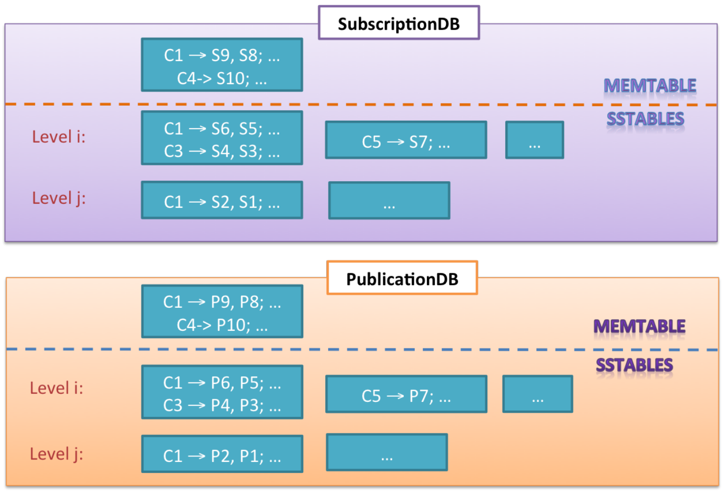

SUBSCRIBE (ID, Tmin, Tmax, TS, Sid)
List <Publication-JSON>
Published within [Tmin, TS]
Tmax ≥ TS
TP ≥ Tmin
GetData(IDP, TS, Tmin)
Put (IDS, Subscription-JSON)
DualDB
PUBLISH (ID, TP, Tmax, Pid, Desc)
List <Subscription>
Put (IDP, Publication-JSON)
Subscribes to this ID, time range intersects TP
GetSubscribers(IDS, TP)
DualDB

(a) Processing of a *SUBSCRIBE* Operation. (b) Processing of a *PUBLISH* Operation.

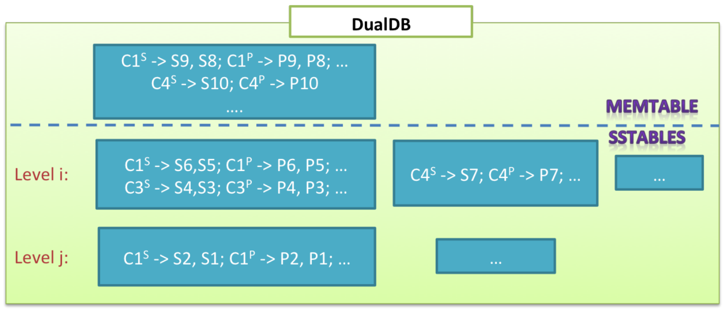
Figure 4.5: DualDB Approach

We propose to study the properties and performance of using a single database, which we call a *dual database (DualDB)*. The key idea is that the key-value data organization must be modified to accommodate a list of subscription and publication items in the value. That is, for a given ID (cell-id or topic-id) in our example, both the list of subscriptions and publications will be stored in the posting list of this ID. As LevelDB does not allow same key inside a component, we need to change the storage architecture to allow the disk and memory components holding at most two lists associated with the same key/cell-id. We assign a bit with the primary key (ID^S, ID^P), which will state whether it's a list of publications or subscriptions. Figures 4.5a and 4.5b show, respectively, how new query and new publication of events are processed in the proposed *DualDB*.

We can see that the key difference with the TwoDB approach is using single database against two and all the operations *Put*, *GetData* and *GetSubscribers* are performed with same *ID* with a assigned flag stating whether its a query or data. Here both *GetData* and *GetSubscribers*



(a) Two-Database Approach


DualDB
C1^S -> S9, S8; C1^P -> P9, P8; …
C4^S -> S10; C4^P -> P10
….
MEMTABLE
SSTABLES
Level i:
C1^S -> S6,S5; C1^P -> P6, P5; …
C3^S -> S4,S3; C3^P -> P4, P3; …
C4^S -> S7; C4^P -> P7; …
…
Level j:
C1^S -> S2, S1; C1^P -> P2, P1; …
…

(b) Single Database Approach

Figure 4.6: Snapshot of storage components for our LevelDB-style LSM database for pub/sub System.

procedure follows the same algorithms 1 and 2 of TwoDB approach respectively. The optimization techniques for lookups described in Section 4.5.2 can also be applied for *DualDB* in similar fashion. *DualDB*, similarly to *TwoDB*, cleans up the invalid records during compaction as described in Section 4.4.1.

As we are combining two databases into one database containing two lists, the storage components are changed accordingly to hold any such two heterogeneous data. Figures 4.6a and 4.6b show, respectively, snapshots of the entries in the corresponding databases for the Two-Database approach and the DualDB approach. In Figure 4.6a, we can see that the TwoDB approach holds posting lists of subscriptions (S1,S2,...) and publications (P2,P1,...) in two different databases and each list associated with a ID (C1,C2 etc.) is fragmented throughout different levels. Figure 4.6b shows the DualDB is holding lists of subscriptions and publications in same components with same key ID which holds a flag ($C1^S$,$C1^P$) that distinguishes between them. As in each component the records are sorted by key, these two lists will always be co-located with each other. This should allow improvements in both LevelDB memory cache and the operating system page cache. These figures assume a leveled storage architecture (as in LevelDB and Cassandra); a stack-based architecture can be handled similarly.

A broker in a pub/sub system can utilize these SUBSCRUBE and PUBLISH APIs as continuous queries if the storage layer is built on top of *TwoDB* or *DualDB*. Here compared to the repetitive approach, both of our approaches work on continuous queries and can guarantee that there is no data loss.

4.6 Extend to Complex Subscriptions

4.6.1 Self-Joining Subscriptions

As discussed in Section 4.5, our *TwoDB* and *DualDB* approaches can efficiently process simple publications and subscriptions. However, some scenarios require publications to also access other publications, in addition to querying the subscriptions (the same holds for subscriptions). We call these *self-joining publications (subscriptions)*.

Motivating Examples. Consider mobile users moving and subscribing to events on a region or cell. Imagine the scenario where a mobile user enters cell c1, then c2 and then c1 again, all within 10 seconds, and all subscriptions have a time interval of ± 10 sec. A desirable property for many applications is that data already returned when the user was in c1 the first time should not be returned again during the second entry. For that, the system must efficiently identify what data has already been returned to a user. This is a classic use-case for using self-join subscriptions where for a new subscription, we need to access not only the list of past matching publications, but also access the list of matching past subscriptions from the user in order to filter these results. As another application, consider a user who subscribes to a sensor temperature (or to a stock price) and wants to be notified if there has been a sudden change/drop in temperature/price. In this case, for a new publication, we need to traverse both the publications list to check past temperatures/prices in this location/stock, and the subscriptions lists to know who subscribes here.

That is, for these operations, a subscription (publication) must join with past subscriptions (publications), hence the name *self-joining* subscriptions. Our existing storage framework gives us flexibility to support such subscriptions efficiently. For both our approaches, if $SelfJoinFlag$ is set we assume that it is a self-joining operation.

For a self-join subscription, we write our subscription and/or publication to the storage and retrieve a list of events and a list of queries from storage by issuing a $GetDataANDSubscribers(ID, T_{P/S}, T_{min})$ operation. Here $T_{P/S}$ means execution time of either publication or subscription.

Algorithm 3 GetDataANDSubscribers ($ID, T_{P/S}, T_{min}$) operation on *TwoDB*

```
1: $ListPB \leftarrow this.PublicationDB.GetData(ID, T_{P/S})$ {[Algorithm 1]}
2: $ListSB \leftarrow this.SubscriptionDB.GetSubscribers(ID, T_{P/S}, T_{min})$ {[Algorithm 2]} $ListPB$, $ListSB$
```

Algorithm 4 GetDataANDSubscribers($ID, T_{P/S}, T_{min}$) operation on *DualDB*

```
1: $ListPB \leftarrow \emptyset$
2: $ID^P \leftarrow Setflag(ID, E)$
3: $ListSB \leftarrow \emptyset)$
4: $ID^S \leftarrow Setflag(ID, Q)$
5: {Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
6: for $j$ from 0 to $L$ do
7:     if $ID^P$ is not in $Cj$ and $ID^S$ not in $Cj$ then
8:         NEXT
9:     end if
10:    $ListSB \leftarrow$ {Apply Line 5-10 in Algorithm 2 for key $ID^S$}.
11:    $ListPB \leftarrow$ {Apply Line 5-12 in Algorithm 1 for key $ID^P$}.
12: end for$ListPB$, $ListSB$
```

In the TwoDB approach, the implementation for self-join subscription is straightforward:

we simply perform an extra GetSubscribers $(ID, T_{P/S})$ operation (Algorithm 2) on PublicationDB (along with *GetData* operation) to return the list of active matching subscribers for SUBSCRIBE operation and perform an extra GetData $(ID, T_{P/S}, T_{min})$ operation ((Algorithm 1)) on PublicationDB (along with *GetSubscribers* operation) to return the list of valid publications for PUBLISH operation.

In DualDB, both the lists of publications and subscriptions associated with the same ID are stored in the same disk block next to each other. We convert the *GetSubscribers* and *GetData* operations performing on same Dual database into a single operation, which means we can get the both the lists of publications and subscriptions on single operation. We first convert the key ID to ID^P and ID^S by setting appropriate bits. Then we search in our DualDB starting from Memtable and moving to SSTables level by level. If we find a match for either ID^P or ID^S , we fetch the resulting block form the storage components which might contain both the lists. We extract both the lists if they are available in that particular fetched block. Here we can see that one disk I/O might sufficient against two if the list of publications and subscriptions are co-located inside same storage file block.

Cost Analysis (*TwoDB Vs DualDB Vs RepQueries*)

Table 4.3 summarizes the number of disk accesses for each PUBLISH/SUBSCRIBE operation for simple and self-joining subscriptions.

For simple subscriptions, only one I/O is needed to write the subscription and the publication in SUBSCRIBE and PUBLISH operation respectively. For both the operation, we issue a read (GetData/GetSubscribers operation) on the database which may involve up to L (number of levels) disk accesses (because in worst case the postings list is scattered across all levels). Here, at most one disk access is required for level except level-0. As level-0 contains files with overlapping key ranges, we need to traverse all files inside level-0. Let us assume l_0 is the number of level-0 files. As both TwoDB and DualDB approach issue one write and one read for their each operation, the total cost becomes $(L + l_0) + 1$ disk accesses for both of them. *RepQueries* performs $\frac{T_{max} - T_S}{T_R}$ repetitive queries on publication database instead of a single subscription. But *RepQueries* publish operation only writes the publication record to the database.

For Self Joining subscriptions, similarly one write is needed for each operation. But for TwoDB approach, we need to submit two reads in two databases of publication and subscriptions (one in PublicationDB and one in SubscriptionsDB). This makes the total cost as $2 * (L + l_0) + 1$ disk accesses for each PUBLISH or SUBSCRIBE operation. But DualDB approach (Algorithm 4) only issues one read in dual database and the total cost becomes $(L + l_0) + 1$ disk accesses in the worst case. Note that we assume there is only one disk access per write operation. But in practice,

Table 4.3: Number of disk accesses for PUBLISH/SUBSCRIBE operation for simple vs self-joining subscriptions.

Approach:Operation	**Simple Subscription**	**Self-Join Subscription**
$TwoDB : SUB$	$(L + l_0) + 1$	$2 \cdot (L + l_0) + 1$
$DualDB : SUB$	$(L + l_0) + 1$	$(L + l_0) + 1$
$RepQueries : SUB$	$\frac{T_{max} - T_S}{T_R} \cdot (L + l_0),$	Not Supported
$TwoDB : PUB$	$(L + l_0) + 1$	$2 \cdot (L + l_0) + 1$
$DualDB : PUB$	$(L + l_0) + 1$	$(L + l_0) + 1$
$RepQueries : PUB$	1	Not Supported

LevelDB suffers from high write amplification factor where same data is written multiple times when moved from one level to another. This factor is excluded from our calculation.

4.6.2 Support for Hierarchical Attributes

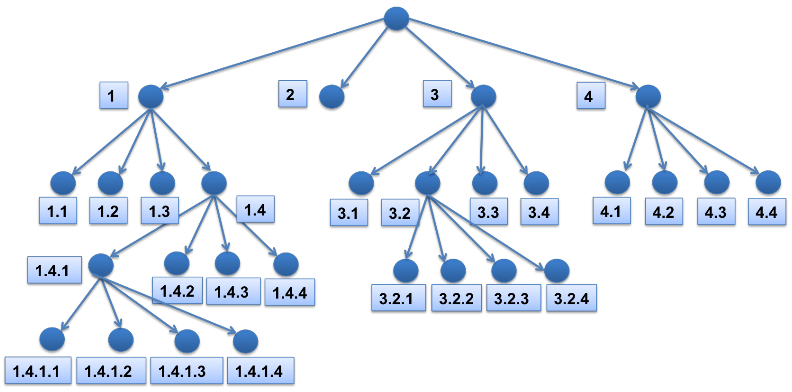
In this section, we introduce an optimization in our storage system to efficiently support multi-granular data, which can be defined by a hierarchical model. For example, we may have a spatial hierarchy (e.g. Country->State->County->City->Block), where each publication or subscription may select spatial cells of different granularity. Similarly, topics from a topic hierarchy can be selected. We propose and compare two approaches to support such hierarchical attributes in a pub/sub storage system. Note that the choice of an approach is orthogonal to the choice of a storage architecture from the ones described in Section 4.5.

Fixed Granularity Approach

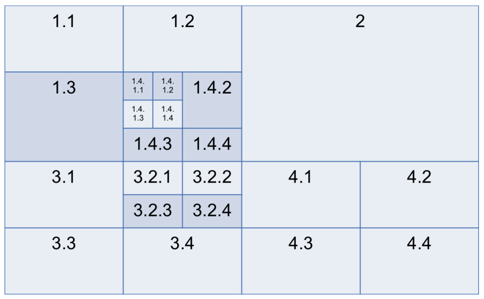
In this approach, both subscriptions and publications use the lowest level keys in the hierarchy. For example, if a subscription specifies a city which has 100 blocks, then we insert 100 copies of the subscription. Similarly we handle publications. A disadvantage is that multiple entries are needed for each subscription or publication. The key advantage is the simplicity, as this can be implemented as a preprocessing step on top of any storage framework.

Variable Granularity Approach

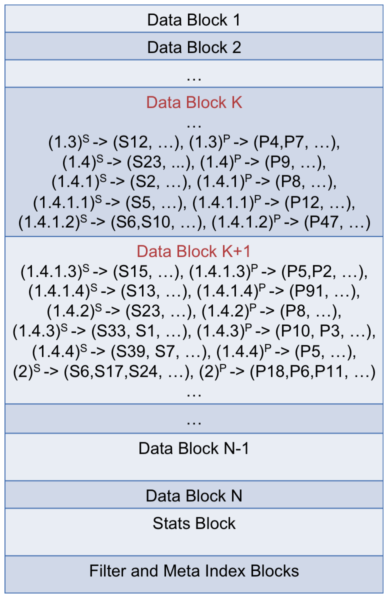
In this approach, we only have one entry per subscription or publication, and the storage framework manages the matchings. We propose to use Dewey encoding [92] (incremented integer numbers separated by dots) to represent each entity in the ontology, and this Dewey id becomes the key (ID) of the publication or subscription. Let us assume a hierarchy is defined by a quad tree.



(a) A Quad Tree and Dewey Encoding for each Node



(b) Partitioning Space using Quad Tree and Dewey Representation in Leaf Nodes


Data Block 1
Data Block 2
…
Data Block K
…
$(1.3)^S$ -> (S12, …), $(1.3)^P$ -> (P4,P7, …),
$(1.4)^S$ -> (S23, ...), $(1.4)^P$ -> (P9, …),
$(1.4.1)^S$ -> (S2, …), $(1.4.1)^P$ -> (P8, …),
$(1.4.1.1)^S$ -> (S5, …), $(1.4.1.1)^P$ -> (P12, …),
$(1.4.1.2)^S$ -> (S6,S10, …), $(1.4.1.2)^P$ -> (P47, …)
Data Block K+1
$(1.4.1.3)^S$ -> (S15, …), $(1.4.1.3)^P$ -> (P5,P2, …),
$(1.4.1.4)^S$ -> (S13, …), $(1.4.1.4)^P$ -> (P91, …),
$(1.4.2)^S$ -> (S23, …), $(1.4.2)^P$ -> (P8, …),
$(1.4.3)^S$ -> (S33, S1, …), $(1.4.3)^P$ -> (P10, P3, …),
$(1.4.4)^S$ -> (S39, S7, …), $(1.4.4)^P$ -> (P5, …),
$(2)^S$ -> (S6,S17,S24, …), $(2)^P$ -> (P18,P6,P11, …)
…
…
Data Block N-1
Data Block N
Stats Block
Filter and Meta Index Blocks

(c) Snapshot of an SSTable in DualDB containing lists in order of Dewey ID of the Key

Figure 4.7: Variable Granularity Approach by Using Dewey Encoding

Figure 4.7a illustrates how each node in different depth of the quad tree can be represented by a Dewey encoding. Figure 4.7b illustrates how we partition the space using that quad tree. During a *SUBSCRIBE/PUBLSIH* operation, we write the entry (publication/subscription) to the database and issue a range query from its Dewey to the Dewey of its next sibling.

For example given the quad tree partition in Figure 4.7, let a publication/subscription cover area (1.4) (or represent topic (1.4) if we use topics instead of locations as ID). The fixed granularity approach will issue PUBLISH/SUBSCRIBE operation for each areas/topics in the leaf level of the subtree of (1.4) [(1.4.1), (1.4.1.1), (1.4.1.2),(1.4.1.3), (1.4.1.4), (1.4.2), (1.4.3), (1.4.4)]. But in our optimized variable granularity approach, we will issue one write for this topic/area (1.4) and issue a range query from (1.4) to (1.5).

Next, we discuss how a range query based on Dewey IDs is efficiently supported, by modifying the approaches in Algorithm 1 and 2. We iterate from upper level to lower level, and for each level when we find the file that contains the start key, retrieve the value list associated with that key, and then instead of returning from there we continue iterating through that file (or next file if necessary) until the end key. As we know that the data in our storage files (i.e.SSTables) are lexicographically sorted by their keys (i.e. Dewey numbers) and partitioned into fixed size blocks. So given a Dewey Number representing an internal node, all the nodes descended from that node by pre-order traversal should be stored. That makes this range query very efficient.

Figure 4.7c represents a snapshot of an SSTable in our DualDB, where the spatial hierarchy is represented by a quad tree in Figure 4.7a and 4.7b. Here we can see that how the list of publications or subscriptions in SSTable are sorted by its key lexicographically. If we issue a range query from (1,4) to (1,5), our algorithm will look for the file which contains (1,4) in each level, retrieve the disk block K of that SSTable and retrieve the list of subscriptions/publications associated with (1,4). Then it will retrieve the next records of that disk block subsequently [(1,4,1),(1,4,1,1),...]

until it reaches the node (2). It might retrieve subsequent disk blocks in order to reach (1,5)/(2). In this figure, the last record of the block K is (1.4.1.2) and we need to retrieve block $(k+1)$. Note that similarly to our simple point lookup algorithm in Section 4.5.2, we check the time predicates of all the publications/subscriptions in each list for validity, and return early if pruning can be utilized.

Algorithm 5 summarizes how a single lookup *GetData* operation (Algorithm 1) for simple subscription queries can be extended to support a range operation *GetDataInRange* based on Dewey representation. Algorithm 2, 3 and 4 are extended similarly to support range operation on hierarchical attributes for simple and self-joining subscriptions.

Cost Analysis (Fixed Vs Variable Granularity Approaches)

We derive in Section 4.6.1 that a single PUBLISH or SUBSCRIBE operation can cost $O(L) = L + l_0 + 1$ number of disk accesses for both simple and self join subscriptions in *DualDB*. If we define the hierarchy as quad tree and the maximum depth as d, the number of leaf nodes in the worse case will be 4^d. So the cost for one operation will be $4^d * O(L)$ as we issue one operation for all leaf nodes separately. But for the variable granularity approach, we issue one write and a range query. Each range query similarly can hit $L + l_0$ number of files in each level and for each file, we may retrieve C consecutive disk blocks. C can be computed by dividing the size of the result set by the block size. The cost for each operation will be $C * (L + l_0) + 1$ number of disk accesses.

Table 4.4: Number of Disk Accesses for Hierarchical Attributes

Approach	**Approximate I/O Cost**
Fixed Granularity	$4^d * O(L)$
Variable Granularity	$C * (L + l_0) + 1$

4.7 Experiments

4.7.1 Experimental Setup

We ran our experiments on a machine with the following configuration: Processor of AMD Phenom(tm) II X6 1055T and 8GB RAM, with Ubuntu version 15.04.

Data and Query Workload

We used the Twitter streaming API to collect 15 million geotagged tweets, taking 12 GB in JSON format, located within New York State. We use this dataset to generate our desired workloads for the experiment. We ran experiments using different Subscription-to-Publication ratios. As there are too many parameters, we set the time intervals for each query and expiration time of each event to a constant value: all subscriptions have T_{min} as 10 seconds behind current time T_S and T_{max} as 10 seconds ahead of T_S, and all the publications have expiration time (T_{max}) of 20s ahead of current time T_P. For baseline *RepQueries*, we convert each SUBSCRIBE operation into 10 repetitive queries, each is executed after every $T_R = 1$ second interval.

There are several public workloads for key-value store databases available online such as YCSB [26]. However, to simulate a highly dynamic publish/subscribe model, we need to generate a mix of dynamically expiring subscriptions and publications. We decided to write our own script to generate different workloads from the real twitter dataset. Our workload generator considers each tweet as either a subscription or a publication depending on the Subscription-to-Publication ratio.

As all our tweets were collected within New York State, we use the bounding box rectangle around New York State and partition it to generate $500 * 500$ uniform sized cells each having a unique identifier cell-id. We map the Geo-location of the tweet to appropriate cell-id and use this cell-id as a primary key for input. If the tweet is an event (publication), the text is considered as event description. Tweet ID is used as either subscriber ID or publication ID. The time intervals are set according to the execution time of that particular operation as discussed above. In our dataset, as we have about 0.25 million cells and 15.3 million tweets, average number of tweets per cell is about 60 and as described, these tweets are converted into events and subscriptions.

Padres Baseline

In addition to the repetitive queries baseline (RepQueries) described in Section 4.5.1, we also compare to a popular Pub/Sub system, Padres. We understand that Padres is a content-based pub/sub and it also has a client-server architecture, which may incur additional overhead in delivering a subscription result, but we show that the performance difference is quite dramatic, which would dominate such overhead. First, we have to express our problem setting using Padres' model. For that, we convert our workload into Padres subscription and publication operations. Padres supports *historic subscriptions* on past queries. So, each subscriptions in our dataset is equivalent to one historic subscription and one regular subscription in Padres. Each event publication is also converted to a publish operation in Padres. We installed Padres locally containing one broker and two clients connecting to the broker. One client is subscribing queries and the other client is publishing events. For clarification, let's illustrate the dataset conversion with an example. Suppose at T_S, a subscription query with cid as cell-id and interval $[T_{min}, T_{max}]$ is issued. We then convert it to a composite subscription (Expression 4.1) and a regular subscription (Expression 4.2).

$$CS[class, eq, historic], [subclass, eq, events], [T, <, T_S], ..., [cellid, eq, cid] \& [T, >, T_{min}], ..., [cellid, eq, cid] \quad (4.1)$$

$$S[class, eq, events], [T, <, T_{max}], ..., [cellid, eq, cid] \quad (4.2)$$

A new generated event publication at T_P with cid as cell-id and T_{max} as expiration time will be converted to the following publication operation (Expression 4.3).

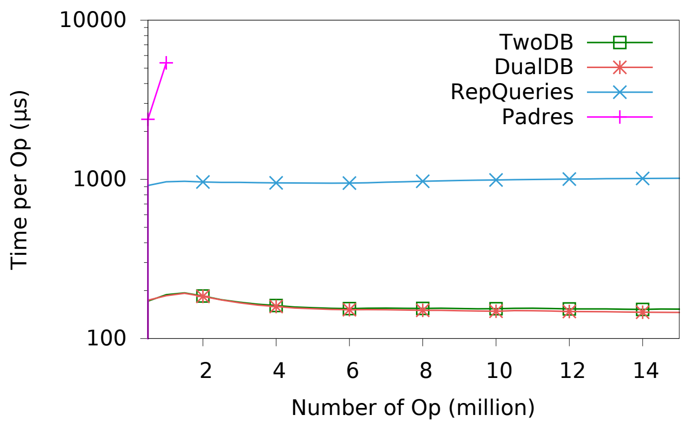
$$P[class, events], [T, T_P], [description, any], [cellid, cid] \quad (4.3)$$

4.7.2 Experimental Results

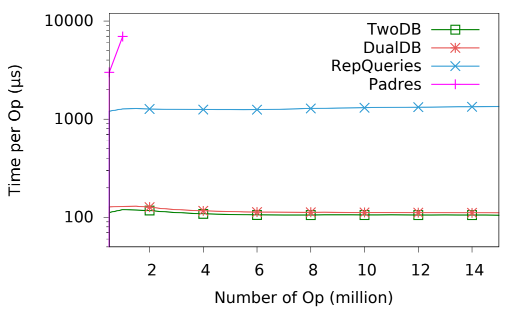
We conduct our experiments on workloads for both simple matching subscription and more advanced self-joining and multi granular subscriptions for different subscription-to-publication ratios, which represent different use cases.

Simple Subscriptions

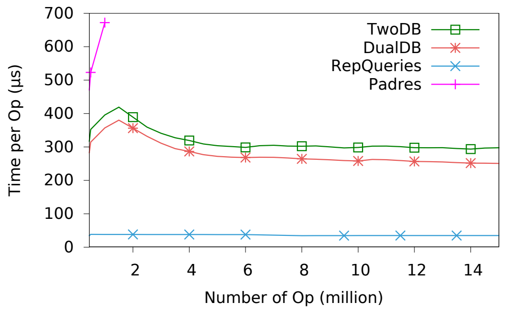
Figures 4.8 and 4.9 show the overall, SUBSCRIBE and PUBLISH performance of all systems subscription heavy ($\frac{Subscription}{Publication} = 3$) and publication heavy ($\frac{Subscription}{Publication} = \frac{1}{3}$) workloads, respectively. For example, our moving subscriber application is a use-case for subscription-heavy workloads whereas following Twitter accounts is a use-case of publication-heavy workloads. In all figures, we record the performance once per million operations. We display the cumulative to-



(a) Overall performance



(b) SUBSCRIBE performance

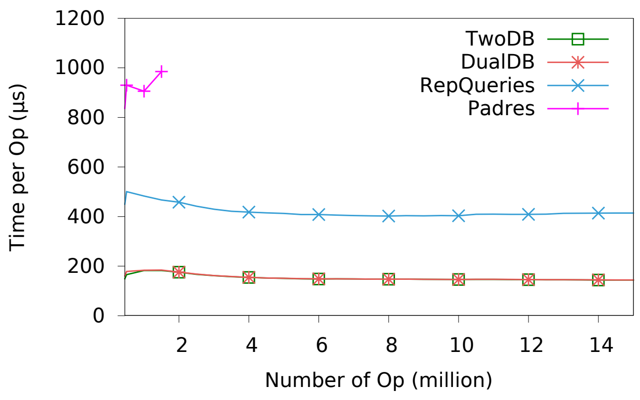


(c) PUBLISH performance

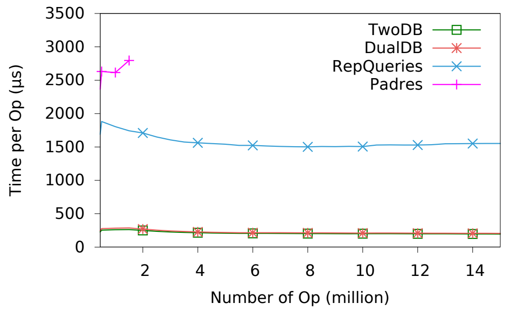
Figure 4.8: Performance of different storage variants for simple subscription queries on subscription heavy workload

tal time taken for both PUBLISH and SUBSCRIBE operation separately and also collectively and calculate average time per operation in every million operations.

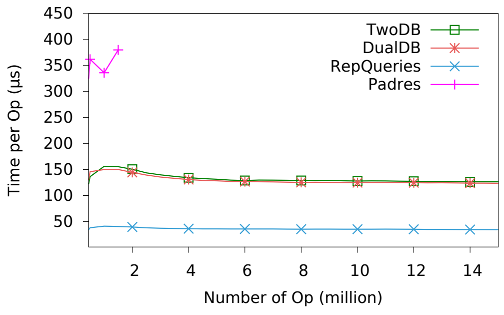
Figures 4.8 and 4.9 show that if the system relies on repetitive queries instead of instant response queries, it can not scale to millions of operations. As *RepQueries* does not issue any read after each publication, and only issues a write to a single database, PUBLISH has very good performance as expected. But SUBSCRIBE has bad performance as *RepQueries*. The overall per-



(a) Overall performance



(b) SUBSCRIBE performance



(c) PUBLISH performance

Figure 4.9: Performance of different storage variants for simple subscription queries on publication heavy workload

formance is far worse than our proposed instant-response variants. Both *TwoDB* and *DualDB* approaches have about 1000% better performance than *RepQueries* for subscription heavy workload, and 300% better for publication heavy workload.

According to the theoretical analysis in Section 4.6.1, both *TwoDB* and *DualDB* approaches cost the same number of disk accesses for each PUBLISH and SUBSCRIBE in the worst case. This is confirmed experimentally in Figure 4.9, where we find that they have identical performance for publication heavy workload.

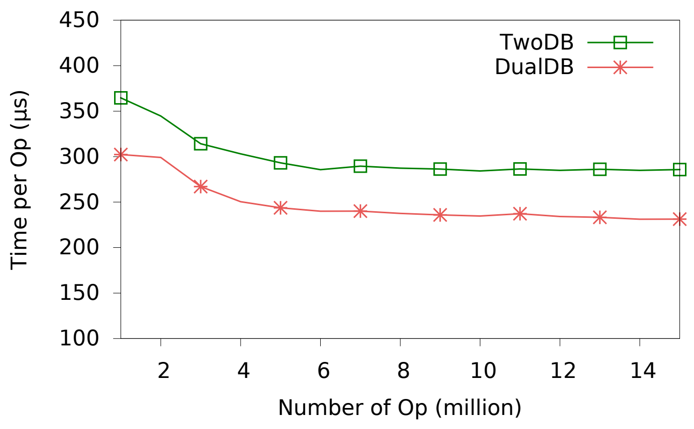
Figure 4.8 shows the results for subscription heavy workload, where *DualDB* is better than *TwoDB* by about 5%. This behavior is expected in *DualDB*, although it does not perform any extra disk access than *TwoDB* theoretically. The reason is that *DualDB* holds both query and data associated with a key inside the same disk block or in a neighbor disk block, which leads to better cache performance for both OS page cache and LevelDB block cache.

We see in Figures 4.8 and 4.9 that Padres performs poorly not only compared to our two approaches, but also to the repetitive baseline. Also, in our experiments Padres is not able to cope with the increasing number of subscriptions, and runs out of memory very quickly (even before 2 million operations) and the system crashes. This is because it relies on an in-memory data structure to manage subscriptions and fails to cope with a very large number of queries. Note that we allocated maximum memory for Padres (i.e. 8GB) and it still runs out of memory. Here we can see that even if we have sufficient memory to support small number of subscriptions, the use of traditional SQL-like database perform much worse (up to 300%) than even our baseline *RepQueries* approach.

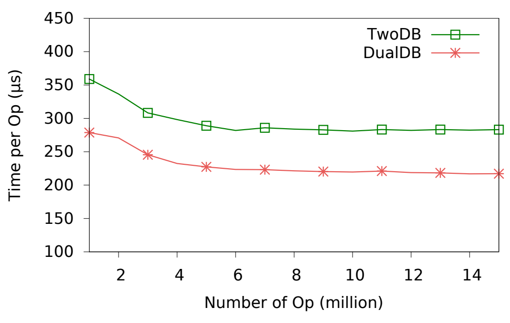
Self-Joining Subscriptions

Similar to the simple matching subscription, we observe the performance for complex self-joining subscriptions. Figures 4.10 and 4.11 show the overall, SUBSCRIBE and PUBLISH performance of *DualDB* and *TwoDB* approaches on both subscription and publication heavy workloads, respectively. Here we do not consider baseline *RepQueries* approach, which relies on repetitive queries on Events database, because it does not store the subscriptions in a database and hence it cannot perform self-joining queries.

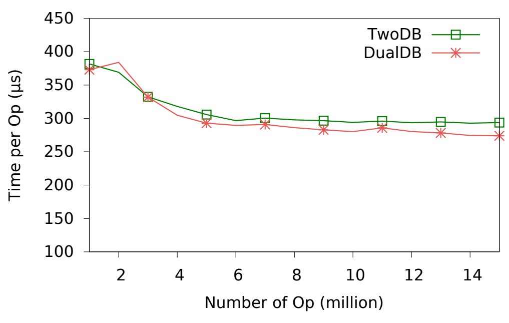
Figure 4.10 shows that *DualDB* is overall 20% (About 6% on PUBLISH and About 25%



(a) Overall performance



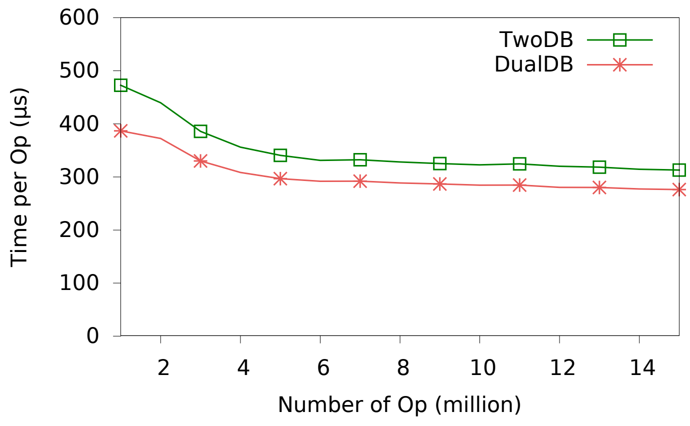
(b) SUBSCRIBE performance



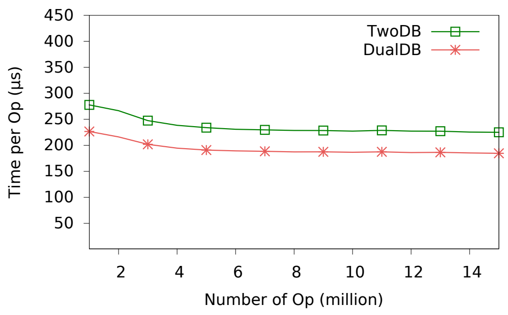
(c) PUBLISH performance

Figure 4.10: Performance of different storage variants for Self-joining subscriptions on subscription heavy workload

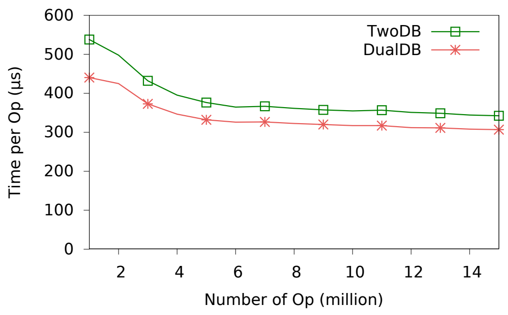
on SUBSCRIBE) faster than *TwoDB* for subscription heavy workload. We see similar performance gain of 12% overall improvement for *DualDB* over *TwoDB* in publication heavy workload. Note that here PUBLISH performs much better than subscription heavy workload which is desired in a publication heavy workload.



(a) Overall performance



(b) SUBSCRIBE performance

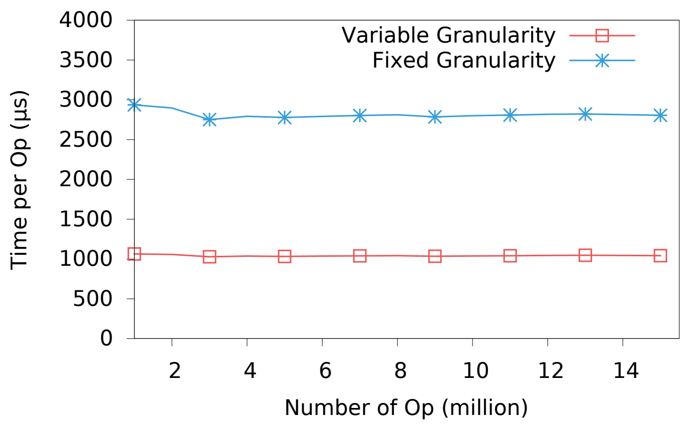


(c) PUBLISH performance

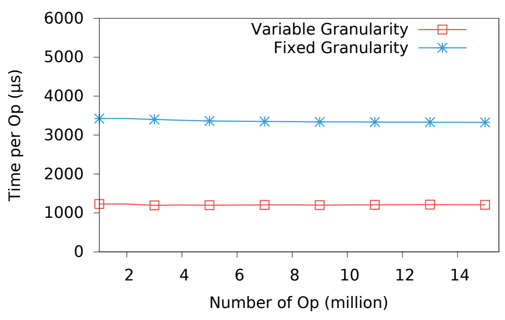
Figure 4.11: Performance of different storage variants for Self-joining subscriptions on publication heavy workload

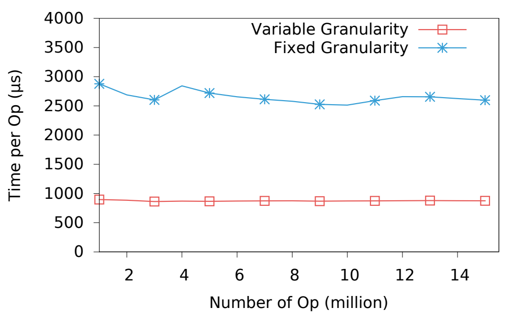
Subscriptions for Multi Granular Data

To simulate the environment for multi granular attributes, we partitioned the space by a balanced quad tree of depth 5 instead of a uniform grid. Here each cell represents a node in the tree and each cell-id is encoded by Dewey Numbering (described in Section 4.6.2 and Figure 4.7). We set subscription-to-publication ratio as 1. This workload generates a mix of subscriptions and publications from different levels. We argue that the smaller cells will likely be accessed more than



(a) Overall performance





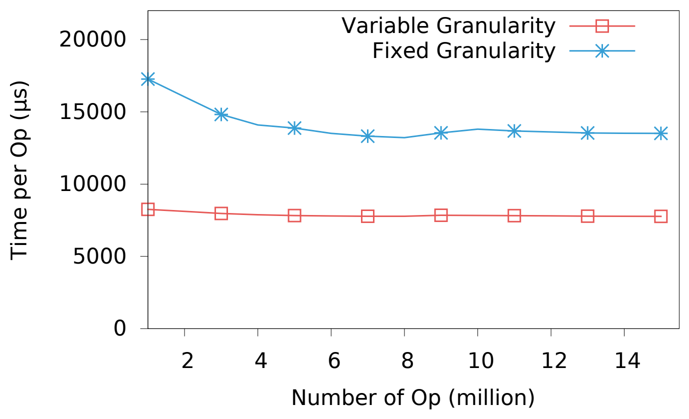
(b) SUBSCRIBE performance

(c) PUBLISH performance

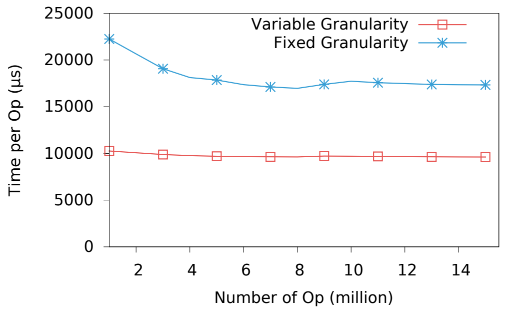
Figure 4.12: Performance of Pub/Sub in DualDB for high granular data in Different levels of hierarchy for whole workload

larger cells up in the hierarchy. For that, we design the workload generator in such way that a cell in level i has probability proportional to $\frac{P}{2^{d-i}}$ to be selected, where d is the maximum depth of the tree. In parallel, we also generated the workload for the fixed granularity approach by translating each operation into a multi operation (one operation for all the lowest level cells).

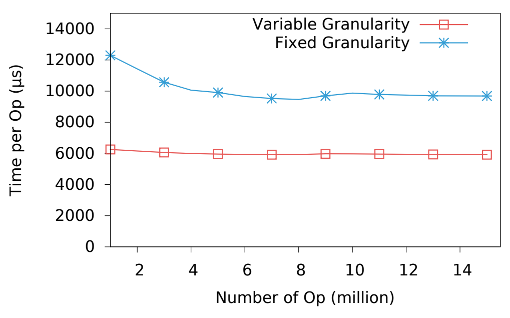
Figure 4.12 shows the overall, SUBSCRIBE and PUBLISH performance of *DualDB* for this workload of variable granularity against the fixed granularity approach. We see that our in-



(a) Overall performance



(b) SUBSCRIBE performance



(c) PUBLISH performance

Figure 4.13: Performance of Pub/Sub in DualDB for high granular spatial data covering area of different radii

telligent range query based variable granularity approach performs about 300% better for both PUBLISH and SUBSCRIBE operation. Figure 4.14 shows the performance for both approaches in different levels. It shows the average time per operation after performing 15 million operation. We see that our variable granularity approach performs even better for upper levels in the hierarchy. These results justify the cost analysis in Section 4.6.2.

In the above experiment, we assume that a subscription or publication must pick one

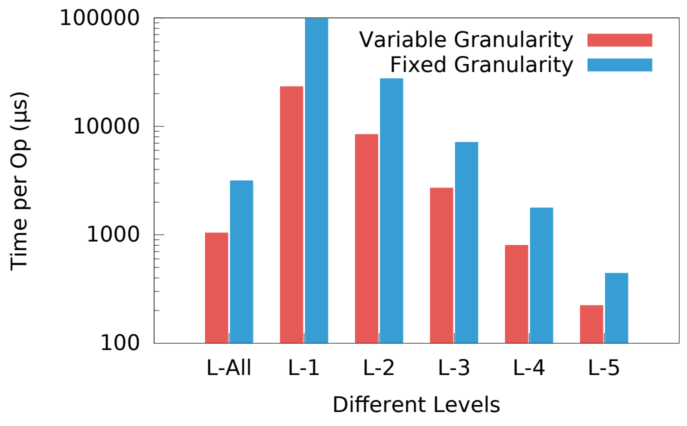


Figure 4.14: Performance comparison in different levels of Hierarchy

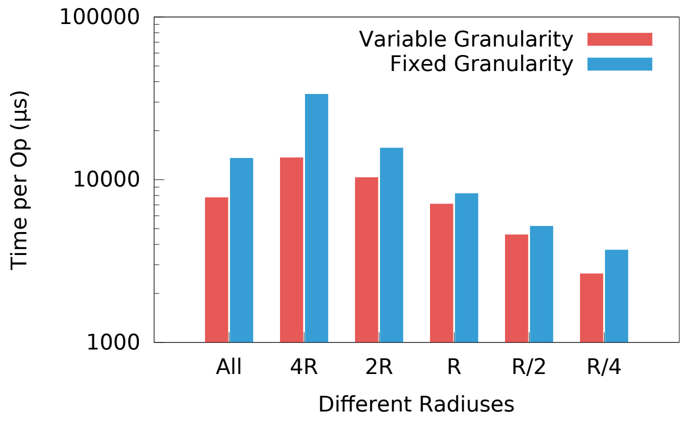


Figure 4.15: Performance comparison for different radii

of the existing geospatial regions of the quad tree. Next, we show how arbitrary spatial regions, specifically circles can be supported, represented by a geo-location point and a radius. We generated a workload from our Twitter dataset for this application, where we select a radius for each operation from a set of five radii ($4*R$, $2*R$, R, $R/2$, $R/4$). Here, R is the length of the smallest cell in the quad tree. For the variable granularity approach, each operation's geo-location and a randomly selected radius (from the pre-defined set) is mapped to the cell in our quad tree whose area fully contains the circle region. For fixed granularity approach, this circle is mapped to all the cells in the lowest level whose rectangular region intersect with this circle. We perform one operation for all of these cells separately. Figure 4.13 shows that our variable granularity approach performs about 200% better for both PUBLISH and SUBSCRIBE operations. Figure 4.15 shows the performance for both approaches for different radii. We see that the performance gain of our variable granularity approach becomes greater for larger radii.

4.8 Conclusions and Future Work

In this paper we present efficient storage and indexing approaches to achieve high throughput publish/subscribe on LSM-based databases where both the number of subscriptions and publications are massive in scale and one or both of them can arrive and expire with time. Our approaches support instant notifications. We also consider a baseline approach that relies on repetitive queries. We also show how hierarchical attributes in multi granular data can be supported efficiently by a Dewey encoded representation.

We implement these storage frameworks on top of the popular LSM-based LevelDB system and conduct extensive experiments using real datasets. In contrast to our *TwoDB* approach, *DualDB* approach is harder (in terms of lines of code) to implement on an existing NoSQL storage, but it offers better performance and more flexibility. The experimental results show that our proposed approaches outperform the state-of-the-art *Padres* pub-sub system (by up to 3000%) and also outperform the repetitive baseline *RepQueries* (by up to 1000%). We also show that our *DualDB* approach outperforms the *TwoDB* approach for simple subscriptions by a small margin (up to 5%) and for self-joining complex subscriptions by larger margin (up to 20%). For hierarchical attributes, we show that our variable granularity approach based on Dewey encoded IDs performs much better (up to 300%) than the fixed granularity approach.

In the future, we plan to extend this work to a distributed environment. We also plan to allow more complex subscription queries that do not match based on a primary key conditions, where any of the existing secondary indexing techniques [86] can be used on top of our proposed frameworks.

Algorithm 5 GetDataInRange ($StartID^P, EndID^P, T_P, T_{min}$)

```
1:  $ListPB \leftarrow \emptyset$
2:  {Starts from Memtable (C0) and then moves to SSTable organized into L Levels. C1 is LevelDB's level-0 SSTable.}
3:  for $j$ from 0 to $L$ do
4:    $ListSSTs \leftarrow getOverlappingSSTablesInsideLevel(j, StartID^P)$
5:    for $f$ in $ListSSTs$ do
6:      $Iterator \leftarrow getIterator(Cf)$
7:      $Iterator \leftarrow Iterator.Seek(StartID^P)$
8:      while $Iterator$ is Valid do
9:        if $Iterator.Key.getFlag()$ is Publication then
10:         List of Publications $P \leftarrow Iterator.Value$
11:         $ListPB \leftarrow$ {Apply Line 6-9 with value of $T_P$ and $T_{min}$ in Algorithm 1 to Populate $ListPB$ from $P$}
12:       end if
13:       $Iterator \leftarrow Iterator.Next()$
14:       if $Iterator.Key >= EndID^P$ then $ListPB$
15:       end if
16:     end while
17:   end for
18: end for$ListPB$
```

Chapter 5

Spatial-LSM: Efficient Spatial Partitioning in LSM-based databases

5.1 Introduction

Spatial indexes in traditional relational databases supported spatial queries in pre-*big data* era. However, the volume and ingestion rate of spatial data is increasing rapidly in modern applications such as Social Media, Disaster Management, Climate Science, Politics, Urban Traffic, Supply Chain Management, Marketing/Advertising etc. These applications need to ingest and maintain billions of spatial data coming in rapid pace. LSM-based databases such as AsterixDB [2] [7], Cassandra [77], LevelDB [29], HBase [73], BigTable [22], InfluxDB [58] has become very popular as NoSQL and NewSQL databases. Many popular big data storage systems begin supporting spatial indexes on top of their NoSQL framework (AsterixDB LSM-Rtree, GeoMesa, STEHIX on Hbase etc). But they all built a separate module on top of existing framework which is optimized for non-spatial indexes. These techniques are usually good for ingestion and also show faster spatial queries than non-spatial indexes. But their low-level storage organization are not optimized for spatial data. For instance, their disk components share spatially overlapped regions. This motivates us to our next research direction: can we spatially organize and partition database components in a optimized way so that their spatial overlapping are minimized. Therefore, we can support significantly faster spatial queries. This will also reduce extra efforts to build separate module on top of an LSM database.

Following this direction, we extensively studied on how LSM-trees can be optimized for one dimensional indexes. Based on that, we designed techniques that can be applied to optimize LSM-tree for two-dimensional spatial indexes. As the disk components in the LSM-tree are immutable, we need to find a good way to reorganize them during background merge compaction. The goal is not only to minimize the spatial overlap between them so that spatial queries can be faster, but also to minimize overhead on insertion and Compaction and to write amplification factor. Our level based merge compaction techniques effectively partition the disk components such a way so that there is no spatial overlapping between them inside a level. This organization helps to prune more disk components for given spatial queries. This technique closely follows one dimensional index compaction for Leveldb. Our experimental results show that our approaches achieve faster spatial query response time with slightly increasing write amplifications.

5.2 Related Work

In chapter 2, we described how LSM-tree index works and how they are implemented on top of level-based compaction (e.g. LevelDB, Cassandra) and stack-based compaction (e.g. AsterixDB). All the LSM based spatial indexes so far is built on top of existing LSM techniques which relies on primary one-dimensional key indexes. For example, GeoMesa[46] has spatial index built on top of a key-value store Accumulo which closely follows Google's BigTable's design. In Geomesa, the space is partitioned by space-filling curve (z-curve or hilbert-curve). Each partition is mapped to a key and based on that they maintain a list of records for each key partition in the kv store. Given a spatial region, the query returns a list of range of these z-partitions. A post-processing is needed to prune records where a segment of a partition is overlapped with the query region.

STEHIX [24] follows similar approach, but their spatial index is built on top of Hbase. They also build their indexes based on space filling curves.

As described in Chapter 2 AsterixDB software framework enables âĂIJLSM-ificationâĂİ (conversion from an in-place update, disk-based data structure to a deferred- update, append-only data structure) of any kind of index structure that supports certain primitive operations, enabling the index to in- gest data efficiently. They use r-tree for spatial index on top of LSM. Here also LSM follows stack-based compaction which is not optimized for spatial data.

5.3 Approaches

In LSM-tree, disk components are immutable, that is why too much computation hurts faster ingestion. On the other hand, too much reorganization of disk components increase write amplification factor, which is (No of bytes written)/(No of bytes expected to be written).

To design our approaches, we try to reach the following objectives: a) Partition LSM disk components during merge compaction based on their recordsâĂŹ spatial attribute. b) Spatial queries should access fewer components. c) Data can be spatially divided into different components. d) Should not rapidly increase write amplification.

5.3.1 MBR based Leveled Partitioning

Each component can be represented by a Minimum Bounding Rectangle (MBR).

In this approach, components will be organized in different levels like LevelDB. We maintain a spatial partitioning of the data in each level. Components in same level should be spatially non-overlapping. Given a spatial query rectangle, we can efficiently find which components overlap with query region Prune lots of disk components.

We extend leveled LSM storage architectures, which have been used to organize records by their primary key, to build spatial (or spatiotemporal) LSM indexes. The key idea is that at each level, given the spatial query region, we can efficiently find which few files overlap and hence minimize the I/O. For that, we maintain a spatial partitioning of the data in each level by viewing each file as an MBR in R-tree nomenclature. A key challenge is how to maintain these MBRs at each level efficiently during compactions.

The idea of level-based partitioning comes from level-based R-tree indexing where it groups nearby objects and represent them with their minimum bounding rectangle (MBR). For our level-based partitioning algorithm, we are considering level-based compaction similar to LevelDB/RocksDB compaction which has been found successful for many workloads. Similar to LevelDB paradigm, in our solution, components within a level should contain different space partitions or MBRs. We can expect overlapping of partitions within a level, but the heuristics will be to minimize that to as lower as possible.

Upper level components cover bigger MBR while as we go down our LSM tree in lower levels, components will cover smaller MBR. All the partitions are expected to have uniform size. As in leveled compaction, we are merging files from upper levels to lower levels, instead of combining these components into a bigger one, we will create smaller components with different MBRs with minimal overlapping. This will enable us to partition into more regions or MBRs and this might also result in uniform sizing among the partitions.

Architecture

One key observation in the architecture of level based partitioning is that here each level is ordered by time and inside each level, components are partitioned by space.

Compaction Policy: We need to find a suitable compaction policy for our architecture. Lets illustrate this LSM paradigm. We assume memory component will have fixed size. When it is full, it will be flushed to disk and kept to level-0 temporarily. This level-0 file will trigger a compaction with all overlapping N number of level-1 files. We will create $N+1$ files at least in such way so that there will be minimum overlapping.

Now we assume each level is exponentially larger than its upper level. We keep the maximum size of the level as our threshold for when to trigger a compaction for components that reside in a level. If the size surpasses the threshold we will pick one file (base on some heuristics) and compact with overlapping (Either all or some of them) files in lower levels and compact them (Ensuring the minimum overlapping).

Figure 6.3 shows how the LSM components in the lsm-tree are moved into different levels and spatially organized into different regions.

Compaction Component Selection: If a level is full, pick a component to merge with next level components. We follow to strategy to select which component to pick for compaction:

- **Round Robin:** Here we follow round-robin strategy on z-ordering to select which file to pick

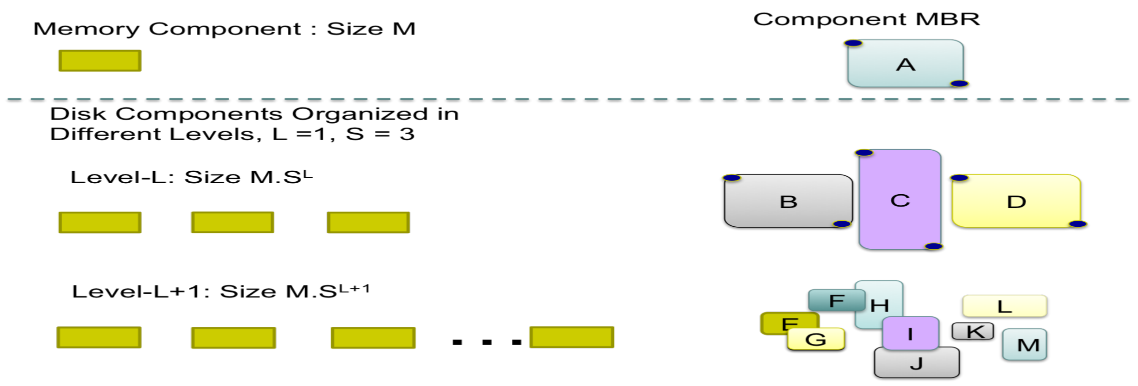

Memory Component : Size M
Component MBR
A
Disk Components Organized in Different Levels, L =1, S = 3
Level-L: Size M.S^L
B
C
D
Level-L+1: Size M.S^L+1
F
H
L
E
G
I
K
M
J

Figure 5.1: Level Based Spatial Partitioning: How the components are moved into different levels and spatially organized into different regions.

for compaction. Here, we maintain a z-value of the center of each componentâĂŹs MBR. A cursor rotates through the z-values representing the disk components. The component next to current cursor is selected for compaction. Cursor moves forward after each selection. This strategy upgrades the selection process used in LevelDB for one-dimensional index to spatial index. Round Robin technique ensures that the data is uniformly distributed in different levels.

- **Pick Best:** Here, we follow a greedy method to heuristically choose the best component to merge. We consider the component which has least number of overlapping components with the next level as the best component.

Partition Algorithm:

Figure 5.2 shows how an ideal partition algorithm inputs the merged components' points and create new components with same size but spatially non-overlapping.

In out implementaion, we used R-tree packing algorithms (e.g. Sort-Tile-Recursive) to partition our components. STR just sort points by x-axis and divide into vertical slices and then sort

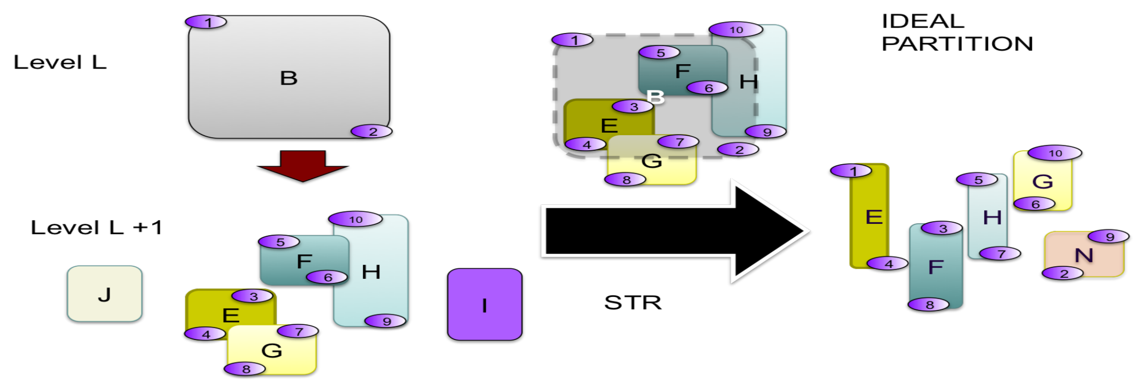


Figure 5.2: An Ideal Partitioning Algorithm.

by y-axis and divide into horizontal slices. In our partitioning algorithm, given X number of components of same size who have spatially overlapping points, STR creates X number of components of same size with no overlapping between them.

Implementation

We implemented our approaches on top of AsterixDB which is a popular open source distributed LSM based big data storage system. For simplicity, at first we want to implement our level based compaction policy only on their secondary indexes. In AsterixDB as previously described, the secondary indexes can be a R-tree on location where inside each disk components the records are indexed on a R-tree. But the merges on different disk components are time-correlated and there is no spatial considerations there. We will implemented level-based partitioning of the disk components to provide spatial partitioning on the secondary indexes on points so that we can achieve faster spatial queries and also have faster non-spatial queries.

5.3.2 Z-value based Leveled Partitioning

Each component can be represented by one dimensional z-value range of their points. In this approach, we convert each two dimensional point to their one dimensional value z-value. In this case, merge compaction will follow exactly the leveldb technique which considers only one dimensional value. Components in same level should have non-overlapping z-ranges. Given the spatial query MBR, we convert it into multiple z-ranges similary to GeoMesa technique and use them to find and prune non-overlapping components.

5.4 Experimental Evaluation

As described earlier, we developed our techniques on top of AsterixDB. Level based spatial merge compaction is applied on their LSM-Rtree indexes. We compare our techniques with default AsterixDB merge policy.

Dataset: For our experiments, we used Open Street Map dataset which consists of almost 100GB (i.e. about 2.7 billion points). We ingested this data through feed. We measure write amplification factor as the database grows. We calculate write amplification factor by diving the total number of bytes written in merge, by number of bytes flushed.

For measuring query response, we randomly selected points from the dataset. The query points are generated randomly from the dataset with different pre-defined rectangle (Selectivity: $10e-9, 10e-6, 10e-3, 10e-1$). We used 1000 queries for all of selectivity. We measure mean mean query response time for all the selectivity queries. Figure 5.3 shows that our leveled partitioning algorithm shows better write amplification when the database is smaller. But AsterixDB

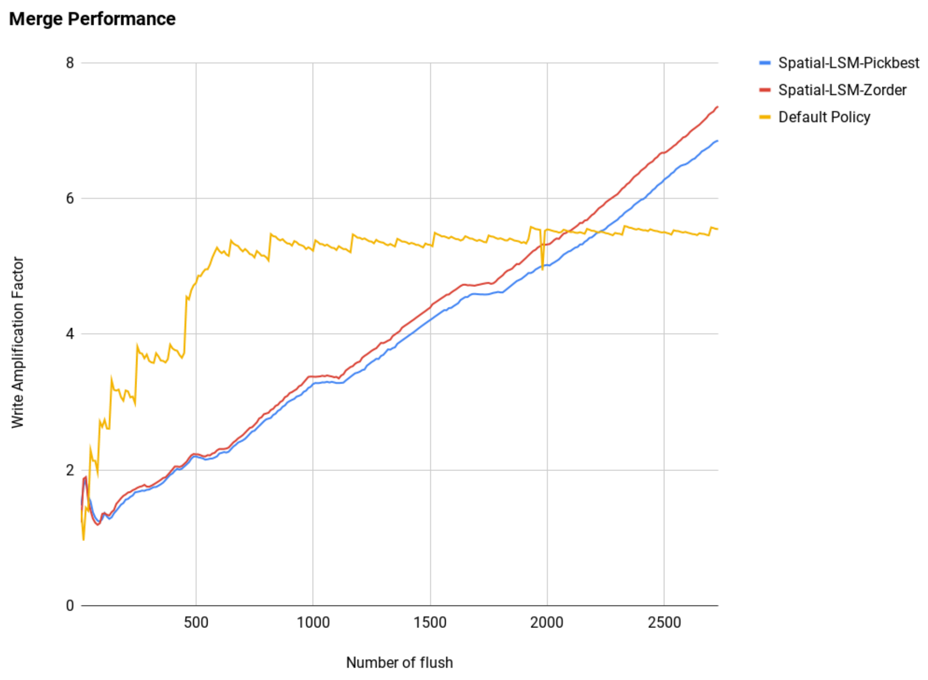


Figure 5.3: Performance of merge compaction in Leveled partitioning

default policy stops merging the components after a component reaches a threshold size. That is why after some time, their write amplification factor become constant. Figure 5.4 shows that our approaches perform better for spatial queries. Our approaches perform even better for bigger query regions.

5.5 Conclusion and Future Direction

In this paper, we present novel approaches to optimize LSM for spatial data. We propose a level based organization of disk components where the spatial regions between the components are non-overlapping. We show that our approaches offer superior performance for spatial queries

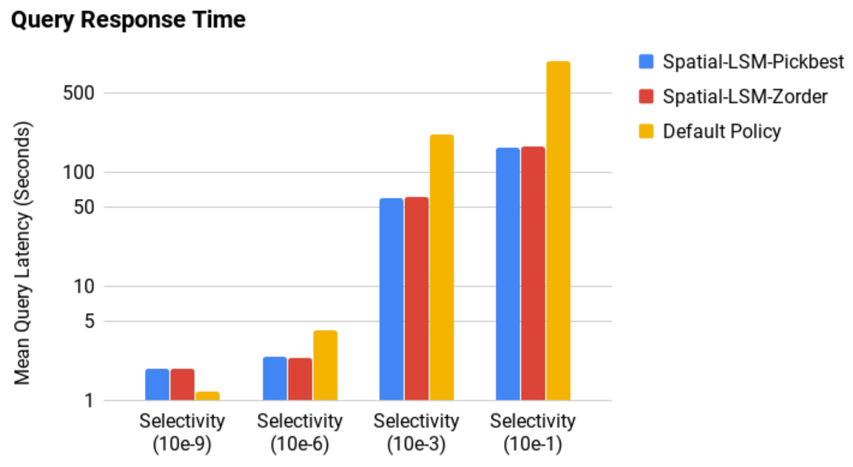


Figure 5.4: Spatial Query Response Time

with a small overhead on compaction. In future, we plan to extend our architecture from spatial to spatio-temporal partitioning LSM Components. We also plan to extend these techniques to primary data tables.

Chapter 6

Efficient Paxos-based Block Storage Replication Engine

6.1 Introduction

Block Storage Systems like AWS-EBS [25], Rackspace [69], Ceph [95] etc becomes very popular nowadays as they can provide fixed-sized raw storage capacity. Each storage volume can be treated as an independent disk drive and controlled by an external server operating system [37]. Block storage is ideal for clustered databases (e.g. Aurora [48]), RAID Volumes, mission-critical applications like Oracle, SAP, Microsoft Exchange etc. The block storage services are expected to be highly available and fault tolerant in a geo-distributed cloud. Typical block storage volume has a simple mapping layer to export the virtual device blocks to physical blocks. This map and as well as the physical data needs to be replicated. For that we need a high performance and I/O efficient fault tolerant system which will act as a stand-alone replication engine maintaining consistency and reliability of the block storage throughout the distributed cloud.

Paxos [78] is a distributed consensus protocol which can work in a network of unreliable processors. Although there exists some replicated block store which does not rely on consensus algorithms (e.g. Linux DRBD [31]), but Paxos is theoretically proven as accurate and used where durability is highly required (e.g. to replicate a file or a database). Paxos is also considred to be as efficient as other alternative consensus algorithm, Raft [85]. Paxos based distributed block storage systems are not explained in depth in literature /open source world. The main reason is that it is very hard to implement Paxos where performance heavily depends on implementation. There is some academic work which proves that Paxos state machine replication is the basis of high performance data store (i.e. Gaois [16], Gnothi [94]). But their work is close source and there is not enough technical details on how a logical volume can be efficiently separated from actual physical storage in different replicas. Also we invent a IO optimized write path for block storage in our architecture.

Contributions:

- We studied how a Paxos variation Multi-Paxos [85] which replicates a Paxos Log in a distributed cluster can be used to implement a reliable block storage replication engine. Based on our study, we designed and implemented a fault-tolerant block storage system Hydra which performs Log Replication (i.e. State machine replication) using WeChatâĂŹs open-source Paxos library PhxPaxos [63] [98].

- We further modify Paxos protocol to achieve I/O optimized block store by redesigning Paxos core architecture. Here we reduce about 50% disk I/O by introducing a shared storage used by Paxos Log and Volume File.

- We embed a persistent logical block to physical block mapping (via a lightweight KV store leveldb [55]) in our replicated state machine which can be used for features like deduplication, snapshot etc. We successfully designed the separation of global logical volume to replicaâĂŹs local physical storage in our replication scheme with maintaining consistency and efficiency.

- Overall, our system is designed in way that ensures availability, strong consistency, atomicity, reliability, IO efficiency in distributed cloud storage.

6.2 Problem Definition/Block Store Framework

In Multi-paxos, there is a distinguished leader and any read, write and delete request is redirected to master. Master always acts as a proposer and triggers all Paxos related operations. We also have a ReadLocal API which can take advantage of eventual consistency and any client can read from its local store for faster read operation.

Write ((int N, int S, bytes[] data): Write operation takes a logical block number N and raw block data as well as its size S. One block/multiple subsequent blocks can be written in the replicated in our distributed block storage. If the operation is sent to a replica who is not master, it will redirect the operation to the master.

ReadGlobal/ReadLocal (int N, int S, bytes[] data): Read operation returns data blocks of size S, starting from block number N.

Delete (int N) Delete is expected to delete a particular block given a block number N from the block storage.

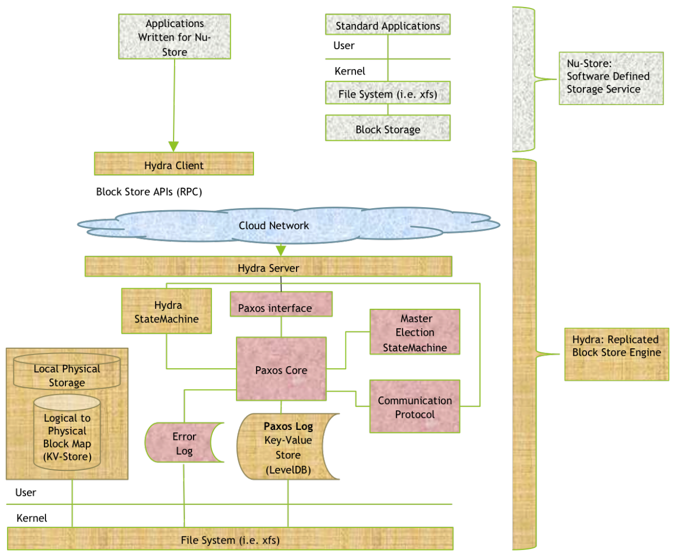


Figure 6.1: Block Store Architecture

6.3 System Architecture

How the Hydra APIs are exposed Our block store Hydra ServerâĂŹs APIs can be exposed to the upper layer software defined storage services. There is two write paths to the block store. First, we can write any application (e.g. database) who will directly call Hydra Client to talk to Hydra Server in cloud network via Remote Procedure Calls (RPC). Second, we can intercept from kernel when a standard application tries to write a block in disk/file system. We will redirect the write operation through iSCSI protocol and directly talk to Hydra Server through cloud network.

How Hydra Servers works Hydra servers talk to each other via paxos protocol. Block size is set fixed at start up of the system. Each Hydra server maintains a block store state machine which is consistent between all the replicas. Each operation coming from Hydra client/block storage is considered as an action in block store state machine. Our state machine consists of two components. One is a write optimized Key-Value store containing logical to physical block number map which is global for all replicas. Other is an append-based local file store where raw data is stored sequentially block by block. We maintain a local counter (i.e. Next available block number) to keep track where to store our next logical block. Consider that here our physical storage is sequential with the order of the writes, regardless of the order from logical block numbers.

Volume size does not need to be set at startup and this way we can dynamically scale the block store. Paxos Node information are set at startup and it is stored in a System State Machine. We can reconfigure Paxos cluster (add/remove member) on run time which is an operation/action on System State Machine and will be automatically updated. A Master State Machine is maintained for consistent master election management.

A volume may consist of multiple âĂIJshardsâĂİ (or segments) which can run in parallel. Each of which is replicated by a paxos group. Each group has their own master and runs in parallel to achieve higher throughput for concurrent block operations. This way we ensure high performance and scalability for block storage. Paxos groups can be selected based on simple hash function on Logical block Numbers. A master is e

6.4 Detailed Operational Flow and Optmizations

Write path and optimization Any write request is written by Paxos Log two times during consensus. (i.e. acceptor state and proposer state). For block store operations, prepare phase is skipped, so one write in Paxos log. Each entry in Paxos log contains Paxos related info and raw block data. After quorum is reached, we update our state machine, i.e. both block data and our logical to physical block mapping. It is inefficient to store big block data multiple times in disk drive. But Paxos log needs to be persistent in order to make the system reliable and consistent. To make it I/O efficient we did a novel IO optimization strategy. Here instead of storing the whole block data couple of times, we store once in our local append-only file store and use that reference in the Paxos Log.

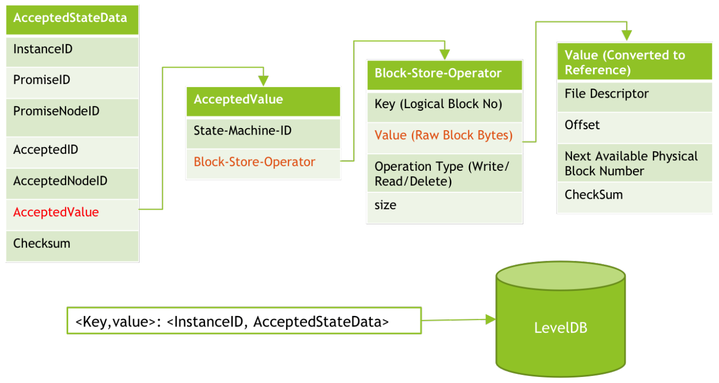


Figure 6.2: Write Optimization in Local Physical Volume

We intercept each read and write request to Paxos Log. If the request is for block store state machine and is a write request, we convert the raw value into a reference. For read request, we similarly convert the reference into value. We maintain a local counter for physical storage. We persist this counter with each paxos log entry to maintain consistency in case of failure. Here is the format of a Paxos Log Entry: *<Key,value>: <InstanceID, AcceptedStateData>*

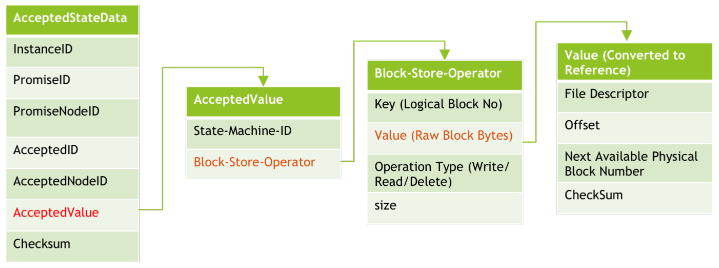


Figure 6.3: Format of a Paxos Log Entry

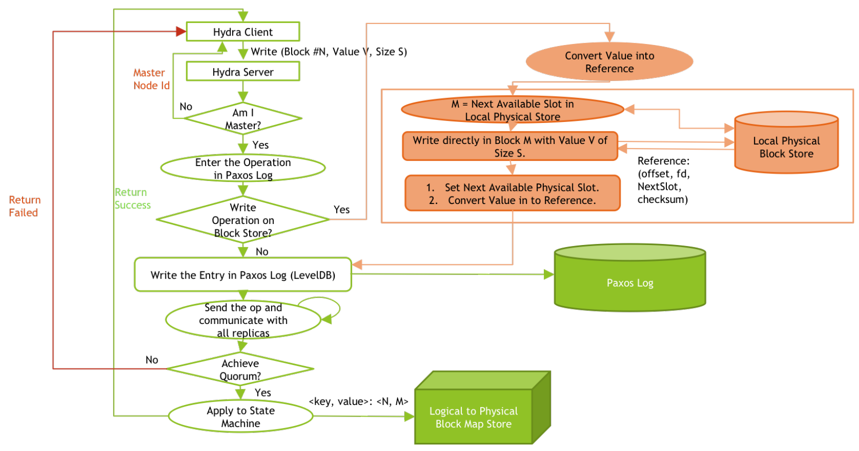

Hydra Client
Write (Block #N, Value V, Size S)
Hydra Server
Master Node Id
No
Am I Master?
Yes
Enter the Operation in Paxos Log
Return Failed
Return Success
Write Operation on Block Store?
Yes
No
Write the Entry in Paxos Log (LevelDB)
Send the op and communicate with all replicas
No
Achieve Quorum?
Yes
Apply to State Machine
<key, value>: <N, M>
Logical to Physical Block Map Store
Paxos Log
Convert Value into Reference
M = Next Available Slot in Local Physical Store
Write directly in Block M with Value V of Size S.
1. Set Next Available Physical Slot.
2. Convert Value in to Reference.
Reference: (offset, fd, NextSlot, checksum)
Local Physical Block Store

Figure 6.4: Write Operational Flow

Write operation Flow Each write operation (let's suppose a write operation on block N) follows the above flow chart. First Server checks whether its master, if it is not, it returns the master node id to redirect the request to master. Master first checks if it is write operation on Block store. If not, it stores it in paxos log without any optimization. If yes then it converts the raw block data into reference. It firsts checks with local physical store on what is the next available block number M in its append based store. Then we directly write our raw data into that block M. The data can span multiple blocks. The physical block store client will write the blocks into store and perform immediate fdatasync operation. Then it will return the following information : (staring offset, file descriptor, Next available physical block number, checksum of the data). We serialize it into a reference and change the paxos value in the paxos log from raw block data to reference. We also set the next physical slot accordingly. Note that by doing so we might have some garbage values in local physical store in some failure scenarios. But that will affect the overall throughput of the system. We also can do offline vacuuming to remove fragments. After writing the entry into the paxos log, master sends the operation to all replicas, they stores the entry in their own paxos log in similar fashion and tries to reach quorum. After that, we apply the write operation in Block Store State Machine by updating the logical to physical block map store. (i.e. Enter <N,M> as key value in kv store).

Read and Delete Operation Flow Here it is a two-level read. Read physical block number from logical block number in our KV Map Store. Compute offset from physical block number. Read desired number of bytes from offset in local file store. Delete operation just deletes the mapping from map store after reaching quorum between all paxos nodes. This is logical delete and that is why this is very fast.

Recovery If any node in Hydra fails, Paxos quorum can be reached if majority of the nodes are available. The recovery process starts once the node reboots by shipping Paxos Log and/or State machine image from a remote replica. When reading from Paxos Log, remote replica convert the Reference into Value before sending it to the recovering node.

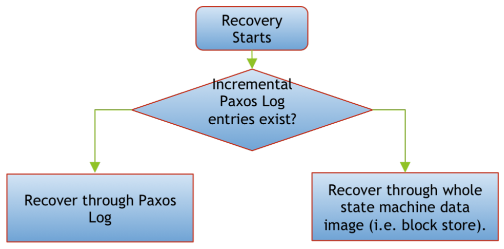

Recovery Starts
Incremental Paxos Log entries exist?
Recover through Paxos Log
Recover through whole state machine data image (i.e. block store).

Figure 6.5: Recovery for Block Store

Example. Let's illustrate an Example with the physical state of a block store after a sequence of operations: Write Block #5 (size: 4K) -> Write Block # 7 (size: 12K) -> Write Block # 1 (size: 6K) -> Delete Block # 8 -> Write Block # 1 (size: 4K) -> Write Block # 15 (size: 4K). Figure 6.6 how

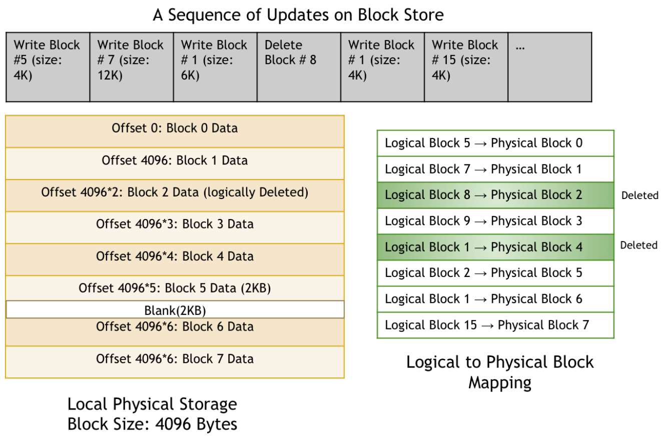

A Sequence of Updates on Block Store
Write Block #5 (size: 4K)
Write Block # 7 (size: 12K)
Write Block # 1 (size: 6K)
Delete Block # 8
Write Block # 1 (size: 4K)
Write Block # 15 (size: 4K)
...
Offset 0: Block 0 Data
Offset 4096: Block 1 Data
Offset 4096*2: Block 2 Data (logically Deleted)
Offset 4096*3: Block 3 Data
Offset 4096*4: Block 4 Data
Offset 4096*5: Block 5 Data (2KB)
Blank(2KB)
Offset 4096*6: Block 6 Data
Offset 4096*6: Block 7 Data
Local Physical Storage
Block Size: 4096 Bytes
Logical Block 5 → Physical Block 0
Logical Block 7 → Physical Block 1
Logical Block 8 → Physical Block 2
Deleted
Logical Block 9 → Physical Block 3
Logical Block 1 → Physical Block 4
Deleted
Logical Block 2 → Physical Block 5
Logical Block 1 → Physical Block 6
Logical Block 15 → Physical Block 7
Logical to Physical Block Mapping

Figure 6.6: The Physical State of a Block Store

local physical store is storing block by block raw data and how we keep the logical map store. ■

Chapter 7

Conclusions

In this thesis, we studied several problems to Optimize NoSQL big data storage based on Indexing, Partitioning and Replication.

In Chapter 3, we study how secondary indexes can be efficiently supported in NoSQL systems which is heavily used in big data applications.

In Chapter 4, we show how NoSQL storage can be designed and optimized to support efficient and high throughput Publish/Subscribe.

In Chapter 5, we designed a novel approach to extend existing LSM techniques to support efficient spatial querying.

Finally in Chapter 6, we show how a block store can be efficiently replicated so that a big data system can remotely store its data in cloud network without too much overhead.

Overall, my thesis shows how NoSQL systems can be extended and optimized to support more sophisticated big data applications Overall, I consider this thesis as a step forward from current NoSQL database to more sophisticated NewSQL big database systems.

Bibliography

[1] The architecture twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.

[2] Asterixdb. `https://asterixdb.ics.uci.edu/`.

[3] Couchdb. http://couchdb.apache.org/.

[4] Live commenting: Behind the scenes. https://code.facebook.com/posts/557771457592035/live-commenting-behind-the-scenes/.

[5] Storm. In *http://storm.incubator.apache.org/*, 2015.

[6] Parag Agrawal, Adam Silberstein, Brian F Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 179–192. ACM, 2009.

[7] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *Proceedings of the VLDB Endowment*, 7(10), 2014.

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer, 2004.

[9] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 15(2):121–142, 2006.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[11] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[12] 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 *CIDR*, volume 11, pages 223–234, 2011.

[13] Basho. Secondary indexes in riak, October 2017. `http://basho.com/posts/technical/secondary-indexes-in-riak`.

[14] Neerja Bhatt, Dieter Gawlick, Ekrem Soylemez, and Rahim Yaseem. Content based publish-and-subscribe system integrated in a relational database system, June 11 2002. US Patent 6,405,191.

[15] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[16] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, 2011.

[17] Michael J Carey, Steven Jacobs, and Vassilis J Tsotras. Breaking bad: a data serving vision for big active data. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 181–186. ACM, 2016.

[18] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[19] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[20] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[21] 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. *TOCS*, 26(2):4, 2008.

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

[23] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM, 2000.

[24] Xiaoying Chen, Chong Zhang, Bin Ge, and Weidong Xiao. Spatio-temporal queries in hbase. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1929–1937. IEEE, 2015.

[25] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[26] 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*, pages 143–154, 2010.

[27] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[28] Debraj De and Lifeng Sang. Qos supported efficient clustered query processing in large collaboration of heterogeneous sensor networks. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*, pages 242–249. IEEE, 2009.

[29] Andy Dent. *Getting Started with LevelDB*. Packt Publishing Ltd, 2013.

[30] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.

[31] Lars Ellenberg. Drbd 9 and device-mapper: Linux block level storage replication. In *Proceedings of the 15th International Linux System Technology Conference*, 2008.

[32] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[33] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[34] A Feinberg. Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[35] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *FIW*, pages 12–30, 2005.

[36] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[37] Xiaoming Gao, Mike Lowe, Yu Ma, and Marlon Pierce. Supporting cloud computing with the virtual block store system. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 71–78. IEEE, 2009.

[38] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[39] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[40] Lei Guo, Dejun Teng, Rubao Lee, Feng Chen, Siyuan Ma, and Xiaodong Zhang. Re-enabling high-speed caching for lsm-trees. *arXiv preprint arXiv:1606.02015*, 2016.

[41] Long Guo, Dongxiang Zhang, Guoliang Li, Kian-Lee Tan, and Zhifeng Bao. Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 843–857. ACM, 2015.

[42] Eric N Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, JB Park, and Albert Vernon. Scalable trigger processing. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 266–275. IEEE, 1999.

[43] Yuan He, Mo Li, and Yunhao Liu. Collaborative query processing among heterogeneous sensor networks. In *Proceedings of the 1st ACM International Workshop on Heterogeneous Sensor and Actor Networks*, HeterSanet '08, pages 25–30, New York, NY, USA, 2008. ACM.

[44] Abdeltawab M Hendawi, Jayant Gupta, Youying Shi, Hossam Fattah, and Mohamed Ali. The microsoft reactive framework meets the internet of moving things.

[45] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34. ACM, 2001.

[46] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial Informatics, Fusion, and Motion Video Analytics V*, volume 9473, page 94730F. International Society for Optics and Photonics, 2015.

[47] Aerospike inc. Aerospike secondary index architecture, October 2017. `https://www.aerospike.com/docs/architecture/secondary-index.html`.

[48] Amazon Inc. Aurora database, October 2017. https://aws.amazon.com/rds/aurora/.

[49] Amazon Inc. Dynamodb. `https://aws.amazon.com/dynamodb/`, Feb 2017.

[50] Amazon Inc. Global secondary indexes - amazon dynamodb, October 2017. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html`.

[51] Basho Inc. Riak kv. `http://basho.com/products/riak-kv/`, Feb 2017.

[52] Facebook Inc. Rocksdb. http://rocksdb.org/, Feb 2017.

[53] Facebook Inc. Strategies to reduce write amplification, October 2017. `https://github.com/facebook/rocksdb/issues/19`.

[54] Google Inc. Google snappy, October 2017. `http://google.github.io/snappy`.

[55] Google Inc. Leveldb. http://leveldb.org/, Feb 2017.

[56] IBM Inc. Ibm big data analytics, October 2017. `https://www.ibm.com/analytics/us/en/big-data/`.

[57] IBM inc. Understanding netezza zone maps, October 2017. https://www.ibm.com/developerworks/community/blogs/Wce085e09749a_4650_a064_bb3f3b738fa3/entry/understanding_netezza_zone_maps?lang=en.

[58] InfluxData Inc. Influxdb. https://www.influxdata.com/, Feb 2017.

[59] MongoDB Inc. Mongodb. https://www.mongodb.com, Feb 2017.

[60] Oracle Inc. Oracle bitmap indexes. https://docs.oracle.com/cd/B10500_01/server.920/a96520/indexes.htm, Feb 2017.

[61] Oracle Inc. Oracle: Using zone maps, October 2017. http://docs.oracle.com/database/121/DWHSG/zone_maps.htm.

[62] Snap Inc. How snaps are stored and deleted, October 2017. https://www.snap.com/en-US/news/post/how-snaps-are-stored-and-deleted.

[63] Tencent Inc. Phxpaxos, November 2017. https://github.com/Tencent/phxpaxos.

[64] Teradata Inc. Teradata teradata analytics for enterprise applications, October 2017. http://www.teradata.com/analyticssolutions.

[65] VMware Inc. Gemfire continuous querying. https://pubs.vmware.com/vfabric5/index.jsp?topic=/com.vmware.vfabric.gemfire.6.6/developing/continuous_querying/how_continuous_querying_works.html, Feb 2017.

[66] Steven Jacobs, Md Yusuf Sarwar Uddin, Michael Carey, Vagelis Hristidis, Vassilis J Tsotras, N Venkatasubramanian, Yao Wu, Syed Safir, Purvi Kaul, Xikui Wang, Mohiuddin Abdul Qader, and Yawei Li. A bad demonstration: towards big active data. *Proceedings of the VLDB Endowment*, 10(12):1941–1944, 2017.

[67] Hans-Arno Jacobsen, Vinod Muthusamy, and Guoli Li. The padres event processing network: Uniform querying of past and future eventsdas padres ereignisverarbeitungsnetzwerk: Einheitliche anfragen auf ereignisse der vergangenheit und zukunft. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 51(5):250–260, 2009.

[68] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 16(4):417–437, 2007.

[69] Rohit Kamboj and Anoopa Arya. Openstack: open source cloud computing iaas platform. *International Journal of Advanced Research in Computer Science and Software Engineering*, 4(5), 2014.

[70] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, and John Ousterhout. Slik: Scalable low-latency indexes for a key-value store.

[71] Bettina Kemme and Gustavo Alonso. Database replication: A tale of research across communities. *Proc. VLDB Endow.*, 3(1-2):5–12, September 2010.

[72] Anne-Marie Kermarrec and Peter Triantafillou. Xl peer-to-peer pub/sub systems. *ACM Computing Surveys (CSUR)*, 46(2):16, 2013.

[73] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, pages 22–28, 2006.

[74] kidsandteensonline. Where do pictures and files we send using whatsapp end up? `http://kidsandteensonline.com/2013/10/10/where-do-pictures-and-files-we-send-using-whatsapp-end-up`, Oct 2013.

[75] UC Riverside Database Lab. Project website for open source code., May 2017. http://dblab.cs.ucr.edu/projects/PubSub-Store/.

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

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

[78] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[79] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[80] Lucas Lersch, Ismail Oukid, Wolfgang Lehner, and Ivan Schreter. An analysis of lsm caching in nvram. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, page 9. ACM, 2017.

[81] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60. ACM, 2002.

[82] Mahdi Tayarani Najaran and Norman C Hutchinson. Innesto: A searchable key/value store for highly dimensional data. In *CloudCom*, pages 411–420. IEEE, 2013.

[83] Bazy Danych Nosql, W Aplikacjach Naukowych, OsÌĄwiadczenie Autora Pracy, and Karnej Za PosÌĄwiadczenie. Nosql databases in scientific applications. 2016.

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

[85] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[86] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 551–566. ACM, 2018.

[87] Mohiuddin Abdul Qader and Vagelis Hristidis. Dualdb: An efficient lsm-based publish/subscribe storage system. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 24. ACM, 2017.

[88] Mohiuddin Abdul Qader and Vagelis Hristidis. Dualdb: An efficient lsm-based publish/subscribe storage system. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2017.

[89] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 469–478. Morgan Kaufmann Publishers Inc., 1991.

[90] 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*, pages 217–228. ACM, 2012.

[91] Wei Tan, Sandeep Tata, Yuzhe Tang, and Liana Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.

[92] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215. ACM, 2002.

[93] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable xml publish/subscribe system using relational database systems. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2004.

[94] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX Annual Technical Conference*, pages 413–424, 2012.

[95] Sage A Weil. *Ceph: reliable, scalable, and high-performance distributed storage*, volume 68. 2007.

[96] Jennifer Widom and Sheldon J Finkelstein. Set-oriented production rules in relational database systems. In *ACM SIGMOD Record*, volume 19, pages 259–270. ACM, 1990.

[97] Jia Yu and Mohamed Sarwat. Two birds, one stone: a fast, yet lightweight, indexing scheme for modern database systems. *Proceedings of the VLDB Endowment*, 10(4):385–396, 2016.

[98] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. Paxosstore: high-availability storage made practical in wechat. *Proceedings of the VLDB Endowment*, 10(12):1730–1741, 2017.

The appendix is organized as follows. Appendix A provides pseudocode of LOOKUP (A_i, a, K) and RANGELOOKUP (A_i, a, b, K) operation in LevelDB implementation for different index variants. Appendix B presents supplementary experimental results, which analyze the effect of bloom filter, compression and concurrency on different secondary indexes. Appendix C describes effect of distributed environment on secondary indexes.

Appendix A

Algorithms

In all algorithms, we assume each $tuple(k,v)$ is associated with a sequence number (*seq*). We show how we add the records on Min-Heap *H* based on this *seq* in Algorithm 6. Algorithm 7, 8, 9, 10 present the pseudocodes of LOOKUP(A_i, a, K) in Stand-Alone Eager, Lazy, and Composite Index and Embedded Index respectively. Algorithm 11, 12,13 present the pseudocodes of RANGELOOKUP(A_i, a, b, K) in Stand-Alone Lazy, and Composite and Embedded Index respectively. Stand-Alone Eager Index uses existing range query API for primary key on the Stand-Alone Index Table in LevelDB.

Algorithm 6 Min-Heap $H.Add(K, \langle k, v\rangle)$ Procedure

```
1: if $H.size() == K \wedge H.top.seq < \langle k,v\rangle.seq$ then
2:     $H.pop()$;
3:     $H.put(\langle k,v\rangle)$;
4: else if $H.size() < K$ then
5:     $H.put(\langle k,v\rangle)$;
6: end if
```

Algorithm 7 LOOKUP Procedure using Eager Index

```
1: Create Min-Heap H;
2: Primary key list $L \leftarrow$ return of GET(a) on index table $T_i$. {Suppose the order is maintained as
   recent key to older keys}
3: for $k$ in $L$ do
4:     $\langle k,v\rangle \leftarrow$ return of GET($k$) on data table.
5:     if $v.val(A_i) == a$ then
6:         H.add($K$, $\langle k,v\rangle$) {Algorithm 6}
7:     end if
8:     if $H.size() == K$ then
9:         BREAK
10:    end if
11: end for
        List of pairs in $H$
```

Algorithm 8 LOOKUP Procedure using Lazy Index

```
1:  Create Min-Heap H;
2:  {starts from MemTable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3:  for $j$ from 0 to $L$ do
4:      if $val(A_i)$ is not in $Cj$ then
5:          NEXT
6:      end if
7:      List of primary keys $P \leftarrow val(A_i)$ in $Cj$
8:      for $k$ in $P$ do
9:          $\langle k, v \rangle \leftarrow$ return of GET($k$) on data table.
10:         if $v.val(A_i) == a$ then
11:             H.add($K$, $\langle k, v \rangle$) {Algorithm 6}
12:         end if
13:         if $H.size() == K$ then $H$
14:         end if
15:     end for
16: end forList of pairs in $H$
```

Algorithm 9 LOOKUP Procedure using Composite Index

```
1: Create Min-Heap H;
2: {starts from MemTable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3: for $j$ from 0 to $L$ do
4:     $it \leftarrow SSTable :: Iterator_j.SEEK$(a)
5:     while $it.valid()$ do
6:         Composite key, $ckey \leftarrow it.key()$
7:         $skey \leftarrow ckey.ExtractSecondarykey()$
8:         if $a == skey$ then
9:             $k \leftarrow ckey.ExtractPrimaryKey()$
10:            $\langle k, v\rangle \leftarrow$ return of GET($k$) on data table.
11:            if $v.val(A_i) == a$ then
12:                H.add($K$, $\langle k, v\rangle$) {Algorithm 6}
13:            end if
14:        else if $skey > a$ then
15:            BREAK
16:        end if
17:        $it.NEXT()$
18:    end while
19: end forList of pairs in $H$
```

Algorithm 10 LOOKUP Procedure using Embedded Index

```
1: function GetLite ($key, level$)
2: for $l = 0 \rightarrow L$ do
3:    if $l.SEEK(key) == True$ then
4:       if $l = level$ then True {SEEK operation Checks only in-memory metadata, index block
   and bloom filters for primary key to lookup key}
5:       elseFalse
6:       end if
7:    end if
8: end for
1: function LOOKUP ($A_i$, $a$, $K$)
2: Create Min-Heap H;
3: for table in MemTable do
4:    for $\langle k, v\rangle$ in table do
5:       if $val(A_i) == a$ then
6:          H.add($K$, $\langle k, v\rangle$) {Algorithm 6}
7:       end if
8:    end for
9: end for
```

```
if $H.size() == K$ then List of $K$ pairs in H
end if
for $l = 0 \rightarrow L$ do
    for $sstable_j$ in level-$l$ do
        if $sstable_j.globalzonemap(A_i).contains(a) == True$ then
            for block $b_m$ in $sstable_j$ do
                if $b_m.bloomfilter(A_i).contains(a) == False$ $b_m.zonemap(A_i).contains(a) == False$ then
                    NEXT
                end if
                load $b_m$ of $sstable_j$ in memory;
                for $\langle k, v\rangle$ in $b_m$ do
                    if $v.val(A_i).equals(a) \wedge GetLite(k, l) == True$ then
                        H.add($K$, $\langle k, v\rangle$) {Algorithm 6}
                    end if
                end for
            end for
        end if
    end for
    if $H.size() == K$ then List of $K$ pairs in $H$
    end if
end forList of pairs in $H$
```

Algorithm 11 RANGELOOKUP Procedure using Lazy Index

```
1:  Create Min-Heap H;
2:  {starts from MemTable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3:  for $j$ from 0 to $L$ do
4:      $it \leftarrow SSTable :: Iterator_j.SEEK$(a)
5:      while $it.valid()$ do
6:          if $it.key()$ in $[a,b]$ then
7:              List of primary keys $P \leftarrow it.value()$
8:              for $k$ in $P$ do
9:                  $\langle k,v \rangle \leftarrow$ return of GET($k$) on data table.
10:                 if $v.val\,(A_i) == it.key()$ then
11:                     H.add($K$, $\langle k,v \rangle$) {Algorithm 6}
12:                 end if
13:             end for
14:         else
15:             BREAK
16:         end if
17:         $it.NEXT()$
18:     end while
19: end forList of pairs in $H$
```

Algorithm 12 RANGELOOKUP Procedure using Composite Index

```
1: Create Min-Heap H;
2: {starts from MemTable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.}
3: for $j$ from 0 to $L$ do
4:     $it \leftarrow SSTable :: Iterator_j.SEEK$(a)
5:     while $it.valid()$ do
6:         Composite key, $ckey \leftarrow it.key()$
7:         $skey \leftarrow ckey.ExtractSecondarykey()$
8:         if $skey$ in $[a,b]$ then
9:             Repeat $\leftarrow$ Line 8-12 from Algorithm 9
10:        else if $skey > b$ then
11:            BREAK
12:        end if
13:        $it.NEXT()$
14:    end while
15: end forList of pairs in $H$
```

Algorithm 13 RANGELOOKUP Procedure using Embedded Index

```
1: Create Min-Heap H;
2: for table in MemTable do
3:     for $\langle k, v \rangle$ in table do
4:         if $v.val(A_i)$ in $[a, b]$ then
5:             H.add($K$, $\langle k, v \rangle$) {Algorithm 6}
6:         end if
7:     end for
8: end for
9: if $H.size() == K$ then List of $K$ pairs in H
10: end if{REPEAT LINE 9 to 20 from Algorithm 10 to populate H, but replace $contains(a)$ function with $intersects(a, b)$ and $equals(a)$ with $within(a, b)$} List of pairs in $H$
```

Appendix B

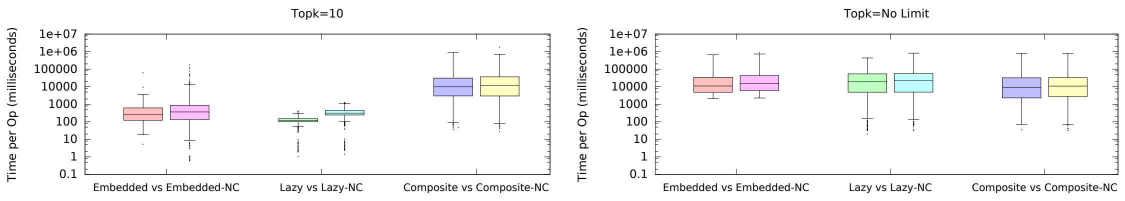
Supplementary Experiments

B.1 Effects of varying bloom filter lengths

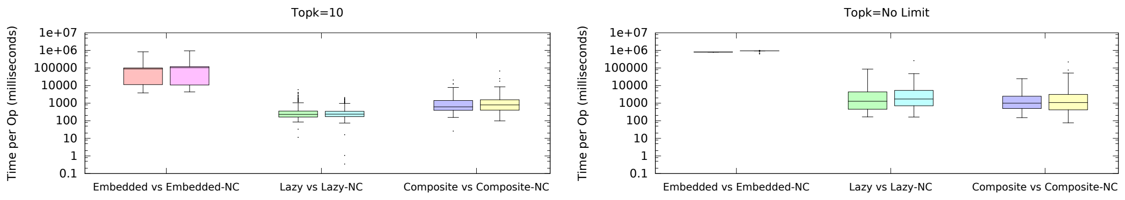
The length of each bloom filter used in Embedded Index also has effect on LOOKUP performance in two folds: (1) with larger bloom filter, the false positive rate will drop thus the number of blocks to be loaded into memory is smaller, and (2) the cost of checking each bloom filter is higher because more hash functions are applied for each key to check its existence as the bloom filter length increases. We conducted the experiment by varying the value of *bits per key* from 20 to 1000 with our seed twitter dataset to determine the optimal choice of bloom filter length for this dataset. We write 20 million tweets with Embedded Index on UserID and performed 200K LOOKUPs. Table B.1 shows the average performance of LOOKUPs. Here we see that the LOOKUP performance starts to increase with increasing bits per key setting of bloom filter (20 to 200) as the false positive rate decreases. But then the performance decreases with larger value of bits per key where the false positive rate is low enough, but the cost of checking bloom filters increases.

Table B.1: How LOOKUP performance, average false positive rates and database size vary with bloom filter length

Bits Per Key	**DB Size (MB)**	**False Positive %**	**Mean LOOKUP Latency (Milliseconds)**
20	10,127	0.64%	52
50	10,256	0.44%	49
100	10,470	0.08%	21
200	10,899	0.08%	20.5
500	12,186	0.08%	22
1000	14,325	0.08%	32



(a) LOOKUP Response Time



(b) RANGELOOKUP Response Time (Selectivity: 10)

Figure B.1: UserID Index performance for different variants with and without compression. (NC = Not Compressed)

We also show the space trade-offs in Table B.1. As the bits per key increases, the bloom filter takes more space. In our experiments, we set *bits per key* as 100, because the LOOKUP performance is very close to setting it as 200 to 300 but with reasonable space overhead.

B.2 Effect of compression on indexes

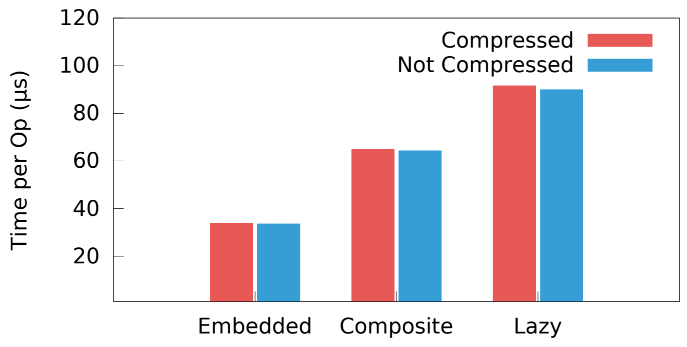
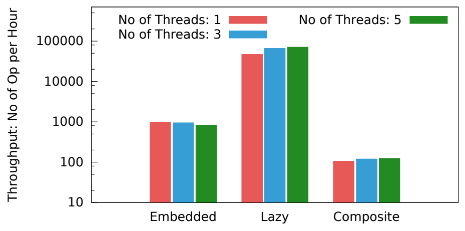


Figure B.2: Mean Time per PUT operation on UserID index

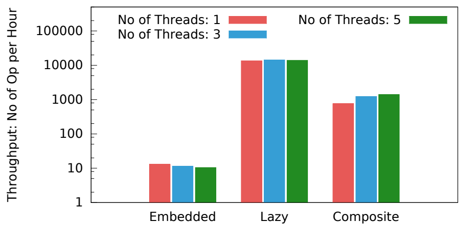
In this section, we report how block compression affects secondary index's performance. Keeping same experimental settings and same dataset from Section 4.7, we rebuild our secondary indexes for UserID index without compression and perform top-*K* LOOKUP and RANGE LOOKUP queries. Figures B.1a and B.1b show varying top-*K* LOOKUP and RANGELOOKUP query response times of different secondary indexes, for default Snappy compression [54] enabled against no compression. We see the same relationship exists between different indexing variants with or without compression. But LOOKUP and RANGELOOKUP response times are slightly higher for all of them if there is no compression enabled. Figure B.2 shows that we also get almost identical results for PUT performance for all secondary indexes with and without compression.

B.3 Effect of multi-threaded environment

In this section, we report our secondary index's performance in multi-threaded environment. We repeat LOOKUP and RANGE LOOKUP queries from Section 4.7 on UserID index and



(a) LOOKUP Throughput



(b) RANGELOOKUP Throughput

Figure B.3: UserID Index performance for multi-threads

measure throughput (i.e. number of operation per hour) varying number of concurrent threads. Top-K is set as 10. Figure B.3 shows that secondary indexes show same relationship between them for multi-threaded environment compare to single thread. Stand-Alone Indexes has slightly higher throughput for high concurrencies over Embedded Index because different threads can efficiently access data and index table in parallel. Although, we see that increasing the number of threads does not necessarily increase throughout much (even decrease throughput in some cases) as different threads compete with each other for system resources (e.g. database components) and eventually decrease system throughput.

Appendix C

Effect of Distributed environment

The main overhead of a distributed NoSQL environment is the cost of maintaining the replication consistency [71]. In this paper, we focus on the storage engine module, which can be conceptually viewed as a state machine maintained by any consensus protocol-based (e.g. Paxos, Raft) replication engine (e.g. PaxosStore [98]). The overhead of the replication engine generally does not depend on the choice of the implementation of the state machine (i.e. implementation of secondary indexes). The other aspects of distributed environment (i.e. partitioning module, cluster membership etc.) can also be designed independent to the storage engine module [76] [7] [13].