Learning How To Learn Within An LSM-based Key-Value Store

Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth[†],
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison [†] *Microsoft Gray Systems Lab*

Abstract. We introduce BOURBON, a log-structured merge (LSM) tree that utilizes machine learning to provide fast lookups. We base the design and implementation of BOURBON on empirically-grounded principles that we derive through careful analysis of LSM design. BOURBON employs greedy piecewise linear regression to learn key distributions, enabling fast lookup with minimal computation, and applies a cost-benefit strategy to decide when learning will be worthwhile. Through a series of experiments on both synthetic and real-world datasets, we show that BOURBON improves lookup performance by 1.23×-1.78× as compared to state-of-the-art production LSMs.

1 Introduction

Machine learning is transforming how we build computer applications and systems. Instead of writing code in the traditional algorithmic mindset, one can instead collect the proper data, train a model, and thus implement a robust and general solution to the task at hand. This data-driven, empirical approach has been called "Software 2.0" [24], hinting at a world where an increasing amount of the code we deploy is realized in this manner; a number of landmark successes over the past decade lend credence to this argument, in areas such as image [30] and speech recognition [22], machine translation [43], game playing [41], and many other areas [6,14,16].

One promising line of work, for using ML to improve core systems is that of the "learned index" [29]. This approach applies machine learning to supplant the traditional index structure found in database systems, namely the ubiquitous B-Tree [8]. To look up a key, the system uses a learned function that predicts the location of the key (and value); when successful, this approach can improve lookup performance, in some cases significantly, and also potentially reduce space overhead. Since this pioneering work, numerous follow ups [12, 18, 28] have been proposed that use better models, better tree structures, and generally improve how learning can reduce tree-based access times and overheads.

However, one critical approach has not yet been transformed in this "learned" manner: the Log-structured Merge Tree (LSM) [35, 37, 39]. LSMs were introduced in the late '90's, gained popularity a decade later through work at Google on BigTable [7] and LevelDB [20], and have become widely used in industry, including in Cassandra [31], RocksDB [17], and many other systems [19,36]. LSMs have many positive properties as compared to B-trees and their cousins, including high insert performance [10,35,38].

In this paper, we apply the idea of the learned index to LSMs. A major challenge is that while learned indexes are primarily tailored for read-only settings, LSMs are optimized for writes. Writes cause disruption to learned indexes because models learned over existing data must now be updated to accommodate the changes; the system thus must re-learn the data repeatedly. However, we find that LSMs are quite well-suited for using learning to speedup lookups while supporting writes. For example, although writes modify the LSM, most portions of the tree are immutable; thus, learning a function to predict key/value locations can be done once, and used as long as the immutable data lives. However, many challenges arise. For example, variable key or value sizes make learning a function to predict locations more difficult, and performing model building too soon may lead to significant resource waste.

Thus, we first study how an existing LSM system, WiscKey [35], functions in great detail (§3). We focus on WiscKey because it is a state-of-the-art LSM implementation that is significantly faster than LevelDB and RocksDB [35]. Our analysis leads to many interesting insights from which we develop five *learning guidelines*: a set of rules that aid an LSM system to successfully incorporate learned indexes. For example, while it is useful to learn the stable, low levels in an LSM, learning higher levels can yield benefits as well because lookups must always search the higher levels. Next, not all files are equal: some files even in the lower levels are very short-lived; a system must avoid learning such files, or resources can be wasted. Finally, workload- and data-awareness is important; based on the workload and how the data is loaded, it may be more beneficial to learn some portions of the tree than others.

We apply these learning guidelines to build BOURBON, a learned-index implementation of WiscKey (§4). BOURBON uses piece-wise linear regression, a simple but effective model that enables both fast training (i.e., learning) and inference (i.e., lookups) with little space overhead. BOURBON employs *file learning*: models are built over files given that an LSM file, once created, is never modified in-place. BOURBON implements a cost-benefit analyzer that dynamically decides whether or not to learn a file, reducing unnecessary learning while maximizing benefits. While most of the prior work on learned indexes [12, 18, 29] has taken strides in optimizing stand-alone data structures, BOURBON integrates learning into a production-quality system that is already highly optimized.

We analyze the performance of Bourbon on a range of synthetic and real-world datasets and workloads (§5). We find that Bourbon reduces the indexing costs of WiscKey significantly and thus offers 1.23× – 1.78× faster lookups for various datasets. Even under workloads with significant write load, Bourbon speeds up a large fraction of lookups and, through cost-benefit, avoids unnecessary (early) model building. Thus, Bourbon matches the performance of an aggressive-learning approach but performs model building more judiciously. Finally, most of our analysis focuses on the case where fast lookups will make the most difference, namely when the data resides in memory (i.e., in the file-system page cache). However, we also experiment with Bourbon when data resides on a fast storage device (an Optane SSD), and show that benefits can still be realized.

This paper makes four contributions. We present the first detailed study of how LSMs function internally with learning in mind. We formulate a set of guidelines on how to integrate learned indexes into an LSM (§3). Third, we present the design and implementation of Bourbon which incorporates learned indexes into a real, highly optimized, production-quality LSM system (§4). Finally, we analyze Bourbon's performance in detail, and demonstrate its benefits (§5).

2 Background

We first describe log-structured merge trees and explain how data is organized in LevelDB. Next, we describe WiscKey, a modified version of LevelDB that we adopt as our baseline. We then provide a brief background on learned indexes.

2.1 LSM and LevelDB

An LSM tree is a persistent data structure used in key-value stores to support efficient inserts and updates [37]. Unlike B-trees that require many random writes to storage upon updates, LSM trees perform writes sequentially, thus achieving high write throughput [37].

An LSM organizes data in multiple *levels*, with the size of each level increasing exponentially. Inserts are initially buffered in an in-memory structure; once full, this structure is merged with the first level of on-disk data. This procedure resembles merge-sort and is referred to as compaction. Data from an on-disk level is also merged with the successive level if the size of the level exceeds a limit. Note that updates do not modify existing records in-place; they follow the same path as inserts. As a result, many versions of the same item can be present in the tree at a time. Throughout this paper, we refer to the levels that contain the newer data as *higher* levels and the older data as *lower* levels.

A lookup request must return the latest version of an item. Because higher levels contain the newer versions, the search starts at the topmost level. First, the key is searched for in the in-memory structure; if not found, it is searched for in the on-disk tree starting from the highest level to the lowest one. The value is returned once the key is found at a level.

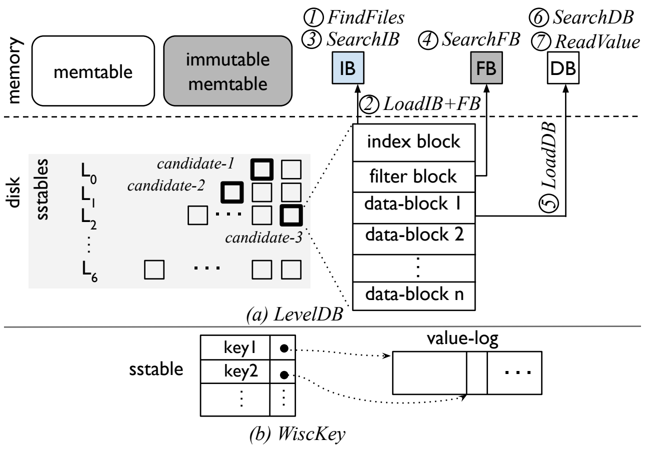


Figure 1: **LevelDB and WiscKey.** *(a) shows how data is organized in LevelDB and how a lookup is processed. The search in in-memory tables is not shown. The candidate sstables are shown in bold boxes. (b) shows WiscKey's key-value separation.*

LevelDB [20] is a widely used key-value store built using LSM. Figure 1(a) shows how data is organized in LevelDB. A new key-value pair is first written to the *memtable*; when full, the memtable is converted into an immutable table which is then compacted and written to disk sequentially as *sstables*. The sstables are organized in seven levels (L_0 being the highest level and L_6 the lowest). LevelDB ensures that key ranges of different sstables at a level are disjoint (two files will not contain overlapping ranges of keys); L_0 is an exception where the ranges can overlap across files. The amount of data at each level increases by a factor of ten; for example, the size of L_1 is 10MB, while L_6 contains several 100s of GBs. If a level exceeds its size limit, one or more sstables from that level are merged with the next level; this is repeated until all levels are within their limits.

Lookup steps. Figure 1(a) also shows how a lookup request for key k proceeds. ① *FindFiles*: If the key is not found in the in-memory tables, LevelDB finds the set of candidate sstable files that may contain k. A key in the worst case may be present in all L_0 files (because of overlapping ranges) and within one file at each successive level. ② *LoadIB+FB*: In each candidate sstable, an index block and a bloom-filter block are first loaded from the disk. ③ *SearchIB*: The index block is binary searched to find the data block that may contain k. ④ *SearchFB*: The filter is queried to check if k is present in the data block. ⑤ *LoadDB*: If the filter indicates presence, the data block is loaded. ⑥ *SearchDB*: The data block is binary searched. ⑦ *ReadValue*: If the key is found in the data block, the associated value is read and the lookup ends. If the filter indicates absence or if the key is not found in the data block, the search continues to the next candidate file. Note that blocks are not always loaded from the disk; index and filter blocks, and frequently accessed data blocks are likely to be present in memory (i.e., file-system cache).

We refer to these search steps at a level that occur as part of a single lookup as an *internal lookup*. A single lookup thus consists of many internal lookups. A *negative internal lookup* does not find the key, while a *positive internal lookup* finds the key and is thus the last step of a lookup request.

We differentiate indexing steps from data-access steps; indexing steps such as *FindFiles*, *SearchIB*, *SearchFB*, and *SearchDB* search through the files and blocks to find the desired key, while data-access steps such as *LoadIB+FB*, *LoadDB*, and *ReadValue* read the data from storage. Our goal is to reduce the time spent in indexing.

2.2 WiscKey

In LevelDB, compaction results in large write amplification because *both* keys and values are sorted and rewritten. Thus, LevelDB suffers from high compaction overheads, affecting foreground workloads.

WiscKey [35] (and Badger [1]) reduces this overhead by storing the values separately; the sstables contain only keys and pointers to the values as shown in Figure 1(b). With this design, compaction sorts and writes only the keys, leaving the values undisturbed, thus reducing I/O amplification and overheads. WiscKey thus performs significantly better than other optimized LSM implementations such as LevelDB and RocksDB. Given these benefits, we adopt WiscKey as the baseline for our design. Further, WiscKey's key-value separation enables our design to handle variable-size records; we describe how in more detail in §4.2.

The write path of WiscKey is similar to that of LevelDB except that values are written to a *value log*. A lookup in WiscKey also involves searching at many levels and a final read into the log once the target key is found. The size of WiscKey's LSM tree is much smaller than LevelDB because it does not contain the values; hence, it can be entirely cached in memory [35]. Thus, a lookup request involves multiple searches in the in-memory tree, and the *ReadValue* step performs one final read to retrieve the value.

2.3 Optimizing Lookups in LSMs

Performing a lookup in LevelDB and WiscKey requires searching at multiple levels. Further, within each sstable, many blocks are searched to find the target key. Given that LSMs form the basis of many embedded key-value stores (e.g., LevelDB, RocksDB [17]) and distributed storage systems (e.g., BigTable [7], Riak [36]), optimizing lookups in LSMs can have huge benefits.

A recent body of work, starting with learned indexes [29], makes a case for replacing or augmenting traditional index structures with machine-learning models. The key idea is to train a model (such as linear regression or neural nets) on the input so that the model can predict the position of a record in the sorted dataset. The model can have inaccuracies, and thus the prediction has an associated error bound. During lookups, if the model-predicted position of the key is correct, the record is returned; if it is wrong, a local search is performed within the error bound. For example, if the predicted position is *pos* and the minimum and maximum error bounds are δ_{min} and δ_{max}, then upon a wrong prediction, a local search is performed between $pos - \delta_{min}$ and $pos + \delta_{max}$.

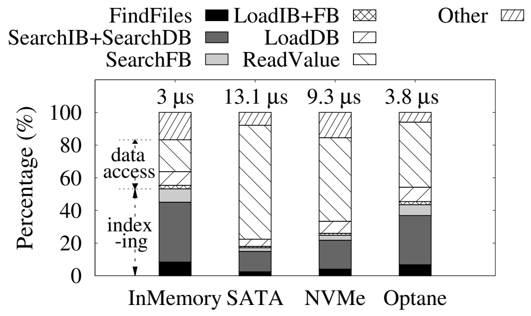


Figure 2: **Lookup Latency Breakdown.** *The figure shows the breakdown of lookup latency in WiscKey. The first bar shows the case when data is cached in memory. The other three bars show the case where the dataset is stored on different types of SSDs. We perform 10M random lookups on the Amazon Reviews dataset [5]; the figure shows the breakdown of the average latency (shown at the top of each bar). The indexing portions are shown in solid colors; data access and other portions are shown in patterns.*

Learned indexes can make lookups significantly faster. Intuitively, a learned index turns a $O(log\text{-}n)$ lookup of a B-tree into a $O(1)$ operation. Empirically, learned indexes provide $1.5\times - 3\times$ faster lookups than B-trees [29]. Given these benefits, we ask the following questions: *can learned indexes for LSMs make lookups faster? If yes, under what scenarios?*

Traditional learned indexes do not support updates because models learned over the existing data would change with modifications [12, 18, 29]. However, LSMs are attractive for their high performance in write-intensive workloads because they perform writes only sequentially. Thus, we examine: *how to realize the benefits of learned indexes while supporting writes for which LSMs are optimized?* We answer these two questions next.

3 Learned Indexes: a Good Match for LSMs?

In this section, we first analyze if learned indexes could be beneficial for LSMs and examine under what scenarios they can improve lookup performance. We then provide our intuition as to why learned indexes might be appropriate for LSMs even when allowing writes. We conduct an in-depth study based on measurements of how WiscKey functions internally under different workloads to validate our intuition. From our analysis, we derive a set of learning guidelines.

3.1 Learned Indexes: Beneficial Regimes

A lookup in LSM involves several indexing and data-access steps. Optimized indexes such as learned indexes can reduce the overheads of indexing but cannot reduce data-access costs. In WiscKey, learned indexes can thus potentially reduce the costs of indexing steps such as *FindFiles*, *SearchIB*, and *SearchDB*, while data-access costs (e.g., *ReadValue*) cannot be significantly reduced. As a result, learned indexes can improve overall lookup performance if indexing contributes to a sizable portion of the total lookup latency. We identify scenarios where this is the case.

First, when the dataset or a portion of it is cached in memory, data-access costs are low, and so indexing costs become significant. Figure 2 shows the breakdown of lookup latencies in WiscKey. The first bar shows the case when the dataset is cached in memory; the second bar shows the case where the data is stored on a flash-based SATA SSD. With caching, data-access and indexing costs contribute almost equally to the latency. Thus, optimizing the indexing portion can reduce lookup latencies by about 2×. When the dataset is not cached, data-access costs dominate and thus optimizing indexes may yield smaller benefits (about 20%).

However, learned indexes are not limited to scenarios where data is cached in memory. They offer benefit on fast storage devices that are currently prevalent and can do more so on emerging faster devices. The last three bars in Figure 2 show the breakdown for three kinds of devices: flash-based SSDs over SATA and NVMe, and an Optane SSD. As the device gets faster, lookup latency (as shown at the top) decreases, but the fraction of time spent on indexing increases. For example, with SATA SSDs, indexing takes about 17% of the total time; in contrast, with Optane SSDs, indexing takes 44% and thus optimizing it with learned indexes can potentially improve performance by 1.8×. More importantly, the trend in storage performance favors the use of learned indexes. With storage performance increasing rapidly and emerging technologies like 3D Xpoint memory providing very low access latencies, indexing costs will dominate and thus learned indexes will yield increasing benefits.

Summary. Learned indexes could be beneficial when the database or a portion of it is cached in memory. With fast storage devices, regardless of caching, indexing contributes to a significant fraction of the lookup time; thus, learned indexes can prove useful in such cases. With storage devices getting faster, learned indexes will be even more beneficial.

3.2 Learned Indexes with Writes

Learned indexes provide higher lookup performance compared to traditional indexes for read-only analytical workloads. However, a major drawback of learned indexes (as described in [29]) is that they do not support modifications such as inserts and updates [12, 18]. The main problem with modifications is that they alter the data distribution and so the models must be re-learned; for write-heavy workloads, models must be rebuilt often, incurring high overheads.

At first, it may seem like learned indexes are not a good match for write-heavy situations for which LSMs are optimized. However, we observe that the design of LSMs fits well with learned indexes. Our key realization is that although updates can change portions of the LSM tree, a large part remains immutable. Specifically, newly modified items are buffered in the in-memory structures or present in the higher levels of the tree, while stable data resides at the lower levels. Given that a large fraction of the dataset resides in the stable, lower levels, lookups to this fraction can be made faster with no or few re-learnings. In contrast, learning in higher levels may be less beneficial: they change at a faster rate and thus must be re-learned often.

We also realize that the immutable nature of sstable files makes them an ideal unit for learning. Once learned, these files are never updated and thus a model can be useful until the file is replaced. Further, the data within an sstable is sorted; such sorted data can be learned using simple models. A level, which is a collection of many immutable files, can also be learned as a whole using simple models. The data in a level is also sorted: the individual sstables are sorted, and there are no overlapping key ranges across sstables.

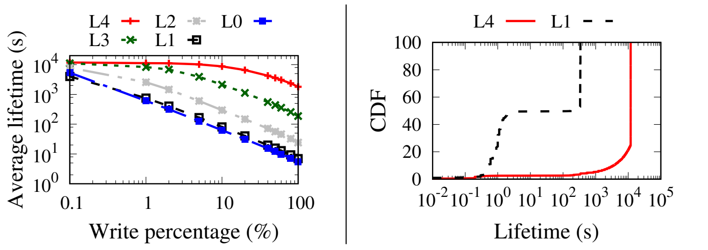
We next conduct a series of in-depth measurements to validate our intuitions. Our experiments confirm that while a part of our intuition is indeed true, there are some subtleties (for example, in learning files at higher levels). Based on these experimental results, we formulate a set of *learning guidelines*: a few simple rules that an LSM that applies learned indexes should follow.

Experiments: goal and setup. The goal of our experiments is to determine how long a model will be useful and how often it will be useful. A model built for a sstable file is useful as long as the file exists; thus, we first measure and analyze sstable lifetimes. How often a model will be used is determined by how many internal lookups it serves; thus, we next measure the number of internal lookups to each file. Since models can also be built for entire levels, we finally measure level lifetimes as well. To perform our analysis, we run workloads with varying amounts of writes and reads, and measure the lifetimes and number of lookups. We conduct our experiments on WiscKey, but we believe our results are applicable to most LSM implementations. We first load the database with 256M key-value pairs. We then run a workload with a single rate-limited client that performs 200M operations, a fraction of which are writes. Our workload chooses keys uniformly at random.

Lifetime of SSTables. To determine how long a model will be useful, we first measure and analyze the lifetimes of sstables. To do so, we track the creation and deletion times of all sstables. For files created during the load phase, we assign the workload-start time as their creation time; for other files, we record the actual creation times. If the file is deleted during the workload, then we calculate its exact lifetime. However, some files are not deleted by the end of the workload and we must estimate their lifetimes.[†]

Figure 3(a) shows the average lifetime of sstable files at different levels. We make three main observations. First, the average lifetime of sstable files at lower levels is greater than that of higher levels. Second, at lower percentages of writes,

[†]If the files are created during load, we assign the workload duration as their lifetimes. If not, we estimate the lifetime of a file based on its creation time (c) and the total workload time (w); the lifetime of the file is at least $w - c$. We thus consider the lifetime distribution of other files that have a lifetime of at least $w - c$. We then pick a random lifetime in this distribution and assign it as this file's lifetime.


(a) Average lifetimes with varying write %

(b) Lifetime distribution with 5% writes

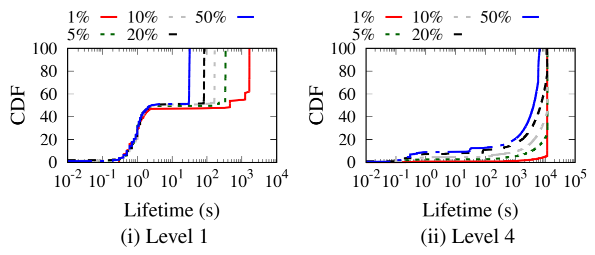
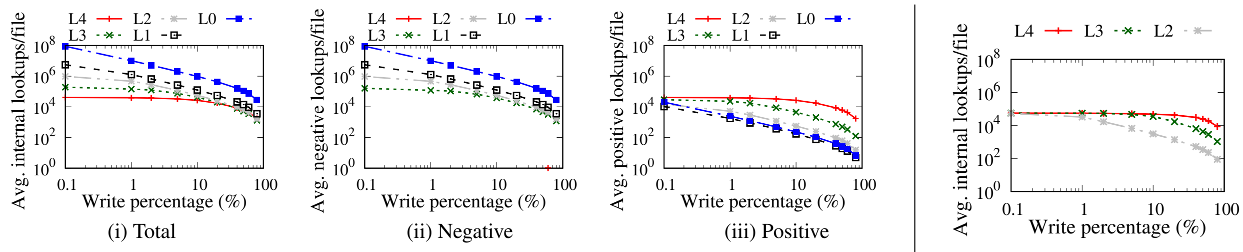

(c) Lifetime distributions with varying write %

Figure 3: **SSTable Lifetimes.** *(a) shows the average lifetime of sstable files in levels L_4 to L_0. (b) shows the distribution of lifetimes of sstables in L_1 and L_4 with 5% writes. (c) shows the distribution of lifetimes of sstables for different write percentages in L_1 and L_4.*


(a) Randomly loaded dataset

(b) Sequentially loaded dataset

Figure 4: **Number of Internal Lookups Per File.** *(a)(i) shows the average internal lookups per file at each level for a randomly loaded dataset. (b) shows the same for sequentially loaded dataset. (a)(ii) and (a)(iii) show the negative and positive internal lookups for the randomly loaded case.*

even files at higher levels have a considerable lifetime; for example, at 5% writes, files at L_0 live for about 2 minutes on an average. Files at lower levels live much longer; files at L_4 live about 150 minutes. Third, although the average lifetime of files reduces with more writes, even with a high amount of writes, files at lower levels live for a long period. For instance, with 50% writes, files at L_4 live for about 60 minutes. In contrast, files at higher level live only for a few seconds; for example, an L_0 file lives only about 10 seconds.

We now take a closer look at the lifetime distribution. Figure 3(b) shows the distributions for L_1 and L_4 files with 5% writes. We first note that some files are very short-lived, while some are long-lived. For example, in L_1, the lifetime of about 50% of the files is only about 2.5 seconds. If files cross this threshold, they tend to live for much longer times; almost all of the remaining L_1 files live over five minutes.

Surprisingly, even at L_4, which has a higher average lifetime for files, a few files are very short-lived. We observe that about 2% of L_4 files live less than a second. We find that there are two reasons why a few files live for a very short time. First, compaction of a L_i file creates a new file in L_{i+1} which is again immediately chosen for compaction to the next level. Second, compaction of a L_i file creates a new file in L_{i+1}, which has overlapping key ranges with the next file that is being compacted from L_i. Figure 3(c) shows that this pattern holds for other percentages of writes too. We observed that this holds for other levels as well. From the above observations, we arrive at our first two learning guidelines.

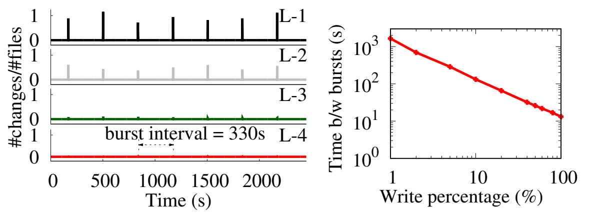
Learning guideline - 1: Favor learning files at lower levels. Files at lower levels live for a long period even for high write percentages; thus, models for these files can be used for a long time and need not be rebuilt often.

Learning guideline - 2: Wait before learning a file. A few files are very short-lived, even at lower levels. Thus, learning must be invoked only after a file has lived up to a threshold lifetime after which it is highly likely to live for a long time.

Internal Lookups at Different Levels. To determine how many times a model will be used, we analyze the number of lookups served by the sstable files. We run a workload and measure the number of lookups served by files at each level and plot the average number of lookups per file at each level. Figure 4(a) shows the result when the dataset is loaded in an uniform random order. The number of internal lookups is higher for higher levels, although a large fraction of data resides at lower levels. This is because, at higher levels, many internal lookups are negative, as shown in Figure 4(a)(ii). The number of positive internal lookups is as expected: higher in lower levels as shown in Figure 4(a)(iii). This result shows that files at higher levels serve many negative lookups and thus are worth optimizing. While bloom filters may already make these negative lookups faster, the index block still needs to be searched (before the filter query).

Figure 4(b) shows the result when the dataset is sequentially loaded, i.e., keys are inserted in ascending order. In contrast to the randomly-loaded case, there are no negative lookups because keys of different sstable files do not overlap even across levels; the *FindFiles* step finds the one file that may contain the key. Thus, lower levels serve more lookups and can have more benefits from learning. From these observations, we arrive at the next two learning guidelines.

Learning guideline - 3: Do not neglect files at higher levels. Although files at lower levels live longer and serve many lookups, files at higher levels can still serve many negative lookups. Thus, learning files at higher levels can make neg-



(a) Timeline of changes (b) Time between bursts for L4

Figure 5: **Changes at Levels.** *(a) shows the timeline of file creations and deletions at different levels. Note that #changes/#files is higher than 1 in L_1 because there are more creations and deletions than the number of files. (b) shows the time between bursts for L4 for different write percentages.*

ative internal lookups faster.

Learning guideline - 4: Be workload- and data-aware. Although most data resides in lower levels, if the workload does not lookup that data, learning those levels will yield less benefit; learning thus must be aware of the workload. Further, the order in which the data is loaded influences which levels receive a large fraction of internal lookups; thus, the system must also be data-aware. The amount of internal lookups acts as a proxy for both the workload and load order. Based on the amount of internal lookups, the system must dynamically decide whether to learn a file or not.

Lifetime of Levels. Given that a level as a whole can also be learned, we now measure and analyze the lifetimes of levels. Level learning cannot be applied at L_0 because it is unsorted: files in L_0 can have overlapping key ranges. Once a level is learned, any change to the level causes a re-learning. A level changes when new sstables are created at that level, or existing ones are deleted. Thus, intuitively, a level would live for an equal or shorter duration than the individual sstables. However, learning at the granularity of a level has the benefit that the candidate sstables need not be found in a separate step; instead, upon a lookup, the model just outputs the sstable and the offset within it.

We examine the changes to a level by plotting the timeline of file creations and deletions at L_1, L_2, L_3, and L_4 in Figure 5(a) for a 5%-write workload; we do not show L_0 for the reason above. On the y-axis, we plot the number of changes divided by the total files present at that level. A value of 0 means there are no changes to the level; a model learned for the level can be used as long as the value remains 0. A value greater than 0 means that there are changes in the level and thus the model has to re-learned. Higher values denote a larger fraction of files are changed.

First, as expected, we observe that the fraction of files that change reduces as we go down the levels because lower levels hold a large volume of data in many files, confirming our intuition. We also observe that changes to levels arrive in bursts. These bursts are caused by compactions that cause many files at a level to be rewritten. Further, these bursts occur at almost the same time across different levels. The reason behind this is that for the dataset we use, levels L_0 through L_3 are full and thus any compaction at one layer results in cascading compactions which finally settle at the non-full L_4 level. The levels remain static between these bursts. The duration for which the levels remain static is longer with a lower amount of writes; for example, with 5% writes, as shown in the figure, this period is about 5 minutes. However, as the amount of writes increases, the lifetime of a level reduces as shown in Figure 5(b); for instance, with 50% writes, the lifetime of L_4 reduces to about 25 seconds. From these observations, we arrive at our final learning guideline.

Learning guideline - 5: Do not learn levels for write-heavy workloads. Learning a level as a whole might be more appropriate when the amount of writes is very low or if the workload is read-only. For write-heavy workloads, level lifetimes are very short and thus will induce frequent re-learnings.

Summary. We analyzed how LSMs behave internally by measuring and analyzing the lifetimes of sstable files and levels, and the amount of lookups served by files at different levels. From our analysis, we derived five learning guidelines. We next describe how we incorporate the learning guidelines in an LSM-based storage system.

4 Bourbon Design

We now describe BOURBON, an LSM-based store that uses learning to make indexing faster. We first describe the model that BOURBON uses to learn the data (§4.1). Then, we discuss how BOURBON supports variable-size values (§4.2) and its basic learning strategy (§4.3). We finally explain BOURBON's cost-benefit analyzer that dynamically makes learning decisions to maximize benefit while reducing cost (§4.4).

4.1 Learning the Data

As we discussed, data can be learned at two granularities: individual sstables or levels. Both these entities are sorted datasets. The goal of a model that tries to learn the data is to predict the location of a key in such a sorted dataset. For example, if the model is constructed for a sstable file, it would predict the file offset given a key. Similarly, a level model would output the target sstable file and the offset within it.

Our requirements for a model is that it must have low overheads during learning and during lookups. Further, we would like the space overheads of the model to be small. We find that piecewise linear regression (PLR) [4, 25] satisfies these requirements well; thus, BOURBON uses PLR to model the data. The intuition behind PLR is to represent a sorted dataset with a number of line segments. PLR constructs a model with an error bound; that is, each data point d is guaranteed to lie within the range $[d_{pos} - \delta, d_{pos} + \delta]$, where d_{pos} is the predicted position of d in the dataset and δ is the error bound specified beforehand.

To train the PLR model, BOURBON uses the Greedy-PLR algorithm [44]. Greedy-PLR processes the data points one at a time; if a data point cannot be added to the current line segment without violating the error bound, then a new line segment is created and the data point is added to it. At the

Workload	Baseline time (s)	File model		Level model	
		Time(s)	% model	Time(s)	% model
Mixed: Write-heavy	82.6	71.5 (1.16 ×)	74.2	95.1 (0.87 ×)	1.5
Mixed: Read-heavy	89.2	62.05 (1.44 ×)	99.8	74.3 (1.2 ×)	21.4
Read-only	48.4	27.2 (1.78 ×)	100	25.2 (1.92 ×)	100

Table 1: **File vs. Level Learning.** *The table compares the time to perform 10M operations in baseline WiscKey, file-learning, and level-learning. The numbers within the parentheses show the improvements over baseline. The table also shows the percentage of lookups that take the model path; remaining take the original path because the models are not rebuilt yet.*

end, Greedy-PLR produces a set of line segments that represents the data. Greedy-PLR runs in linear time with respect to the number of data points.

Once the model is learned, inference is quick: first, the correct line segment that contains the key is found (using binary search); within that line segment, the position of the target key is obtained by multiplying the key with the line's slope and adding the intercept. If the key is not present in the predicted position, a local search is done in the range determined by the error bound. Thus, lookups take $O(log\text{-}s)$ time, where s is the number of segments, in addition to a constant time to do the local search. The space overheads of PLR are small: a few tens of bytes for every line segment.

4.2 Supporting Variable-size Values

Learning a model that predicts the offset of a key-value pair is much easier if the key-value pairs are the same size. The model then can multiply the predicted position of a key by the size of the pair to produce the final offset. However, many systems allow keys and values to be of arbitrary sizes.

BOURBON requires keys to be of a fixed size, while values can be of any size. We believe this is a reasonable design choice because most datasets have fixed-size keys (e.g., user-ids are usually 16 bytes), while value sizes vary significantly. Even if keys vary in size, they can be padded to make all keys of the same size. BOURBON supports variable-size values by borrowing the idea of key-value separation from WiscKey [35]. With key-value separation, sstables in BOURBON just contain the keys and the pointer to the values; values are maintained in the value log separately. With this, BOURBON obtains the offset of a required key-value pair by getting the predicted position from the model and multiplying it with the record size (which is $keysize + pointersize$.) The value pointer serves as the offset into the value log from which the value is finally read.

4.3 Level vs. File Learning

BOURBON can learn individual sstables files or entire levels. Our analysis in the previous section showed that files live longer than levels under write-heavy workloads, hinting that learning at the file granularity might be the best choice. We now closely examine this tradeoff to design BOURBON's basic learning strategy. To do so, we compare the performance of file learning and level learning for different workloads. We initially load a dataset and build the models. For the read-only workload, the models need not be re-learned. In the mixed workloads, the models are re-learned as data changes. The results are shown in Table 1.

For mixed workloads, level learning performs worse than file learning. For a write-heavy (50%-write) workload, with level learning, only a small percentage of internal lookups are able to use the model because with a steady stream of incoming writes, the system is unable to learn the levels. Only a mere 1.5% of internal lookups take the model path; these lookups are the ones performed just after loading the data and when the initial level models are available. We observe that all the 66 attempted level learnings failed because the level changed before the learning completed. Because of the additional cost of re-learnings, level learning performs even worse than the baseline with 50% writes. On the other hand, with file models, a large fraction of lookups benefit from the models and thus file learning performs better than the baseline. For read-heavy mixed workload (5%), although level learning has benefits over the baseline, it performs worse than file learning for the same reasons above.

Level learning can be beneficial for read-only settings: as shown in the table, level learning provides 10% improvements over file learning. Thus, deployments that have only read-only workloads can benefit from level learning. Given that BOURBON's goal is to provide faster lookups while supporting writes, levels are not an appropriate choice of granularity for learning. Thus, BOURBON uses file learning by default. However, BOURBON supports level learning as a configuration option that can be useful in read-only scenarios.

4.4 Cost vs. Benefit Analyzer

Before learning a file, BOURBON must ensure that the time spent in learning is worthwhile. If a file is short-lived, then the time spent learning that file wastes resources. Such a file will serve few lookups and thus the model would have little benefit. Thus, to decide whether or not to learn a file, BOURBON implements an online cost vs. benefit analysis.

4.4.1 Wait Before Learning

As our analysis showed, even in the lower levels, many files are short-lived. To avoid the cost of learning short-lived files, BOURBON waits for a time threshold, T_{wait}, before learning a file. The exact value of T_{wait} presents a cost vs. performance tradeoff. A very low T_{wait} leads to some short-lived files still being learned, incurring overheads; a large value causes many lookups to take the baseline path (because there is no model built yet), thus missing opportunities to make lookups faster. BOURBON sets the value of T_{wait} to the time it takes to learn a file. Our approach is never more than a factor of two worse than the optimal solution, where the optimal solution knows apriori the lifetime and decides to either immediately or never learn the file (i.e., it is two-competitive [23]). Through measurements, we found that the maximum time to learn a file is about 40 ms. We conservatively set T_{wait} to be 50 ms in BOURBON's implementation.

4.4.2 To Learn a File or Not

BOURBON waits for T_{wait} before learning a file. However, learning a file even if it lives for a long time may not be beneficial. For example, our analysis shows that although lower-level files live longer, for some workloads and datasets, they serve relatively fewer lookups than higher-level files; higher-level files, although short-lived, serve a large percentage of negative internal lookups in some scenarios. BOURBON, thus, must consider the potential benefits that a model can bring, in addition to considering the cost to build the model. It is profitable to learn a file if the benefit of the model (B_{model}) outweighs the cost to build the model (C_{model}).

***Estimating* $\mathbf{C_{model}}$.** One way to estimate C_{model} is to assume that the learning is completely performed in the background and will not affect the rest of the system; i.e., C_{model} is 0. This is true if there are many idle cores which the learning threads can utilize and thus do not interfere with the foreground tasks (e.g., the workload) or other background tasks (e.g., compaction). However, BOURBON takes a conservative approach and assumes that the learning threads will interfere and slow down the other parts of the system. As a result, BOURBON assumes C_{model} to be equal to T_{build}. We define T_{build} as the time to train the PLR model for a file. We find that this time is linearly proportional to the number of data points in the file. We calculate T_{build} for a file by multiplying the average time to a train a data point (measured offline) and the number of data points in the file.

***Estimating* $\mathbf{B_{model}}$.** Estimating the potential benefit of learning a file, B_{model}, is more involved. Intuitively, the benefit offered by the model for an internal lookup is given by $T_b - T_m$, where T_b and T_m are the average times for the lookup in baseline and model paths, respectively. If the file serves N lookups in its lifetime, the net benefit of the model is: $B_{model} = (T_b - T_m) * N$. We divide the internal lookups into negative and positive because most negative lookups terminate at the filer, whereas positive ones do not; thus,

$$B_{model} = ((T_{n.b} - T_{n.m}) * N_n) + ((T_{p.b} - T_{p.m}) * N_p)$$

where N_n and N_p are the number of negative and positive internal lookups, respectively. $T_{n.b}$ and $T_{p.b}$ are the time in the baseline path for a negative and a positive lookup, respectively; $T_{n.m}$ and $T_{p.m}$ are the model counterparts.

B_{model} for a file cannot be calculated without knowing the number of lookups that the file will serve or how much time the lookups will take. The analyzer, to estimate these quantities, maintains statistics of files that have lived their lifetime, i.e., files that were created, served many lookups, and then were replaced. To estimate these quantities for a file F, the analyzer uses the statistics of other files at the same level as F; we consider statistics only at the same level because these statistics vary significantly across levels.

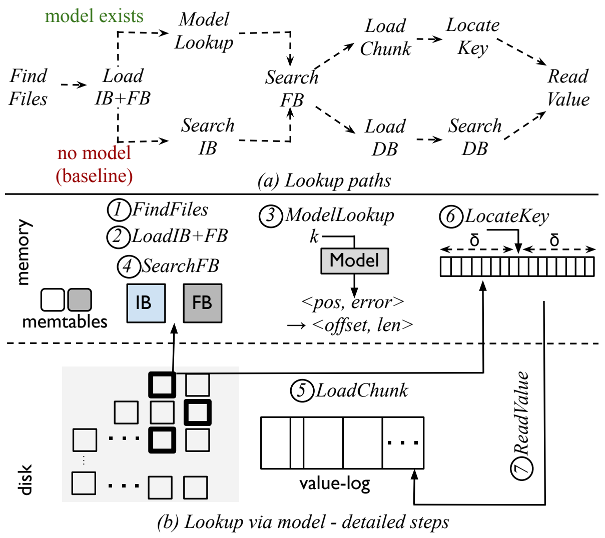


Figure 6: **BOURBON Lookups.** *(a) shows that lookups can take two different paths: when the model is available (shown at the top), and when the model is not learned yet and so lookups take the baseline path (bottom); some steps are common to both paths. (b) shows the detailed steps for a lookup via a model; we show the case where models are built for files.*

Recall that BOURBON waits before learning a file. During this time, the lookups are served in the baseline path. BOURBON uses the time taken for these lookups to estimate $T_{n.b}$ and $T_{p.b}$. Next, $T_{n.m}$ and $T_{p.m}$ are estimated as the average negative and positive model lookup times of other files at the same level. Finally, N_n and N_p are estimated as follows. The analyzer first takes the average negative and positive lookups for other files in that level; then, it is scaled by a factor $f = s/\bar{s_l}$, where s if the size of the file and $\bar{s_l}$ is the average file size at this level. While estimating the above quantities, BOURBON filters out very short-lived files.

While bootstrapping, the analyzer might not have enough statistics collected. Therefore, initially, BOURBON runs in an always-learn mode (with T_{wait} still in place.) Once enough statistics are collected, the analyzer performs the cost vs. benefit analysis and chooses to learn a file if $C_{model} < B_{model}$, i.e., benefit of a model outweighs the cost. If multiple files are chosen to be learned at the same time, BOURBON puts them in a max priority queue ordered by $B_{model} - C_{model}$, thus prioritizing files that would deliver the most benefit.

4.5 Bourbon: Putting it All Together

We describe how the different pieces of BOURBON work together. Figure 6 shows the path of lookups in BOURBON. As shown in (a), lookups can either be processed via the model (if the target file is already learned), or in the baseline path (if the model is not built yet.) The baseline path in BOURBON is similar to the one shown in Figure 1 for LevelDB, except that BOURBON stores the values separately and so *ReadValue* reads the value from the log.

Once BOURBON learns a sstable file, lookups to that file will be processed via the learned model as shown in Figure 6(b). ① *FindFiles*: BOURBON finds the candidate ssta-

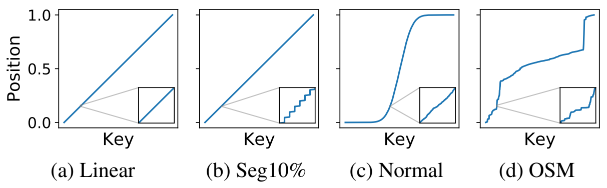


Figure 7: **Datasets.** *The figure shows the cumulative distribution functions (CDF) of three synthetic datasets (linear, segmented-10%, and normal) and one real-world dataset (OpenStreetMaps). Each dataset is magnified around the 15% percentile to show a detailed view of its distribution.*

bles; this step required because BOURBON uses file learning. ② *LoadIB+FB*: BOURBON loads the index and filter blocks; these blocks are likely to be already cached. ③ *ModelLookup*: BOURBON performs a look up for the desired key k in the candidate sstable's model. The model outputs a predicted position of k within the file (pos) and the error bound (δ). From this, BOURBON calculates the data block that contains records $pos - \delta$ through $pos + \delta$.[†] ④ *SearchFB*: The filter for that block is queried to check if k is present. If present, BOURBON calculates the range of bytes of the block that must be loaded; this is simple because keys and pointers to values are of fixed size. ⑤ *LoadChunk*: The byte range is loaded. ⑥ *LocateKey*: The key is located in the loaded chunk. The key will likely be present in the predicted position (the midpoint of the loaded chunk); if not, BOURBON performs a binary search in the chunk. ⑦ *ReadValue*: The value is read from the value log using the pointer.

Possible improvements. Although BOURBON's implementation is highly-optimized and provides many features common to real systems, it lacks a few features. For example, in the current implementation, we do not support string keys and key compression (although we can support value compression). Also, BOURBON does not support adaptive switching between level and file models; it is a static configuration. We leave supporting these features to future work.

5 Evaluation

To evaluate BOURBON, we ask the following questions:

- Which portions of lookup does BOURBON optimize? (§5.1)
- How does BOURBON perform with models available and no writes? How does performance change with datasets, load orders, and request distributions? (§5.2)
- In the presence of writes, how does BOURBON's cost-benefit analyzer perform compared to other approaches that always or never re-learn? (§5.3)
- Does BOURBON perform well on real benchmarks? (§5.4)
- Is BOURBON beneficial when data is on storage? (§5.5)
- What are the error and space tradeoffs of BOURBON? (§5.6)

Setup. We run our experiments on a 20-core Intel Xeon CPU E5-2660 machine with 160-GB memory and a 480-GB SATA SSD. We use 16B integer keys and 64B values, and set the error bound of BOURBON's PLR as 8. Unless specified, our workloads perform 10M operations. We use a variety of datasets. We construct four synthetic datasets: linear, segmented-1%, segmented-10% , and normal, each with 64M key-value pairs. In the linear dataset, keys are all consecutive. In the seg-1% dataset, there is a gap after a consecutive segment of 100 keys (i.e., every 1% causes a new segment). The segmented-10% dataset is similar, but there is a gap after 10 consecutive keys. We generate the normal dataset by sampling 64M unique values from the standard normal distribution $N(0,1)$ and scale to integers. We also use two real-world datasets: Amazon reviews (AR) [5] and New York OpenStreetMaps (OSM) [2]. AR and OSM have 33.5M and 21.9M key-value pairs, respectively. These datasets vary widely in how the keys are distributed. Figure 7 shows the distribution for a few datasets. Most of our experiments focus on the case where the data resides in memory; however, we also analyze cases where data is present on storage.

[†]Sometimes, records $pos - \delta$ through $pos + \delta$ span multiple data blocks; in such cases, BOURBON consults the index block (which specifies the maximum key in each data block) to find the data block for *pos*.

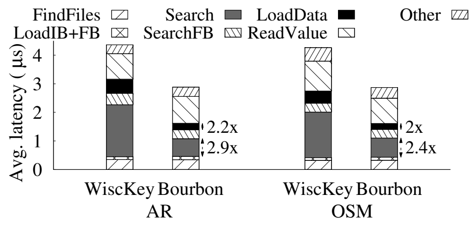


Figure 8: **Latency Breakdown.** *The figure shows latency breakdown for WiscKey and* BOURBON. *Search denotes SearchIB and SearchDB in WiscKey; the same denotes ModelLookup and LocateKey in* BOURBON. *LoadData denotes LoadDB in WiscKey; the same denotes LoadChunk in* BOURBON. *These two steps are optimized by* BOURBON *and are shown in solid colors; the number next to a step shows the factor by which it is made faster in* BOURBON.

5.1 Which Portions does BOURBON Optimize?

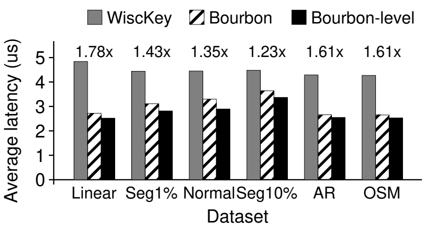
We first analyze which portions of the lookup BOURBON optimizes. We perform 10M random lookups on the AR and OSM datasets and show the latency breakdown in Figure 8. As expected, BOURBON reduces the time spent in indexing. The portion marked *Search* in the figure corresponds to *SearchIB* and *SearchDB* in the baseline, versus *ModelLookup* and *LocateKey* in BOURBON. The steps in BOURBON have lower latency than their baseline counterparts. Interestingly, BOURBON reduces data-access costs too, because BOURBON loads a smaller byte range than the entire block loaded by the baseline.

5.2 Performance under No Writes

We next analyze BOURBON's performance when the models are already built and there are no updates. For each experiment, we load a dataset and allow the system to build the models; during the workload, we issue only lookups.

5.2.1 Datasets

To analyze how the performance is influenced by the dataset, we run the workload on all six datasets and compare

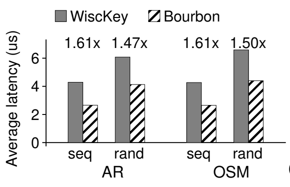


(a) Average lookup latency

Dataset	#segs	latency (μs)
Linear	900	2.72
Seg1%	640K	3.11
Normal	705K	3.3
Seg10%	6.4M	3.64
AR	129K	2.66
OSM	295K	2.65

(b) Number of segments

Figure 9: **Datasets.** *(a) compares the average lookup latencies of* BOURBON, BOURBON*-level, and WiscKey for different datasets; the numbers on the top show the improvements of* BOURBON *over WiscKey. (b) shows the number of segments for different datasets in* BOURBON.



(a) Average latency

Dataset	Positive		Negative	
	#	Speedup	#	Speedup
AR	10M	2.15×	23M	1.83×
OSM	10M	1.99×	22M	1.82×

(b) Positive vs. negative internal lookups for randomly loaded case

Figure 10: **Load Orders.** *(a) shows the performance for AR and OSM datasets for sequential (seq) and random (rand) load orders. (b) compares the speedup of positive and negative internal lookups.*

BOURBON's lookup performance against WiscKey. Figure 9 show the results. As shown in 9(a), BOURBON is faster than WiscKey for all datasets; depending upon the dataset, the improvements vary (1.23× to 1.78×). BOURBON provides the most benefit for the linear dataset because it has the smallest number of segments (one per model); with fewer segments, fewer searches are needed to find the target line segment. From 9(b), we observe that latencies increase with the number of segments (e.g., latency of seg-1% is greater than that of linear). We cannot compare the number of segments in AR and OSM with others because the size of these datasets is significantly different.

Level learning. Given that level learning is suitable for read-only scenarios, we configure BOURBON to use level learning and analyze its performance. As shown in Figure 9(a), BOURBON-level is 1.33× – 1.92× faster than the baseline. BOURBON-level offers more benefits than BOURBON because a level-model lookup is faster than finding the candidate sstables and then doing a file-model lookup. This confirms that BOURBON-level is an attractive option for read-only scenarios. However, since level models only provide benefits for read-only workloads and give at most 10% improvement compared to file models, we focus on BOURBON with file learning for our remaining experiments.

5.2.2 Load Orders

We now explore how the order in which the data is loaded affects performance. For this experiment, we use the AR and OSM datasets and load them in two ways: sequential (keys are inserted in ascending order) and random (keys are inserted in an uniformly random order). With sequential loading, sstables do not have overlapping key ranges even across levels; whereas, with random loading, sstables at one level can overlap with sstables at other levels.

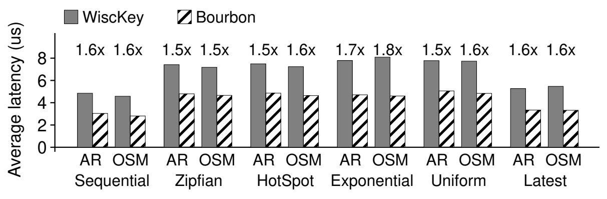


Figure 11: **Request Distributions.** *The figure shows the average lookup latencies for different request distributions for AR and OSM datasets.*

Figure 10 shows the result. First, regardless of the load order, BOURBON offers significant benefit over baseline (1.47× – 1.61×). Second, the average lookup latencies increase in the randomly-loaded case compared to the sequential case (e.g., 6μs vs. 4μs in WiscKey for the AR dataset). This is because while there are no negative internal lookups in the sequential case, there are many (23M) negative lookups in the random case (as shown in 10(b)). Thus, with random load, the total number of internal lookups increases by 3×, increasing lookup latencies.

Next, we note that the speedup over baseline in the random case is less than that of the sequential case (e.g., 1.47× vs. 1.61× for AR). Although BOURBON optimizes both positive and negative internal lookups, the gain for negative lookups is smaller (as shown in 10(b)). This is because most negative lookups in the baseline and BOURBON end just after the filter is queried (filter indicates absence); the data block is not loaded or searched. Given there are more negative than positive lookups, BOURBON offers less speedup than the sequential case. However, this speedup is still significant (1.47×).

5.2.3 Request Distributions

Next, we analyze how request distributions affect BOURBON's performance. We measure the lookup latencies under six request distributions: sequential, zipfian, hotspot, exponential, uniform, and latest. We first randomly load the AR and OSM datasets and then run the workloads; thus, the data can be segmented and there can be many negative internal lookups. As shown in Figure 11, BOURBON makes lookups faster by 1.54× – 1.76× than the baseline. Overall, BOURBON reduces latencies regardless of request distributions.

Read-only performance summary. When the models are already built and when there are no writes, BOURBON provides significant speedup over baseline for a variety of datasets, load orders, and request distributions.

5.3 Efficacy of Cost-benefit Analyzer with Writes

We next analyze how BOURBON performs in the presence of writes. Writes modify the data and so the models must be re-learned. In such cases, the efficacy of BOURBON's cost-benefit analyzer (cba) is critical. We thus compare BOURBON's cba against two strategies: BOURBON-offline and BOURBON-always. BOURBON-offline performs no learning as writes happen; models exist only for the initially loaded

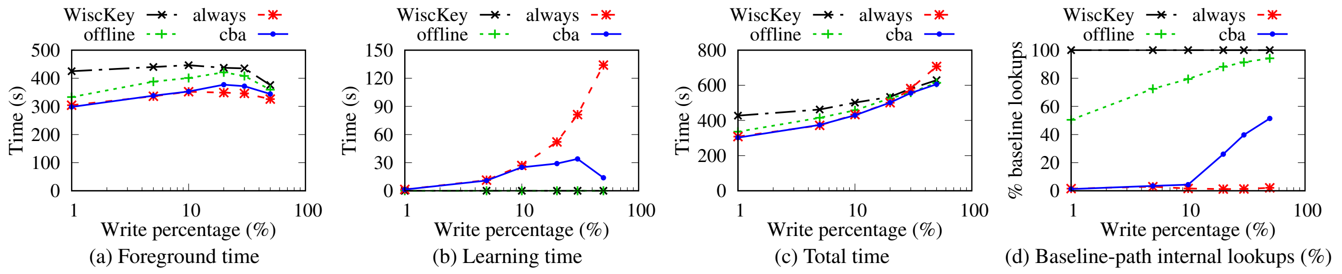


Figure 12: **Mixed Workloads.** *(a) compares the foreground times of WiscKey,* Bourbon*-offline (offline),* Bourbon*-always (always), and* Bourbon*-cba (cba); (b) and (c) compare the learning time and total time, respectively; (d) shows the fraction of internal lookups that take the baseline path.*

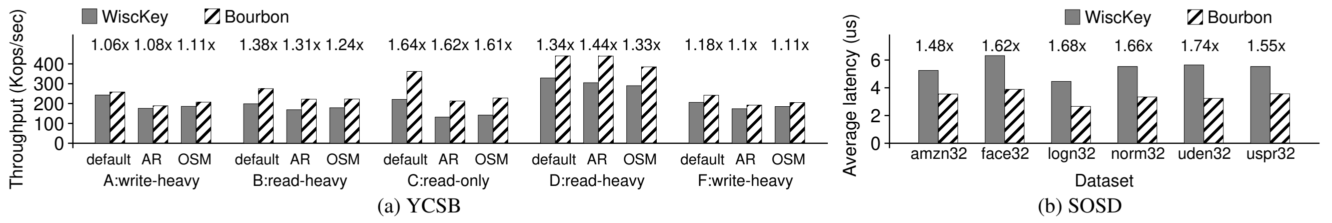


Figure 13: **Macrobenchmarks.** *(a) compares the throughput of* Bourbon *against WiscKey for five YCSB workloads for three datasets (higher is better). (b) compares lookup latencies (lower is better) for the SOSD benchmark. The numbers on the top show* Bourbon*'s improvements over the baseline.*

data. Bourbon-always re-learns the data as writes happen; it always decides to learn a file without considering cost. Bourbon-cba re-learns as well, but it uses the cost-benefit analysis to decide whether or not to learn a file.

We run a workload that issues 50M operations with varying percentages of writes on the AR dataset. To calculate the total amount of work performed for each workload, we sum together the time spent on the foreground lookups and inserts (Figure 12(a)), the time spent learning (12(b)), and the time spent on compaction (not shown); the total amount of work is shown in Figure 12(c). The figure also shows the fraction of internal lookups that take the baseline path (12(d)).

First, as shown in 12(a), all Bourbon variants reduce the workload time compared to WiscKey. The gains are lower with more writes because Bourbon has fewer lookups to optimize. Next, Bourbon-offline performs worse than Bourbon-always and Bourbon-cba. Even with just 1% writes, a significant fraction of internal lookups take the baseline path in Bourbon-offline as shown in 12(d); this shows re-learning as data changes is crucial.

Bourbon-always learns aggressively and thus almost no lookups take the baseline path even for 50% writes. As a result, Bourbon-always has the lowest foreground time. However, this comes at the cost of increased learning time; for example, with 50% writes, Bourbon-always spends about 134 seconds learning. Thus, the total time spent increases with more writes for Bourbon-always and is even higher than baseline WiscKey as shown in 12(c). Thus, aggressively learning is not ideal.

Given a low percentage of writes, Bourbon-cba decides to learn almost all the files, and thus matches the characteristics of Bourbon-always: both have a similar fraction of lookups taking the baseline path, both require the same time learning, and both perform the same amount of work. With a high percentage of writes, Bourbon-cba chooses not to learn many files, reducing learning time; for example, with 50% writes, Bourbon-cba spends only 13.9 seconds in learning (10× lower than Bourbon-always). Consequently, many lookups take the baseline path. Bourbon-cba takes this action because there is less benefit to learning as the data is changing rapidly and there are fewer lookups. Thus, it almost matches the foreground time of Bourbon-always. But, by avoiding learning, the total work done by Bourbon-cba is significantly lower.

Summary. Aggressive learning offers fast lookups but with high costs; no re-learning provides little speedup. Neither is ideal. In contrast, Bourbon provides high benefits similar to aggressive learning while lowering total cost significantly.

5.4 Real Macrobenchmarks

We next analyze how Bourbon performs under two real benchmarks: YCSB [9] and SOSD [26].

5.4.1 YCSB

We use five workloads that have different read-write ratios and access patterns: A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), F (read-modify-write:50%, r:50%). We use three datasets: YCSB's default dataset (created using *ycsb-load* [3]), AR, and OSM, and load them in a random order. Figure 13(a) shows the results.

For the read-only workload (ycsb-C), all operations benefit and Bourbon offers the most gains (about 1.6×). For read-heavy workloads (ycsb-B and D), most operations benefit, while writes are not improved and thus Bourbon is 1.24× – 1.44× faster than the baseline. Finally, for write-heavy workloads (ycsb-A and F), Bourbon improves performance only a little (1.06× – 1.18×). First, a large fraction of operations are writes; second, the number of the internal lookups taking the model path decreases (by about 30%

Dataset	WiscKey latency (μs)	BOURBON Latency(μs)	BOURBON Speedup
Amazon Reviews (AR)	3.53	2.75	1.28×
NewYork OpenStreetMaps (OSM)	3.14	2.51	1.25×

Figure 14: **Performance on Fast Storage.** *The table shows* BOURBON*'s lookup latencies when the data is stored on an Optane SSD.*

compared to the read-heavy workload because BOURBON chooses not to learn some files). In summary, as expected, BOURBON improves the performance of read operations; at the same time, BOURBON does not affect the performance of writes.

5.4.2 SOSD

We next measure BOURBON's performance on the SOSD benchmark designed for learned indexes [26]. We use the following six datasets: book sale popularity (amzn32), Facebook user ids (face32), lognormally (logn32) and normally (norm32) distributed datasets, uniformly distributed dense (uden32) and sparse (uspr32) integers. Figure 13(b) shows the average lookup latency. As shown, BOURBON is about 1.48× – 1.74× faster than the baseline for all datasets.

5.5 Performance on Fast Storage

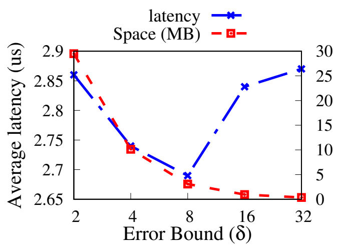
Our analyses so far focused on the case where the data resides in memory. We now analyze if BOURBON will offer benefit when the data resides on a fast storage device. We run a read-only workload on sequentially loaded AR and OSM datasets on an Intel Optane SSD. Table 14 shows the result. Even when the data is present on a storage device, BOURBON offers benefit (1.25× – 1.28× faster lookups than WiscKey). With the emerging storage technologies (e.g., 3D XPoint memory), BOURBON will offer even more benefits.

5.6 Error Bound and Space Overheads

We finally discuss the characteristics of BOURBON's ML model, specifically its error bound (δ) and space overheads. Figure 15(a) plots δ against the average lookup latency (left y-axis) for AR dataset. As δ increases, fewer line segments are created, leading to fewer searches, thus reducing latency. However, beyond $\delta = 8$, although the time to find the segment reduces, the time to search within a segment increases, increasing latency. We find that BOURBON's choice of $\delta = 8$ is optimal for other datasets too. 15(a) also shows how space overheads (right y-axis) vary with δ. As δ increases, fewer line segments are created, leading to low space overheads. Table 15(b) shows the space overheads for different datasets. As shown, for a variety of datasets, the overhead compared to the total dataset size is little (0% – 2%).

6 Related Work

Learned indexes. The core idea of our work, replacing indexing structures with ML models, is inspired from the pioneering work on learned indexes [29]. However, learned indexes do not support updates, an essential operation that an storage-system index must support. Recent research tries



(a) Error-bound tradeoff

Dataset	Space Overheads MB	Space Overheads %
Linear	0.02	0.0
Seg1%	15.38	0.21
Seg10%	153.6	2.05
Normal	16.94	0.23
AR	3.09	0.08
OSM	7.08	0.26

(b) Space overheads

Figure 15: **Error-bound Tradeoffs and Space Overheads.** *(a) shows how the PLR error bound affects lookup latency and memory overheads; (b) shows the space overheads for different datasets.*

to address this limitation. For instance, XIndex [42], FITing-Tree [18], and AIDEL [33] support writes using an additional array (delta index) and with periodic re-training, whereas Alex [12] uses gapped array at the leaf nodes of a B-tree to support writes.

Most prior efforts optimize B- tree variants, while our work is the first to deeply focus on LSMs. Further, while most prior efforts implement learned indexes to stand-alone data structures, our work is the first to show how learning can be integrated and implemented into an existing, optimized, production-quality system. While SageDB [28] is a full database system that uses learned components, it is built from scratch with learning in mind. Our work, in contrast, shows how learning can be integrated into an existing, practical system. Finally, instead of "fixing" new read-optimized learned index structures to handle writes (like previous work), we incorporate learning into an already write-optimized, production-quality LSM.

LSM optimizations. Prior work has built many LSM optimizations. Monkey [10] carefully adjusts the bloom filter allocations for better filter hit rates and memory utilization. Dostoevsky [11], HyperLevelDB [15], and bLSM [39] develop optimized compaction policies to achieve lower write amplification and latency. cLSM [21] and RocksDB [17] use non-blocking synchronization to increase parallelism. We take a different yet complimentary approach to LSM optimization by incorporating models as auxiliary index structures to improve lookup latency, but each of the others are orthogonal and compatible to our core design.

Model choices. Duvignau et al. [13] compare a variety of piecewise linear regression algorithms. Greedy-PLR, which we utilize, is a good choice to realize fast lookups, low learning time, and small memory overheads. Neural networks are also widely used to approximate data distributions, especially datasets with complex non-linear structures [32]. However, theoretical analysis [34] and experiments [40] show that training a complex neural network can be prohibitively expensive. Similar to Greedy-PLR, recent work proposes a one-pass learning algorithm based on splines [27] and identifies that such an algorithm could be useful for learning sorted data in LSMs; we leave their exploration within LSMs for future work.

7 Conclusion

In this paper, we examine if learned indexes are suitable for write-optimized log-structured merge (LSM) trees. Through in-depth measurements and analysis, we derive a set of guidelines to integrate learned indexes into LSMs. Using these guidelines, we design and build BOURBON, a learned-index implementation for a highly-optimized LSM system. We experimentally demonstrate that BOURBON offers significantly faster lookups for a range of workloads and datasets.

References

[1] BadgerDB. `https://github.com/dgraph-io/badger`.

[2] Open Street Maps. `https://www.openstreetmap.org/#map=4/38.01/-95.84`.

[3] Running a Workload. `https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload`.

[4] Jayadev Acharya, Ilias Diakonikolas, Jerry Li, and Ludwig Schmidt. Fast Algorithms for Segmented Regression. *arXiv preprint arXiv:1607.03990*, 2016.

[5] Amazon. Amazon Customer Reviews Dataset. `https://registry.opendata.aws/amazon-reviews/`.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, and Jiakai Zhang. End to End Learning for Self-driving Cars. *arXiv preprint arXiv:1604.07316*, 2016.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, WA, November 2006.

[8] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), June 1979.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD '17)*, Chicago, IL, May 2017.

[11] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-time Trade-offs for LSM-tree based Key-value Stores via Adaptive removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[12] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. ALEX: An Updatable Adaptive Learned Index. *arXiv preprint arXiv:1905.08898*, 2019.

[13] Romaric Duvignau, Vincenzo Gulisano, Marina Papatriantafilou, and Vladimir Savic. Piecewise linear approximation in data streaming: Algorithmic implementations and experimental analysis. *arXiv preprint arXiv:1808.08877*, 2018.

[14] Sarah M Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and Large-scale Anomaly Detection using a Linear One-class SVM with Deep Learning. *Pattern Recognition*, 58:121–134, 2016.

[15] Robert Escriva, Sanjay Ghemawat, David Grogan, Jeremy Fitzhardinge, and Chris Mumford. HyperLevelDB. https://github.com/rescrv/HyperLevelDB, 2013.

[16] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A Guide to Deep Learning in Healthcare. *Nature medicine*, 25(1):24–29, 2019.

[17] Facebook. RocksDB. `http://rocksdb.org/`.

[18] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*, Amsterdam, Netherlands, June 2019.

[19] Lars George. *HBase: The Definitive Guide: Random Access to Your Planet-size Data*. OReilly Media, Inc., 2011.

[20] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[21] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.

[22] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[23] Anna R Karlin, Kai Li, Mark S Manasse, and Susan Owicki. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. *ACM SIGOPS Operating Systems Review*, 25(5):41–55, 1991.

[24] Andrej Karpathy. Software 2.0. https://medium.com/@karpathy/software-2-0-a64152b37c35, November 2017.

[25] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An Online Algorithm for Segmenting Time Series. In *Proceedings 2001 IEEE international conference on data mining*, 2001.

[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A Benchmark for Learned Indexes, 2019.

[27] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. *arXiv preprint arXiv:2004.14541*, may 2020.

[28] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *Proceedings of 9th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2019.

[29] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*, Houston, TX, June 2018.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, Lake Tahoe, NV, December 2012.

[31] Avinash Lakshman and Prashant Malik. Cassandra – A Decentralized Structured Storage System. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Montana, Oct 2009.

[32] Stéphane Lathuilière, Pablo Mesejo, Xavier Alameda-Pineda, and Radu Horaud. A comprehensive analysis of deep regression. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[33] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A Scalable Learned Index Scheme in Storage Systems. *arXiv preprint arXiv:1905.06256*, 2019.

[34] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In *Advances in neural information processing systems*, pages 855–863, 2014.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[36] Mathias Meyer. The Riak Handbook, 2012.

[37] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[38] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shangai, China, October 2017.

[39] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, Scottsdale, AZ, May 2012.

[40] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[41] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go With Deep Neural Networks and Tree Search. 529(7587), January 2016.

[42] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 308–320, 2020.

[43] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. https://arxiv.org/abs/1609.08144, September 2016.

[44] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. Maximum Error-bounded Piecewise Linear Representation for Online Stream Approximation. *The VLDB journal*, 23(6), 2014.