


Article

An Empirical Study of Segmented Linear Regression Search in LevelDB

Agung Rahmat Ramadhan¹, Min-guk Choi¹, Yoojin Chung² and Jongmoo Choi^{1,*} ¹ Department of Software, Dankook University, Yongin 16890, Republic of Korea² Division of Computer Engineering, Hankuk University of Foreign Studies, Yongin 17035, Republic of Korea

* Correspondence: choijm@dankook.ac.kr; Tel.: +82-31-8005-3242

Abstract: The purpose of this paper is proposing a novel search mechanism, called SLR (Segmented Linear Regression) search, based on the concept of learned index. It is motivated by our observation that a lot of big data, collected and used by previous studies, have a *linearity* property, meaning that keys and their stored locations show a strong linear correlation. This observation leads us to design SLR search where we apply segmentation into the well-known machine learning algorithm, linear regression, for identifying a location from a given key. We devise two segmentation techniques, *equal-size* and *error-aware*, with the consideration of both prediction accuracy and segmentation overhead. We implement our proposal in LevelDB, Google's key-value store, and verify that it can improve search performance by up to 12.7%. In addition, we find that the equal-size technique provides efficiency in training while the error-aware one is tolerable to noisy data.

Keywords: big data; key-value store; learned index; segmented linear regression; LevelDB; implementation; evaluation



Citation: Ramadhan, A.R.; Choi, M.-g.; Chung, Y.; Choi, J. An Empirical Study of Segmented Linear Regression Search in LevelDB. *Electronics* **2023**, *12*, 1018. <https://doi.org/10.3390/electronics12041018>

Academic Editors: Juan M. Corchado, Byung-Gyu Kim, Carlos A. Iglesias, In Lee, Fuji Ren and Rashid Mehmood

Received: 20 December 2022

Revised: 12 February 2023

Accepted: 13 February 2023

Published: 17 February 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Search is an essential ingredient for retrieving data in databases and for finding a block in file systems. For instance, in order to handle a user request to lookup a value from a given key, many key-value stores equip themselves with various search mechanisms such as binary search and hash [1,2]. As another example, file systems have an internal index structure, such as inode and extent, to search an LBA (Logical Block Address) in storage associated with an offset of a requested file [3,4].

Many data structures have been developed to enhance performance and to reduce memory usage of search. Typical examples include B+-tree [5], trie [6], radix tree [7], ordered index [8], and so on. Recently, a new approach, called learned index, has gained attention [9–11]. The fundamental idea of learned index is making use of machine learning algorithms for search. Specifically, in the learned index taxonomy, creating a data structure is replaced with training that creates a model, while searching is replaced with inference that predicts a location of a given key using the model.

In this paper, we propose a new search mechanism based on the learned index concept. We observe several real datasets such as OSM (Open Street Map) [12] and TUD (Twitter User Data) [13], and find out that they have a specific property that we refer to as *linearity*. This property means that keys and their stored locations have a relationship that can be graphically represented linearly and can be expressed as a linear function. In other words, keys can be used as an offset to search for their locations in a sorted dataset.

Our observation opens up an opportunity to apply machine learning algorithms such as linear regression, decision tree, support vector machine, or multilayer perceptron for search. In this study, we choose the linear regression since it is well matched with the observed linearity and can be implemented efficiently in a real key-value store.

However, our initial applying of the linear regression to whole data reveals that it often causes a huge *error margin*. Here, the error margin is defined as the difference between an

predicted location and an actual location. Therefore, it can be used to assess the prediction accuracy. Most keys shows a small error margin, but some keys that violate the linearity lead to a huge one, which is discussed further in Section 3.

To overcome this problem, we employ a segmented linear regression where whole data are divided into multiple segments, and the linear regression is applied to each segment separately. Now, the question is how to make a segmentation. We devise two techniques, called *equal-size* and *error-aware* segmentation. The equal-size segmentation divides data into segments whose size are equal, while the error-aware segmentation divides data into segments wisely by finding points that violate the linearity so that each segment has stronger linear correlation between keys and their locations. The benefit of the latter approach can minimize the overall error margin while that of the first approach can achieve segmentation without scanning all keys to measure their error margins.

Our SLR search consists of two procedures: one is for training and the other is for searching. The training procedure makes a segmentation, either equal-size or error-aware, and applies the linear regression to each segment. The output of training is a model that is a set of linear functions where each function is associated with each segment. The searching procedure is composed of three steps: (1) identifying a segment related to a requested key, (2) calculating a predicted location using the linear function of the identified segment, and (3) performing *last-mile* search from the predicted location.

The final step, last-mile search, is required for all learned index based mechanisms since inaccurate predictions are inevitable when we apply machine learning [10,11]. The last-mile search is a process to resolve this prediction inaccuracy, trying to find the request key in neighbor locations, either linearly or exponentially. Note that our error-aware segmentation technique can bound the number of searched neighbors in the last-mile search.

We have implemented our proposed SLR search in LevelDB [14], a key value store developed by Google that is popularly used both for research [15,16] and products [17]. The original LevelDB utilizes binary search to locate a value from a key in an SSTable file. We extend this SSTable structure so that it can make use of SLR search when there is the linearity in stored data. Evaluation results demonstrate that SLR search indeed improves search performance by up to 12.7%, compared with the original binary search of LevelDB.

The contributions of this paper can be summarized as follows:

- We observe that there exist several real world datasets that have the linearity, providing an opportunity to apply machine learning algorithms for search;
- We design SLR search that integrates the linear regression and segmentation in a cooperative manner to mitigate the error margin caused by noisy data;
- We explore diverse design space for segmentation (equal-space and error-aware) and for last-mile search (linear, binary, and exponential);
- We implement SLR search in an actual key-value store and evaluate its effectiveness in terms of the lookup latency and number of comparisons.

The remainder of this paper is organized as follows: In Section 2, we discuss the background of this study such as learned index and LevelDB. Then, we explain our observations that motivate this work in Section 3. Design and evaluation results of SLR search are given in Section 4 and Section 5, respectively. We survey related work in Section 6. Finally, we present the conclusions and future work in Section 7.

2. Background

In this section, we discuss the key concept of learned index. Then, we explain the internal structure of LevelDB and details of the SSTable.

2.1. Learned Index

Traditionally, search mechanisms make use of various index structures such as binary tree, hash, B+-tree, trie, skip list, ordered index, and so on [1,2,5–8]. Learned index is a recently developed idea that integrates machine learning algorithms into index structures to improve search performance and to reduce the memory footprint for index structures [9].

Figure 1 illustrates the difference between traditional binary search and learned index search with a conceptual perspective. In traditional binary search, when we insert a new key-value pair, it constructs an index structure (binary tree in this figure, but our explanation is also applicable to other structures such as B+-tree and hash). When we look up a key, it uses the constructed index structure to find a key-value pair location and returns a value of the requested key. It has a time complexity of $O(\log n)$ and requires memory space to maintain the index structure.

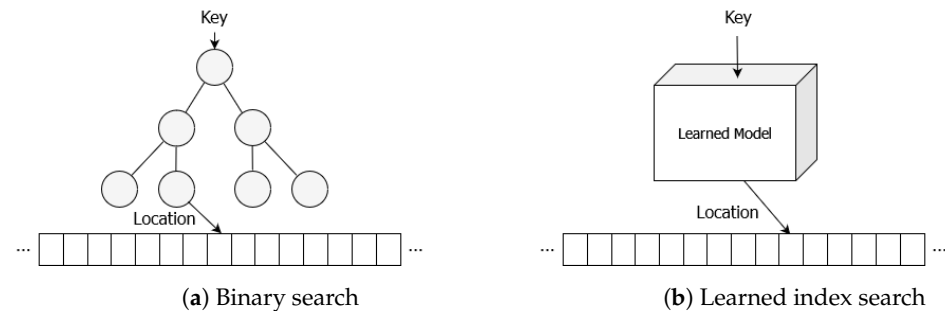


Figure 1. Conceptual comparison between traditional binary search and learned index search.

On the contrary, in learned index search, it constructs a model by training stored keys and their locations. Then, when we lookup a key, it makes use of the model to predict the location. Since the model may not be perfect, it is required to do the last-mile search to find the exact location. Finally, it returns a value of the requested key from that location. In this case, the time complexity for lookup is $O(1)$, and the memory usage of the model is usually smaller than that of the traditional index structure. This is why learned index has become popular these days [10,11].

However, to achieve the performance and space advantage of learned index, several challenges exist. The first one is how to enhance the accuracy of a model. For this purpose, several techniques, such as recursive model index [9], piecewise geometric model [18], radix spline [19], and FITing-tree [20], have been proposed. The second challenge is how to evolve a model when a new key-value pair is inserted. The third challenge is how to minimize the training and prediction overhead. In this paper, we mainly focus on the second and third challenges. We address the second challenge by exploiting the log-structured update characteristic of LevelDB and the third one by devising efficient segmentation techniques, respectively.

2.2. LevelDB

A key-value store is regarded as a de facto standard database for unstructured bigdata. There are a lot of key-value stores in industry including Google's LevelDB [14], Facebook's RocksDB [1], Amazon's Dynamo [21], and so on. In addition, there are diverse research key-value stores in academia such as Wisckey [15], PebblesDB [22], Silk [23], Pink [24], Bourbon [25], and so on. In this paper, we choose LevelDB as an experimental environment since it is well-known, open-source, and popularly used by both industry and academia. Note that our proposal can be applied to other key-value stores that are based on the LSM (Log-Structured Merge)-tree structure [15,26].

Figure 2 shows the internal structure of LevelDB. It supports user interfaces such as *put()* and *get()* for a new key-value pair insertion and lookup, respectively. It consists of two core components, called *Memtable* and *SSTable*, where *Memtable* is located in DRAM while *SSTable* is in storage. *Memtable* is a kind of write buffer where newly inserted key-value pairs are maintained. LevelDB uses the skip list data structure for *Memtable*.

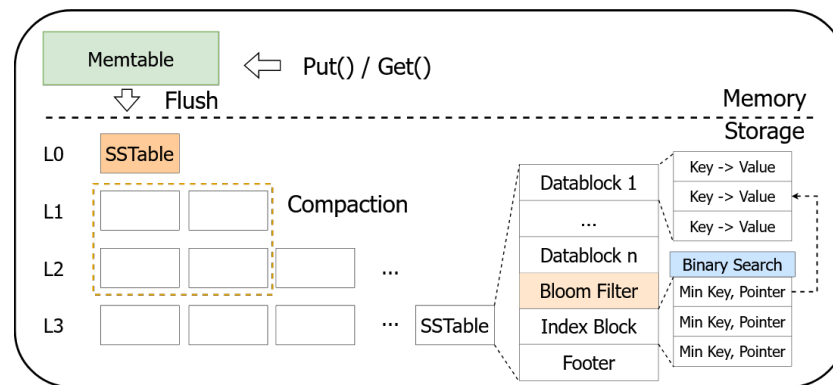


Figure 2. Internal structure of LevelDB.

When the size of Memtable becomes larger than a predefined threshold (e.g., 64 MB), it is written into storage, becoming an SSTable file. This operation is called *flush*. SSTable files are organized in multiple levels. Specifically, a flushed SSTable file is inserted into the top level (L0 in the figure). Then, when the size of SSTable files in a level becomes larger than another predefined threshold (e.g., 640 MB), key-overlapped SSTable files in the level and the next level are merged all together, becoming new SSTable files in the next level. This operation is called *compaction*. The flush and compaction operations play a momentous role in the performance of a key-value store [15,23].

Now let us discuss how a lookup request is handled. LevelDB deals with a lookup request hierarchically, from Memtable to block cache, and to SSTable files. It checks the Memtable first since newly inserted key-value pairs are managed there. If missed, it checks block cache, a DRAM component of LevelDB that manages recently accessed blocks of SSTable files. If missed again, it finally goes to SSTable files.

An SSTable file consists of numerous key-value pairs that are stored in a sorted manner. Assume that an SSTable file size is 64 MB, and an average key-value size is 1 KB; then, there exists 64,000 key-value pairs in an SSTable file. Each file has a range of keys, from the minimum to a maximum key. Using this range information, LevelDB can identify SSTable files that have a potential to contain a requested key. This identification is performed from level 0 to the next levels since upper levels have more recently inserted key-value pairs.

The detailed format of an SSTable file is presented on the right side of Figure 2. It consists of four components, namely data blocks, bloom filter, index block, and footer. All key-value pairs are distributed across data blocks. The bloom filter is used to check the existence of a key without accessing the actual data block. The index block is used to find a data block that contains a requested key. Finally, the footer manages metadata of a file and locations of the bloom filter and index block.

When a lookup request arrives at an SSTable file, LevelDB first accesses the footer, located in the end of the file, to find the location of the bloom filter. Then, based on the bloom filter, it determines whether the request key is present in the file. If the key is present, even though it can be a false positive due to the nature of the bloom filter algorithm, it examines the index block. It consists of entries for all data blocks, where each entry has the minimum key of a data block and pointer to the data block. At this stage, LevelDB performs binary search, using entries in the index block, to find a data block of the requested key. Lastly, in the found data block, it conducts binary search again to find the final location of the requested key-value pair.

The goal of this paper is revising this binary search into a learned index search, discussed in Figure 1. Our careful investigation reveals that LevelDB is well matched with the learned index concept due to the following three characteristics. First, in the SSTable, numerous key-value pairs are written in a batch style during the flush or compaction operation, which is a good timing for training. In addition, the training overhead can be hidden by overlapping with I/Os. Second, the LSM-tree structure used by many key-value stores makes use of the log-structured update, meaning that key-value pairs are written

once and read multiple times in a file. This is an appropriate workload for learned index. Finally, real unstructured datasets managed by key-value stores show the linearity, making it feasible to employ an efficient learned index mechanism in LevelDB.

3. Observation

Learned index makes use of a model in order to predict the stored location from a requested key. In a sorted dataset, a model can be considered as a method to approximate the CDF (Cumulative Distribution Function) of the dataset [9,10]. To analyze how CDF looks and which machine learning algorithm is appropriate for training a model in LevelDB, we examine two real datasets, called OSM (Open Street Map) [12] and Twitter user data [13], as shown in Figure 3. In this figure, the x -axis represents keys while the y -axis is the CDF, the location of a corresponding key in a sorted form.

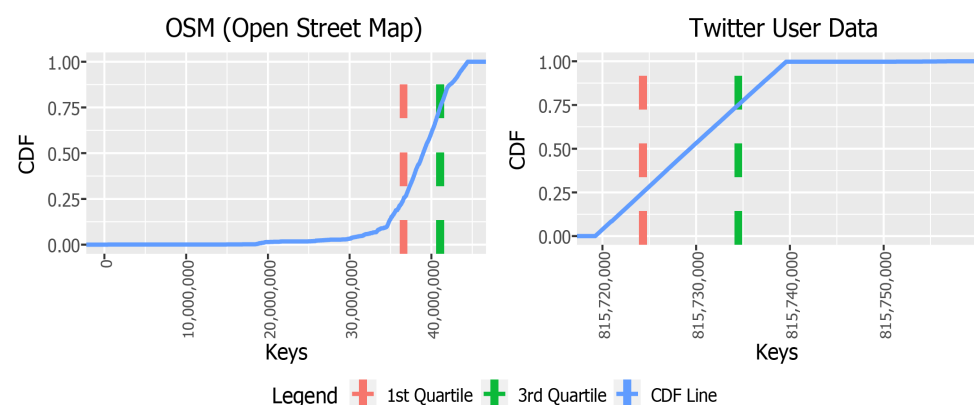


Figure 3. Cumulative distribution function of two datasets.

From Figure 3, we can make the following two observations. First, keys are distributed unevenly. For example, in OSM, keys ranging from 0 to 3×10^7 occupy only 5%, while keys ranging from 3.5×10^7 to 4.2×10^7 occupy around 75% of this dataset. This unevenness is also observed in Twitter user data. Second, keys and their locations in CDF have a linear correlation. Specifically, when we cautiously divide a dataset into multiple segments, we can obtain a linear function that can approximate keys and their locations appropriately.

To investigate our second observation more closely, we choose some samples, keys, and their locations in CDF, from OSM, and present them in Table 1 and Table 2, respectively. From Table 1, we can notice that keys and their locations satisfy a property, called *linearity*. In this paper, the linearity is defined as a property where data can be represented as a straight line so that they can be approximated as a linear function. Specifically, keys and locations in Table 1 can be approximated as a linear function with a slope of 0.8 and an intercept of $-62,301$. This approximation allows for using a key as an offset to predict the location of the key.

Note that this approximation may cause an *error margin* that is defined as the difference between the predicted and actual location. For instance, the actual location of the key 77,892 is 12 while its predicted location using the approximated linear function is 12.6. Hence, the error margin becomes 0.6 at this key. This is why we need the last-mile search to find the actual location from the predicted one. In the Table 1 case, the worst error margin is 1 at the key 77,885. It implies that we need at most one additional comparison for the last-mile search. Hence, the time complexity of this learning based search is still $O(1)$ even considering the last-mile search, which is smaller than the time complexity of binary search, $O(\log n)$.

However, keys and locations in Table 2 do not follow the linearity. There is a considerable key gap between the 92th and 93th locations, which makes the prediction inaccurate when we try to derive a linear function over all these keys. This inaccuracy leads to a big error margin, which eventually deteriorates the search performance. However, when we

divide these keys into two segments, one is from 78,101 to 78,215 and the other is from 219,850 to 219,969, and try to obtain a linear function for each segment separately, we can enhance the prediction accuracy, resulting in reducing the error margin. Now, the question is how to do this segmentation efficiently.

Table 1. Samples that satisfy the linearity.

Keys	Locations
77,880	3
77,881	4
77,883	5
77,885	6
77,886	7
77,887	8
77,888	9
77,889	10
77,890	11
77,892	12
77,893	13
77,894	14
77,895	15
77,897	16
77,898	17
77,899	18

Table 2. Samples that violate the linearity.

Keys	Locations
78,101	82
78,102	83
78,103	84
78,104	85
78,105	86
78,109	87
78,110	88
78,197	89
78,198	90
78,204	91
78,215	92
219,850	93
219,851	94
219,966	95
219,968	96
219,969	97

4. Design

In this section, we first discuss design considerations for efficient segmentation. Then, we describe two segmentation techniques devised in this paper. Finally, we explain how we materialize our proposed SLR search in LevelDB.

4.1. Segmentation

Our observations, discussed in Section 3, reveal that a lot of datasets follow the linearity, meaning that they can be approximated with a linear function. They also uncover that segmentation plays an important role on the prediction accuracy. These observations lead us to design a new search mechanism, called SLR (Segmented Linear Regression) search, that integrates the linear approximation and segmentation into a cooperative manner. In specific, SLR search makes use of two techniques for segmentation and applies the linear regression to each segment. Then, if necessary, it further divides a segment into sub-segments and applies the linear regression again to each sub-segment.

When we devise a segmentation technique, we consider the following three criteria: prediction accuracy, prediction overhead, and segmentation overhead. The prediction accuracy is defined as the worst error margin in a segment while the prediction overhead is defined as how much time it takes to find a related segment from a given key. Finally, the segmentation overhead is defined as the elapsed time required to fulfill segmentation. Based on these criteria, we propose two techniques, namely *equal-size* and *error-aware*, as shown in Figure 4.

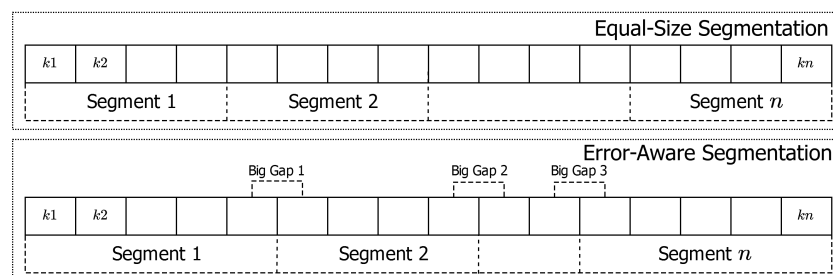


Figure 4. Two segmentation techniques.

The first technique divides a given dataset into multiple segments where each segment has the equal number of keys. The number of segments are set as a control parameter (e.g., 4 in the figure). Then, it applies the linear regression to each segment to obtain a linear function that can approximate keys and their locations in a segment. The benefit of this approach is that it can provide the best performance in terms of the segmentation overhead since it does not require calculating the error margin per each key. However, the downside is that it may degrade the prediction accuracy.

To overcome the downside, we devise our second segmentation technique that considers the error margin. It conducts the linear regression and segmentation recursively, as described in Algorithm 1. The core idea of this technique is that it divides a segment with the consideration of the error margin so that the divided segments have a smaller error margin than the original one. The benefit is that it can enhance the prediction accuracy. However, it increases the segmentation overhead to calculate the worst error margin of each segment and to conduct segmentation recursively. Note that, in a key-value store, this segmentation is performed during the flush or compaction operation, which provides an opportunity to hide this overhead by overlapping it with I/Os.

There exists a diverse design space when we design segmentation techniques. First, instead of equal-size, we can design a segmentation technique that divides segments with an equal-range of keys. For instance, assume that the minimum and maximum key in an SSTable file are 1 and 100, respectively, and the number of segments is set as 4. Then, each segment has 25 keys in it. The advantage of this approach is that it can provide the best performance in the viewpoint of the prediction overhead since finding a related segment

from a key is conducted with $O(1)$, while both the equal-size and error-aware techniques require $O(\log N)$ where N is the number of segments. However, the downside of the equal-range approach is that it may cause imbalance of segments due to the unevenness observed in Figure 3.

Algorithm 1: Error-aware segmentation

Input: A segment

```

1 obtain a linear function from the input segment;
2 calculate the worst error margin;
3 if (worst error margin > predefined error_margin threshold) then
4   if (the number of segments > predefined maximum_segment threshold) then
5     | stop segmentation and go back to the original binary search;
6   end if
7   divide the segment into two at the key of the worst error margin;
8   call the error-aware segmentation with the left segment as an input;
9   call the error-aware segmentation with the right segment as an input;
10 else
11   | add the obtained linear function into the trained model;
12 end if
  
```

Second design space is regarding the linear approximation. For this purpose, this study uses the linear regression. However, it is a heavy work, requiring the square root calculation for all keys. Hence, to assess this overhead, the linear interpolation can be a viable alternative [27]. Third, we can utilize sampling when we assess the worst error margin, which can decrease the segmentation overhead. We leave these explorations as a future work.

4.2. LevelDB Extension for SLR Search

Figure 5 shows how we modify SSTable to adopt SLR search. In the current SSTable structure, LevelDB uses the binary search in the index block for finding a data block that contains a requested key. On the contrary, in our modified SSTable structure, we extend the index block so that it has a new area, called a model area, that manages the trained linear functions for all segments. Therefore, to find a data block, we can make use of SLR search as an alternative. Note that, if the number of segments becomes larger than a threshold in our error-aware SLR search, we go back to the original binary search. For this purpose, the original index block is still maintained in the modified SSTable structure as shown in Figure 5. Finally, in the footer, a new flag is added to indicate whether this SSTable file has the model area in order to support SLR search or not.

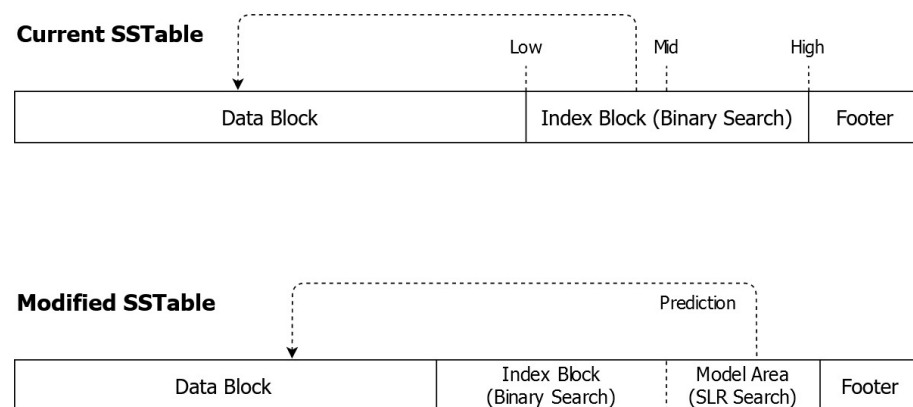


Figure 5. SSTable modification for SLR search.

The details of the index block are given in Figure 6. In the current index block, there exist n entries where each entry consists of the minimum key of a corresponding data block and pointer to the data block. In addition, there exist r entries, restart points that are used for a key prefix compression technique [14]. In a key-value store, consecutive keys have a tendency to share a same key prefix. To reduce memory usage, LevelDB drops the prefix shared with the previous key. In addition, for every k key, it does not apply the compression and store the entire key, which is called a restart point. There are r restart points where r is either n in the case where there is no shared prefix or $\lceil n/k \rceil$ in the best sharing case. As the result, in LevelDB, binary search is actually performed among restart points.

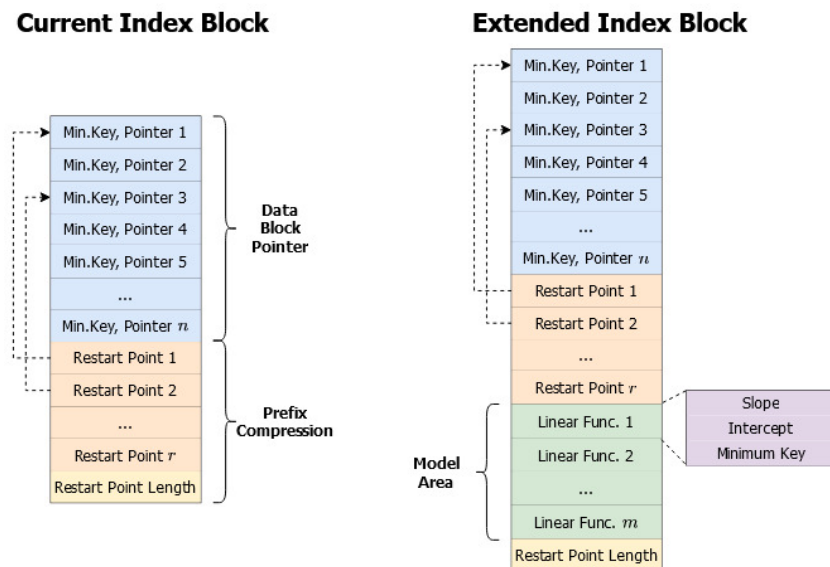


Figure 6. Index block in detail.

The right side of Figure 6 presents the extended index block for SLR search. The model area that is built as the result of training consists of m linear functions where m is the number of segments. The value of m is set as a control parameter in the equal-size SLR search or is determined based on Algorithm 1 in the error-aware SLR search. Each linear function is expressed by three fields: slope, intercept, and minimum key of a segment. The slope and intercept are used for prediction while the minimum key is used for segment finding during a lookup request handling.

Initially, we plan to train using the minimum keys in the data block pointer area. However, some key prefixes of this area are omitted due to the prefix compression technique. Hence, we use restart points for training. Since binary search is performed using restart points in original LevelDB, our choice can lead to a fair comparison when we compare SLR search with binary search.

The overall lookup process using the extended index block is conducted as follows: When a lookup request arrives at an SSTable file, LevelDB first checks the footer to locate the index block. Then, it checks whether SLR search is feasible or not. If not, it performs binary search at the cost of $O(\log r)$ using restart points to locate the target restart point related to the requested key. If feasible, it performs binary search at the cost of $O(\log m)$ using the minimum keys of segments in the model area and finds a linear function related to the request key. Then, it predicts the target restart point using the function at the cost of $O(1)$. Note that the number of segments, m , is much smaller than that of restart points, r , in the index block (e.g., r is 2048 if the SSTable size is 64 MB, and the data block size is 4 KB, and $k = 8$). Note that the default value of m is set as 20 in this study.)

The final step of the lookup process is the last-mile search. The predicted restart point may not be the correct one due to the prediction inaccuracy. To overcome this problem, SLR search utilizes the last-mile search that scans neighbors of the predicted location. Several candidates such as linear, binary, and exponential search can be used for the last-mile search.

The linear search candidate scans to the left or right neighbors one by one, while the binary search candidate performs binary search using restart points ranging from the predicted location to the location of the worst error margin distance. Finally, the exponential search candidate jumps 2^i neighbors and makes a comparison at the jumped location, as illustrated in Figure 7. When the comparison becomes above or below the requested key, it performs binary search from the previous jumped location to the final one.

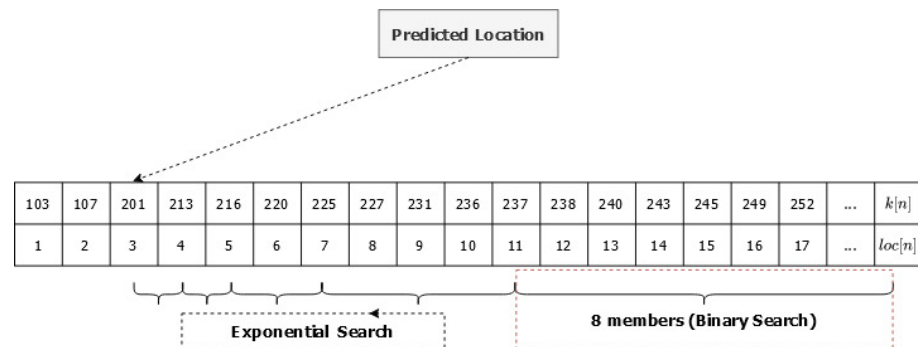


Figure 7. Last-mile search: exponential example.

There are various trade-offs among three candidates. When the error margin is small, the linear search candidate can be a simple and effective choice. On the contrary, when the error margin is large, exponential search can be a useful solution that covers a wide range quickly. The binary search candidate is in the middle, balancing between simplicity and coverage. In this study, we choose linear search for the error-aware segmentation technique, while choosing exponential search for the equal-size one. Note that the overhead of the last-mile search is important for the success of SLR search on LevelDB, and our experiments reveal that the linearity of datasets makes it insignificant, which will be further discussed in Section 5.

Until now, we have discussed how we employ SLR search into the index block. We can employ SLR search into the data block in the same way since the data block has a structure similar to the index block, shown on the left side of Figure 6. Specifically, the data block consists of two areas: (1) key-value pairs, instead of min. key and pointer pairs as discussed in Figure 2, and (2) restart points for the prefix compression. However, the number of key-value pairs is much smaller than the number of min. key and pointer pairs. For instance, assume that the data block size is 4 KB, and the average key-value pair size is 100 B; then, there are only 40 pairs in the data block. In contrast, assume that the SSTable size is 64 MB; then, the number of min. key and pointer pairs in the index block is 16,384. When the number of pairs is small, the benefit of learned index is not noticeable and its overhead rather degrades performance. Hence, we employ SLR search into the data block a little differently. We define a control parameter, called *minimum-pair threshold*, and do not use SLR search if the number of key-value pairs is less than this threshold.

5. Evaluation

This section presents performance evaluation results. We first explain our experimental environment. Then, we discuss several measurements such as lookup latency, throughput, and the number of comparisons during search.

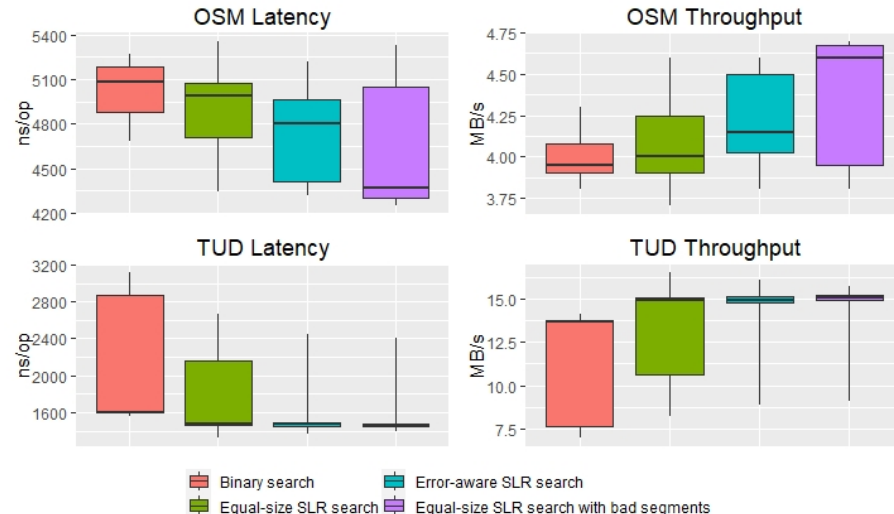
Table 3 summarizes our experimental environment. Hardware platform consists of 16 cores, 16 GB DRAM, and 150 GB NVMe SSD. On these hardware, we install the Debian 11 distribution whose Linux kernel version is 5.10. We also install the LevelDB version 1.23 and modify it so that it can utilize SLR search. Then, we use OSM (Open Street Map) [12] and TUD (Twitter User Data) [13] as workloads for our evaluation.

Table 3. Experimental system.

CPU	Intel(R) Core(TM) i7-10700K CPU @ 3.80 GHz
Memory	16,384 MB
Storage	150 GB NVMe SSD
OS	Debian 11 (Bullseye)
Key-value store	LevelDB version 1.23
Workloads	OSM (Open Street Map), TUD (Twitter User Data)

SLR search has four control parameters: number of segments for the equal-size segmentation, worst-case error margin threshold and maximum segment number threshold for the error-aware segmentation, and minimum-pair threshold for deciding whether it applies SLR search or not. In this study, we set the default values of these parameters as 20, 100, 20, and 32, respectively. The sensitivity study of the parameters will be discussed later.

Figure 8 presents the lookup latency and throughput results. In this figure, we compare four search mechanisms. The first one is binary search that is the original mechanism used by LevelDB. The second and third one, equal-size SLR search and error-aware SLR search, are our proposed mechanisms. The fourth one is called equal-size SLR search with bad segments, which is designed to evaluate if we apply equal-size SLR search with the consideration of a bad segment. Specifically, a segment that has an error margin larger than the worst-case error margin threshold is defined as a bad segment. Then, it applies binary search to bad segments instead of SLR search. In the figure, the horizontal line is the average value, while the vertical line and square represent the min/max and 1st/3rd quartiles, respectively.

**Figure 8.** Latency and throughput.

From Figure 8, we can make the following four observations. First, equal-size SLR search performs better than binary search. For instance, in the OSM case, it can reduce the average latency from 5100 ns to 4950 ns, which eventually leads to improved throughput from 3.95 MB/s to 4.06 MB/s. It uncovers that employing the learned index concept in a key-value store has a potential to enhance performance. Second, the error-aware SLR search mechanism enhances performance further, enhancing the throughput by up to 4.15 MB/s in the OSM case. It means that the error-awareness can increase the prediction accuracy, having a positive impact on performance.

Third, compared with the equal-size SLR search mechanism, the equal-size SLR search with bad segments mechanism shows better performance. It implies that it is better to use binary search instead of SLR search for the bad segment that has a big error margin.

The final observation is that the variance of two datasets exhibits different trends. For OSM, binary search shows smaller variance than SLR search. In contrast, for TUD, SLR search yields much smaller variance. Our detailed analysis shows that TUD has a stronger linearity than OSM, providing more accurate predictions, which results in less variance in SLR search.

Figure 9 presents the actual number of comparisons measured in the OSM dataset. It shows that SLR search indeed decreases the number of comparisons. In this experiment, we set the SSTable size as 64 MB and the data block size as 4 KB. Hence, there can be at most 16,384 (2^{14}) min. key and pointer pairs in the index block (due to the key prefix compression technique, the actual number is smaller than that according to the degree of prefix sharing). Note that LevelDB maintains the minimum key using a virtual key that is lower than the smallest key of the corresponding data block and that is higher than the largest key in the previous data block. Therefore, in binary search, finding a target pair is conducted at the leaf node, not in the internal node. As the result, the comparison numbers of binary search become from 9 to 13, as shown in the figure.

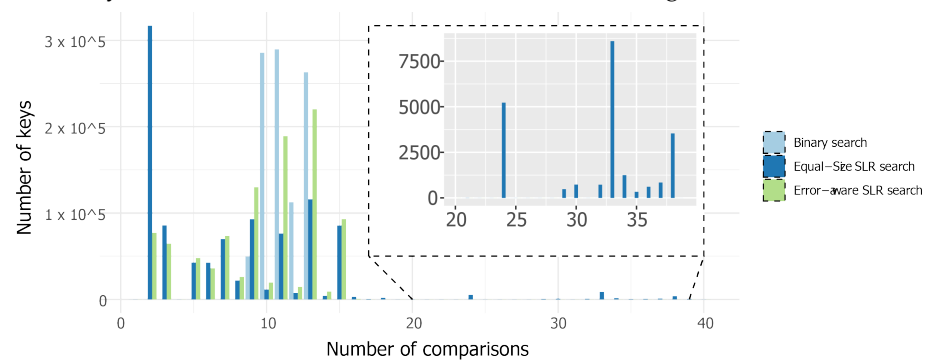


Figure 9. Number of comparisons.

On the contrary, equal-size SLR search can decrease the comparison numbers by up to 2 when it predicts the location accurately (one for finding a segment and the other for verifying the predicted location). However, it suffers from long-tail comparisons, ranging from 24 to 38, due to the bad segments where key and pointer pairs violate the linearity. Error-aware SLR search overcomes these long-tail comparisons by dividing them further, eventually reducing the error margin of a segment.

Figure 10 shows the overall lookup latency breakdown in LevelDB. We obtain these results using the *uftrace* tool that allows for tracing and analyzing the CPU execution time consumed by a specific part of a program [28]. The overall SSTable-related lookup process consists of finding SSTables, processing for loading blocks, searching in the index block, searching in data blocks, and so on. Among them, the index block search latency is the heaviest one as presented in this figure, since the index block is used more frequently than data blocks and has the largest number of elements to be searched. This is why we focus on the index block in this study.

As a sensitivity analysis, we change the number of segments in the equal-size SLR search mechanism and measure latency as shown in Figure 11. This analysis shows that, when the number of segments increases from 2 to 20, equal-size SLR search performs better by increasing the prediction accuracy due to the finer segmentation. However, when it becomes too large (e.g., 30 or 40), it performs worse due to the overhead of identifying the corresponding segment of a requested key. This result leads us to choose 20 as the default value of the number of segments in the equal-size segmentation mechanism.

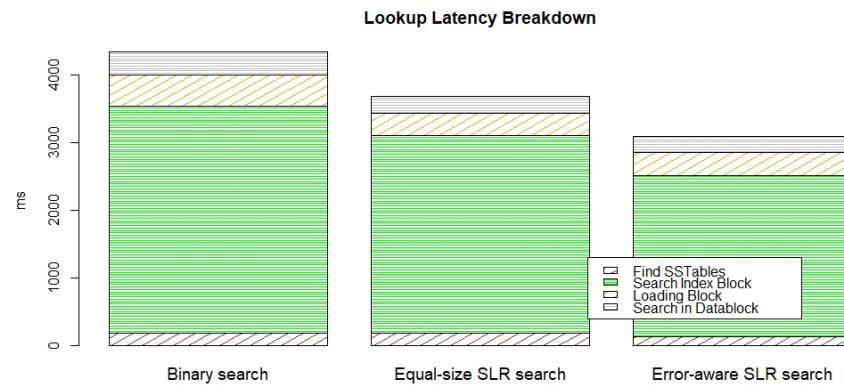


Figure 10. Lookup latency breakdown.

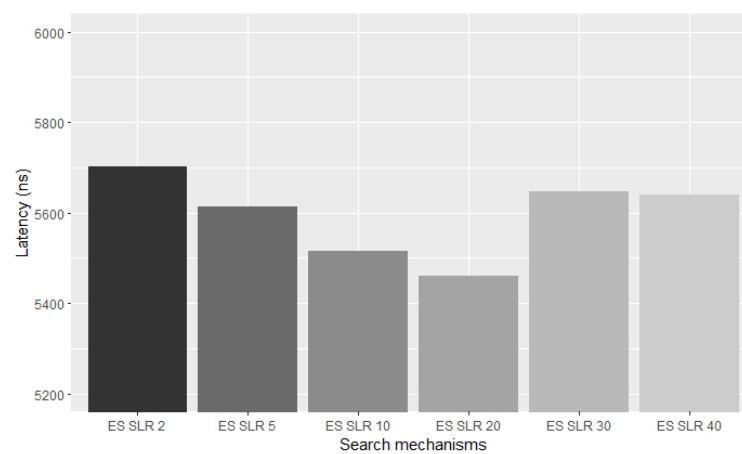


Figure 11. Effect of the number of segments.

6. Related Work

Learned index and key-value store are two prominent research topics that are studied actively these days. This section surveys previous studies that are related to our work.

Kraska et al. published a historic paper about learned index [9]. They argue that traditional indexes such as range index (e.g., B-tree), point index (e.g., hash), and existence index (e.g., bloom filter) can be replaced with learned indexes where constructing a data structure is replaced with training a model and lookup is replaced with prediction. Since then, studies on learned index have been vigorously conducted, and these can be largely divided into three categories.

The first category is investigating how to enhance the prediction accuracy [18–20]. For instance, Ferragina and Vinciguerra propose a new approach, called a Piecewise Geometric Model index (PGM-index) that is a multi-level structure where each level is constructed by piecewise linear regression [18]. Kipf et al. suggest a learned index that integrates a linear spline for approximation and a radix tree to manage spline points [19]. Galakatos et al. devise an index structure that makes use of a tunable error parameter so that it can balance lookup performance and space consumption [20]. Our approach is fundamentally similar to these, but we differ from them in that we implement a learned index based search mechanism in an actual key-value store and evaluate performance.

The second category is designing updatable learned index [29–31]. Updating data is considered an Achilles heel of learned index since it requires re-training. Ding et al. propose an updatable learned index structure, Alex, based on 4 components: gapped array, model-based insertion, exponential search, and adaptive tree structure [29]. Wu et al. design a framework, called LIPP (Learned Index with Precise Positions), that utilizes a metric to decide the layout of learned index so that it can predict precisely the location of a key [30]. Since SSTable is updated

only once while searched multiple times in an LSM-tree based key-value store, updating is not an issue in our study.

The third category is regarding fair comparison and open datasets [10,11]. As diverse learned index structures are suggested actively, a need for standard benchmarks and datasets has emerged. Marcus et al. examine various learned index structures and suggest benchmarking methodologies [10]. Wongkham et al. discuss trade-offs of various updatable learned index structures and evaluate them comprehensively with new perspectives such as concurrency and NUMA (Non-Uniform Memory Architecture) [11].

Key-value stores are also popularly studied these days [1,15,21–26,32]. For instance, Lu et al. propose an outstanding idea that separates keys and values in the LSM-tree so that it can reduce the compaction overhead in a key-value store [15]. Balmau et al. design a new key-value store, SILK, that utilizes an I/O scheduler to alleviate interference of internal operations, such as flush and compaction, to user latencies [23]. Im et al. offload a key-value store to SSDs (Solid State Devices) to adopt the benefit of in-storage processing [24].

Abu-Libdeh et al. integrate learned index into Google's Bigtable and evaluate its practicality [32]. They make use of the linear regression model like our work. However, they use the model for allocation instead of prediction, which always allows accurate prediction at the cost of increased space. Lu et al. propose a key-value store, TridentKV, with three new techniques, namely an adaptive learned index, space-efficient partition, and asynchronous read [26]. For learned index, they apply both training-while-writing and training-while-reading adaptively.

Dai et al. design a key-value store, called Bourbon [25], that is closely related to our work. They utilize the piecewise linear regression for training and implement their idea in LevelDB like us. However, our work differs from Bourbon in that (1) we explore various segmentation techniques including equal-size and error-aware, (2) we support the key prefix compression in our proposal while they do not, and (3) we employ the exponential search as the last-mile search, while Bourbon uses binary search, which has the potential to cause high overhead when predictions are inaccurate.

7. Conclusions

Learned index is a promising concept that can reduce the lookup overhead and memory usage. However, applying this concept into a real key-value store is challenging due to various obstacles such as noisy data, last-mile search, and cooperation with the prefix compression technique used already. We overcome these obstacles by exploiting segmentation, exponential search, and restart points based training, respectively. Real implementation based evaluation shows that our proposed SLR search indeed reaps the benefits of learned index in LevelDB.

There are three research directions as future work. The first one is validating our proposal with more diverse workloads such as Amazon's book popularity dataset and Wiki's timestamp dataset [10]. In addition, we will evaluate the effect of our control parameters, such as the number of segments and worst error margin threshold, under these datasets. The second direction is exploring other machine learning algorithms, including linear interpolation and regression, and sampling to reduce the training overhead. The final direction is examining the impact of learned index on LevelDB's operations such as flush, compaction, and range scan.

Author Contributions: Conceptualization, J.C. and A.R.R.; methodology, A.R.R.; software, A.R.R.; validation, M.-g.C., J.C., and Y.J.; formal analysis, A.R.R.; investigation, A.R.R.; resources, J.C.; data curation, M.-g.C.; writing—original draft preparation, A.R.R.; writing—review and editing, J.C.; visualization, A.R.R.; supervision, J.C.; project administration, Y.C.; All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-01475, (SW StarLab) Development of Novel Key-Value DB for Unstructured Bigdata), and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2022R1A2C1006050) and National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1A2C1006331).

Data Availability Statement: Not applicable.

Acknowledgments: The authors appreciate all the reviewers and editors for their precious comments and work on this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dong, S.; Kryczka, A.; Jin, Y.; Stumm, M. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21), Online Conference, 23–25 February 2021.
2. Chen, J.; Chen, L.; Wang, S.; Zhu, G.; Sun, Y.; Liu, H.; Li, F. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20), Santa Clara, CA, USA, 24–27 February 2020.
3. Arpaci-Dusseau, R.H.; Arpaci-Dusseau, A.C. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books. Available online: <https://pages.cs.wisc.edu/~remzi/OSTEP/> (accessed on 10 January 2023).
4. Neal, I.; Zuo, G.; Shiple, E.; Khan, T.A.; Kwon, Y.; Peter, S.; Kasikci, B. Rethinking File Mapping for Persistent Memory. In Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21), Online Conference, 23–25 February 2021.
5. Qiao, Y.; Chen, X.; Zheng, N.; Li, J.; Liu, Y.; Zhang, T. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22), Online Conference, 22–24 February 2022.
6. Wu, X.; Xu, Y.; Shao, Z.; Jiang, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15), Santa Clara, CA, USA, 8–2 April 2015.
7. Yu, G.; Song, Y.; Zhao, G.; Sun, W.; Han, D.; Qiao, B.; Wang, G.; Yuan, Y. Cracking In-Memory Database Index: A Case Study for Adaptive Radix Tree Index. In Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), Edinburgh, UK, 30 March–2 April 2020.
8. Wu, X.; Ni, F.; Jiang, S. Wormhole: A Fast Ordered Index for In-memory Data Management. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys'19), Dresden, Germany, 25–28 March 2019.
9. Kraska, T.; Beutel, A.; Chi, E.H.; Dean, J.; Polyzotis, N. The Case for Learned Index Structures. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18), Houston, TX, USA, 10–15 June 2018.
10. Marcus, R.; Kipf, A.; Renen, A.; Stoian, M.; Misra, S.; Kemper, A.; Neumann, T.; Kraska, T. Benchmarking learned indexes. *Proc. VLDB Endow.* **2020**, *14*, 1.
11. Wongkham, C.; Lu, B.; Liu, C.; Zhong, Z.; Lo, E.; Wang, T. Are updatable learned indexes ready? *Proc. VLDB Endow.* **2022**, *15*, 11.
12. Fazal, N.; Marinescu-Istodor, R.; Fränti, P. Using Open Street Map for Content Creation in Location-Based Games. In Proceedings of the 29th Conference of Open Innovations Association (FRUCT'21), Tampere, Finland, 12–14 May 2021.
13. Twitter User Data. Available online: <https://data.world/data-society/twitter-user-data> (accessed on 17 November 2022).
14. LevelDB: A Fast Key-Value Storage Library Written at Google. Available online: <https://github.com/google/leveldb> (accessed on 17 November 2022).
15. Lu, L.; Pillai, T. S.; Arpaci-Dusseau, A. C.; Arpaci-Dusseau, R. H. 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, USA, 22–25 February 2016.
16. Kaiyakhmet, D.; Lee, S.; Nam, B.; Noh, S.H.; Choi, Y. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19), Boston, MA, USA, 22–25 February 2019.
17. Products Related to LevelDB. Available online: <https://discovery.hgdata.com/product/leveldb> (accessed on 17 November 2022).
18. Ferragina, P.; Vinciguerra, G. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* **2020**, *13*, 8.
19. Kipf, A.; Marcus, R.; Renen, A.; Stoian, M.; Kemper, A.; Kraska, T.; Neumann, T. RadixSpline: A single-pass learned index. In Proceedings of the third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'20), Portland, OR, USA, 19 June 2020.
20. Galakatos, A.; Markovitch, M.; Binnig, C.; Fonseca, R.; Kraska, T. FITing-Tree: A Data-aware Index Structure. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19), Amsterdam, Netherlands, 30 June–5 July 2019.
21. Elhemali, M.; Gallagher, N.; Gordon, N.; Idziorek, J.; Krog, R.; Lazier, C.; Mo, E.; Mritunjai, A.; Perianayagam, S.; Rath, T.; et al. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22), Carlsbad, CA, USA, 11–13 July 2022.
22. Raju, P.; Kadekodi, R.; Chidambaram, V.; Abraham, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China, 28–31 October 2017.
23. Balmau, O.; Dinu, F.; Zwaenepoel, W.; Gupta, K.; Chandhiramoorthi, R.; Didona, D. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19), Renton, WA, USA, 10–12 July 2019.

24. Im, J.; Bae, J.; Chung, C.; Arvind; Lee, S. PinK: High-speed In-storage Key-value Store with Bounded Tails. In Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20), Online Conference, 15–17 July 2020.
25. Dai, Y.; Xu, Y.; Ganesan, A.; Alagappan, R.; Kroth, B.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20), Online Conference, 4–6 November 2021.
26. Lu, K.; Zhao, N.; Wan, J.; Fei, C.; Zhao, W.; Deng, T. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 8.
27. Setiawan, N.F.; Rubinstein, B.I.P.; Borovica-Gajic, R. Function Interpolation for Learned Index Structures. In Proceedings of the 31st Australasian Database Conference (ADC'20), Melbourne, Australia, 3–7 February 2020.
28. Uftrace. Available online: <https://github.com/namhyung/uftrace> (accessed on 17 November 2022).
29. Ding, J.; Minhas, U. F.; Yu, J.; Wang, C.; Do, J.; Li, Y.; Zhang, H.; Chandramouli, B.; Gehrke, J.; Kossmann, D.; Lomet, D.; Kraska, T. Alex: An updatable adaptive learned index. In Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20), Portland, OR, USA, 14–19 June 2020.
30. Wu, J.; Zhang, Y.; Chen, S.; Wang, J.; Chen, Y.; Xing, C. Updatable learned index with precise positions. *Proc. VLDB Endow.* **2021**, *14*, 8.
31. Wang, Y.; Tang, C.; Wang, Z.; Chen, H. SIndex: A Scalable Learned Index for String Keys. In Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), Tsukuba, Japan, 24–25 August 2020.
32. Abu-Libdeh, H.; Altınbüken, D.; Beutel, A.; Chi, E.; Doshi, L.; Kraska, T.; Li, X.; Ly, A.; Olston, C. Learned Indexes for a Google-scale Disk-based Database. In Workshop on ML for Systems at NeurIPS 2020, Vancouver, BC, Canada, 12 December 2020.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.