CS736 Project 2 Final Report: Range Query Optimizations and Lazy Learned Indexes for Bourbon

Suryadev Sahadevan Rajesh*
sahadevanraj@wisc.edu
UW-Madison

Shyam Murthy*
shyamm@cs.wisc.edu
UW-Madison

Guanzhou Hu*
guanzhou.hu@wisc.edu
UW-Madison

ABSTRACT

Bourbon is a LevelDB variant equipped with learned indexes to speedup single-point query indexing steps [2]. It does not optimize for range query performance. It aggressively builds up file models whenever a compaction is triggered and the cost-benefit analyzer gives a positive score. We propose *learned file iterators* to speed up iterator seeking and optimize for small range queries. We use multithreading to parallelize large range queries in Bourbon. Additionally, we introduce *lazy learned indexes* to delay model learning until the first query arrives at the file. Our range query optimizations can reduce up to 15% latency for small range queries of length 128 and speed up large range queries of length 64K by at most 1.41x. In addition to the range query optimizations, the lazy learning strategy shows an improvement by learning 13% fewer file models compared to the existing Bourbon implementation for workloads with hot spots.

1 INTRODUCTION

Bourbon [2] is a learned indexes enhanced *log-structured merge tree* (LSM tree) database system built upon LevelDB [3] and WiscKey [5]. In an LSM tree database, table files are sorted by key and are read-only. Bourbon trains every table file into a *piece-wise linear regression* (PLR) model and uses the models to speedup indexing steps of single-point `Get` queries.

Though the idea of using a segmented linear regression model to boost sorted file indexing is general enough to be applied to all LevelDB operations, including point queries, range queries, and deletion, Bourbon focuses on improving the performance of single-point `Get` queries. We argue that learned file models can also be applied to range queries to speed up the iterator `Seek` operation done once at the beginning of every range query. Typical database range queries are small - they only cover around 5 to 100 existing keys in the database. For such small range queries, seeking latency dominates. We propose *learned file iterators*, enhanced table file iterators that use pre-learned file models to avoid binary search seeking, while still maintaining the same constant-time `Next` operation implementation. Learned file iterators can reduce range query latency by 9% for a range length of 1024 up to 15% for a range length of 128. For larger range queries where seeking latency no longer dominates, we apply multithreading to parallelize it, allowing up to 1.41x speedup when using an optimal number of worker threads.

From the learning strategy standpoint, Bourbon learns all the files that satisfy certain criteria based on the cost-benefit score and its life time [2]. In real world workloads, however, some files may not be used for reading data. As LSM is a write optimized storage structure, usually, more data is written to it compared to what is read. Hence, resources consumed for learning models for these files go without benefit. Based on the assumption that there can be hot spots in the database for certain workloads and data that are closely stored will be accessed repeatedly, we can provide an optimized strategy. To make the process of file model learning resource efficient, we propose a new strategy called lazy learning, which triggers the model learning upon making the first request for data in the file, with an assumption a file accessed once will be accessed again lot more. We saw 13% fewer models being learned for the workload with hot spots with lazy learning and without any or very minimal degradation in the latency of `Get` and `Put` operations.

Section 2 provides background knowledge on Bourbon PLR file models and original LevelDB range query support through the iterator interface. Section 3 presents our optimization techniques of learned file iterators, parallel range queries, and lazy learned indexes. Implementation details of these three techniques are described in section 4 and experiment results follow in section 5. Section 6 briefly discusses possible future directions of further optimization for Bourbon. Section 7 concludes this report.

2 BACKGROUND

Section 2.1 explains the internal structure of the PLR file models used by Bourbon. A detailed anatomy of original LevelDB range query semantics through the database iterator interface is given in section 2.2. Our work is closely related to these two aspects of Bourbon.

Definition of LSM tree and query workflows of LevelDB, WiscKey, and Bourbon have been explained in great detail in lectures, so we decided not to repeat these basic ideas in this report.

2.1 PLR Model in Bourbon

A table file in LevelDB is a list of key-value pair entries, sorted by the keys using a user-defined comparator function. Files in level 0 may cover overlapping ranges with each other, while files in lower levels (starting from level 1) cover distinct ranges and form sorted levels.

Bourbon learns a PLR file model for each compacted table file. A PLR model is a collection of monotonically growing segments, with the key as x-axis and its entry offset in file as y-axis. Figure 1 is an example visualization of a learned PLR file model.

Keys in Bourbon must have fixed lengths. Values can have variable lengths in LevelDB mode, but must have fixed lengths in WiscKey mode because only pointers to actual values (in a separate value log) will be stored in table files. Hence, in WiscKey mode, differences between nearby y-axis offsets are fixed as the entry size of a key-value pair.

*All authors contributed equally to this research.

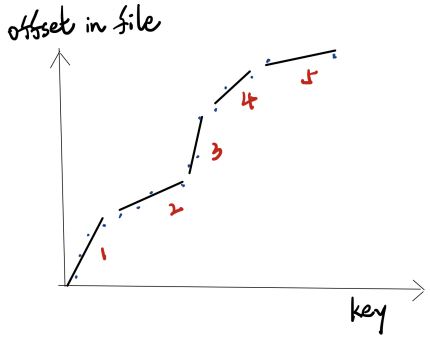

Figure 1: PLR File Model Visualization.

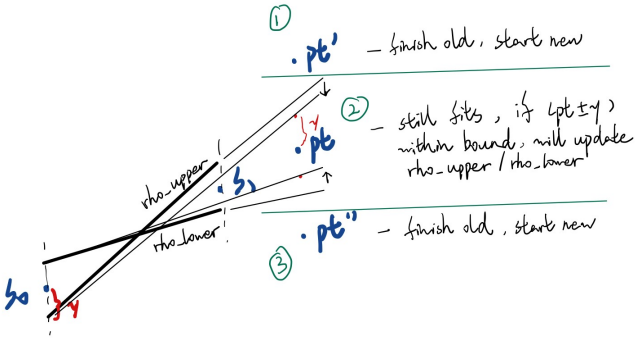

Figure 2: Greedy Training Step Demonstration.

Bourbon adopts a greedy PLR training algorithm. Building up the model for a file takes $O(n)$ time, where n is the number of keys in file. Training starts with the segment computed from the first two points. Then, it loops through the rest of the keys. For each new point encountered, there are three possible situations as demonstrated in Figure 2.

(1) *New point is above the error bound of current segment.* Bourbon will end the current segment and start a new segment with the new point.
(2) *New point still fits in the error bound of current segment.* Bourbon will extend the current segment to include the new point. If the new point pt yields a more strict error bound ($pt \pm \gamma$) than the current segment bound, Bourbon will shrink the segment error bound accordingly.
(3) *New point is below the error bound of current segment.* This situation cannot happen in a correct LevelDB image, because a larger key must have a larger offset in a sorted table file.

When a file model has been learned, looking up a key in the file takes only $O(\log S + \log e)$ time, where S is the number of segments in the model and e is the size of error bound. Bourbon will first binary search for the segment covering that key, use the chosen segment's slop and intersection to predict target key's offset, and then perform a local binary search within error bound to locate the correct entry.

The above design has two weak points which can potentially be optimized. First, learning a file model is time-consuming. Bourbon uses a *cost-benefit analyzer* (CBA) to dynamically decide whether or not to learn a file model for a newly compacted table file, depending on the file's predicted lifetime. We propose a further optimization of lazy learned indexes to delay the learning until the first query arrives at the file. This avoids the overhead of learning un-queried table files. Second, the model does not guarantee accurate prediction of a non-existing key. If target key falls into the range after the end point of some segment and before the start point of the next segment, then the predicted error bound may not even cover the closest keys, given the left segment having a large slope. This affects how we support learned file iterators for range queries and will be elaborated in more detail in section 4.2.

2.2 LevelDB Range Query Anatomy

Bourbon focuses on optimizing for single-point `Get` queries. Range queries are also common workloads in database systems, which deserve active optimizations as well.

Original LevelDB already has a decent support for range queries through the `Iterator` interface. To perform a range query on the range `[start, limit]` to retrieve all entries with keys falling in the range, the client can use the code snippet below:

```
Iterator *it = db->NewIterator(read_options);

for (it->Seek(start);
     it->Valid() && it->key().ToString() <= limit;
     it->Next())
{
  std::cout << "Key: "     << it->key().ToString()
            << ", Value: " << it->value().ToString()
            << std::endl;
}
```

The internal structure of a LevelDB iterator is way more complex than what it logically looks like. Figure 3 shows a detailed explanation of the internal structure of a LevelDB iterator. A LevelDB iterator is a "super" merge iterator composed of a set of "children" iterators:

- A memory table iterator;
- File (`Table`) iterator per file in level 0;
- Concatenating iterator per level, starting from level 1.

Performing a `Seek` on a LevelDB iterator will logically position the iterator at the smallest key in DB which is greater or equal to the seek target. The seek operation internally triggers seeks on every child iterator and then selects the smallest key pointed to by children iterators. Once seek to the start of a range, calling `Next` is very efficient, because it only advances the last chosen child iterator and then do the smallest comparison once more. The following two code snippets show the core logic of these two iterator operations:

```
// table/merger.cc

virtual void Seek(const Slice& target) {
  for (int i = 0; i < n_; i++)
    children_[i].Seek(target);
  FindSmallest();
  direction_ = kForward;
}
```

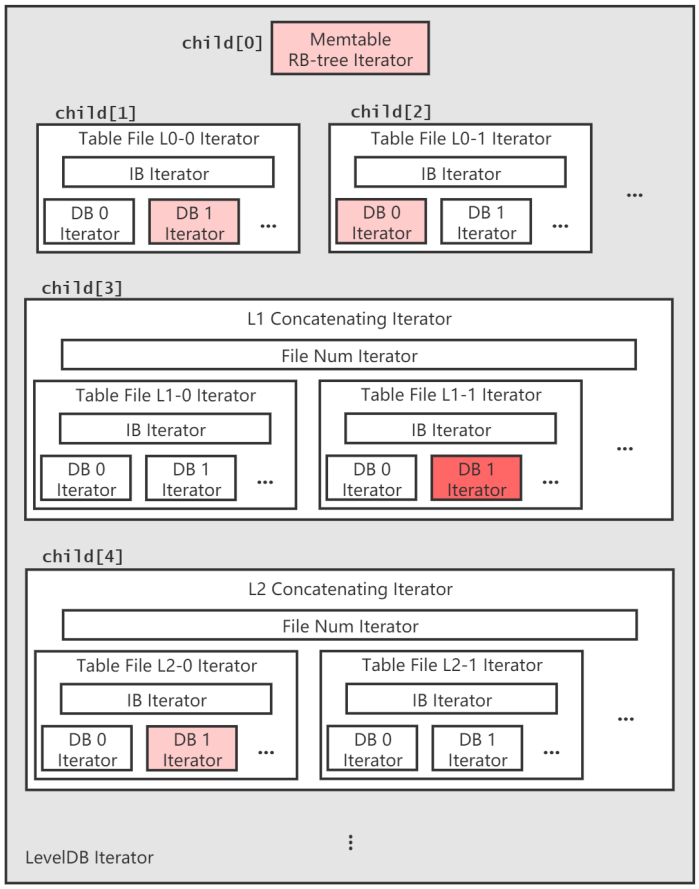

Figure 3: LevelDB Iterator Internal Structure and Explanations of the Seek Operation.

```
virtual void Next() {
  assert(Valid());
  current_->Next();
  FindSmallest();
}
```

Using an iterator is much more efficient than doing a collection of single-point queries for keys in range, because the iterator is able to seek through the files sequentially and totally skip non-existent keys. However, there are still opportunities for latency optimizations. First, the `Seek` operation is around 10x slower than the `next` operation. For small range queries with only 5 to 100 existing keys within range, latency of seeking will dominate. We boost iterator seeking for small range queries using learned file iterators. Second, WiscKey and Bourbon are designed to fit on SSD devices, which have a large degree of internal parallelism. Lu et al. discovered in the development of WiscKey that large range queries covering more than thousands of keys can be parallelized to fully utilized the internal parallelism of underlying SSD devices [5] [4]. We implemented range query parallelization support for Bourbon to boost large range queries.

3 OPTIMIZATIONS

We present the techniques we used to enable three optimizations to Bourbon: learned file iterators seeking in section 3.1, large range query parallelization in section 3.2, and lazy learned indexes in section 3.3.

3.1 Learned File Iterators Seeking

The time complexity of a range query can be modeled as:

$$T_{rq} = T_{seek} + k \cdot T_{next},$$

- T_{seek} is the time to perform seeking on the super LevelDB iterator;
- T_{next} is the time to perform a `Next` on the iterator, which is a small constant time less than 1 μs;
- k is the number of keys actually existing in the DB within given range.

The seek operation is expensive, as it forces a seek on every non-empty child iterator. Time for a seek operation can be further decomposed into:

$$T_{seek} = T_{mem\_seek} + f_0 \cdot T_{file\_seek} + \sum_{l \geq 1} (\log f_l + T_{file\_seek}),$$

- T_{mem_seek} is the time to seek the memtable, which is a tiny constant time;
- T_{file_seek} is the time to seek a file;
- f_l is the number of files in level l, for $l \geq 0$. All files in level 0 must be searched. For lower levels, LevelDB will first binary search by file ranges to locate the possible file covering target key, then perform a seek on that file.

Our approach is to replace every file iterator component in the super LevelDB iterator with a learned PLR file model. Figure 4 demonstrates our modifications. The replacement includes not only all the level 0 file iterators children, but also all the file iterators residing in level concatenating iterators for lower levels.

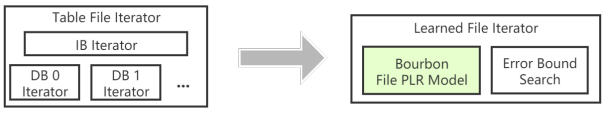

Figure 4: Replacing Every File Iterator Component with a Learned File Iterator Powered by a PLR Model.

Originally, seeking a file iterator to a target key will be translated as: first binary search the index iterator for the appropriate data block, then binary search the data block for the desired entry. Its time complexity can be modeled as:

$$T_{file\_seek} = O(\log B + \log E),$$

where B is the number of data blocks in the file and E is the number of entries within a data block. By replacing the file iterator with a Bourbon PLR file model, the work reduces to a PLR model lookup plus a binary search in the small error bound:

$$T_{learned\_file\_seek} = O(\log S + \log e),$$

where S is the number of PLR segments in the model and e is the size of error bound. Notice that $S << E$ for a file and $\log e$ is a tiny constant time since the error bound is no larger than eight entries.

3.2 Range Query Parallelization

For large range queries covering more than thousands of existing keys, the time of seeking no longer dominate. Hence, for large range queries, we take the parallelization approach to let multiple threads process distinct sections of the range and fully utilize the internal parallelism of underlying SSD devices [5].

We enabled parallelization of a range query by creating a thread pool when opening a database instance. We partition the key range into multiple non-overlapping partitions and let each thread in pool handle one partition. Partitioning is better than interleaving different threads with each other on the range, because then each thread can perform its own iterator scan over its partition, without affecting other threads. Assuming a fairly uniform key distribution within range, each thread will get a similar amount of workload (i.e., one seek plus a similar amount of `Next` operations).

The number of worker threads is decided when opening the LevelDB instance. The threads will then sleep and wait until jobs come in. We believe the optimal number of threads should be carefully chosen to match the internal parallelism degree of the underlying SSD device and, of course, should not exceed the actual number of cores of the server CPU. A reasonable number for most commercial SSDs is 32.

3.3 Lazy Learned Indexes

In this approach, we try to incorporate a lazy learning strategy in learning indices for Bourbon. The idea is to learn the file model for a file only when the file is looked up, so that in some cases we can avoid the learning cost or resources used in learning models for files which are not referenced at all.

This strategy is based on the intuition that, for some workloads, there will be more locality, i.e., keys referenced in a file may be referenced again (temporal locality) or other nearby keys in the files are likely to be referenced more (spatial locality). However, in LSM trees, the amount of data written is much larger than the amount of data read. By our hypothesis, we can eliminate learning file models for lots of files, which is highly beneficial if we are running the DB in a highly resource-constrained environment.

Our analysis is that, assume the resource cost of learning a file model is C_F. T_F is the total number of files in the database. There are T_{LF} files in the LSM database that are learned by clearing the wait time and cost-benefit score threshold, and only R_F of them are required to serve read requests. R_{LF} is the read files clearing the same threshold and wait time.

The invariant in the system is that the number of files read is always smaller than or equal to the total number of files in the system:

$$R_F \leq T_F$$

$$T_{LF} \leq T_F$$

$$R_{LF} \leq R_F$$

Since the same criteria for cost-benefit analysis of learning is applied for selecting a file for learning, we have:

$$R_{LF} \cap T_{LF} = R_{LF}$$

Hence, we can say that:

$$R_{LF} \leq T_{LF}$$

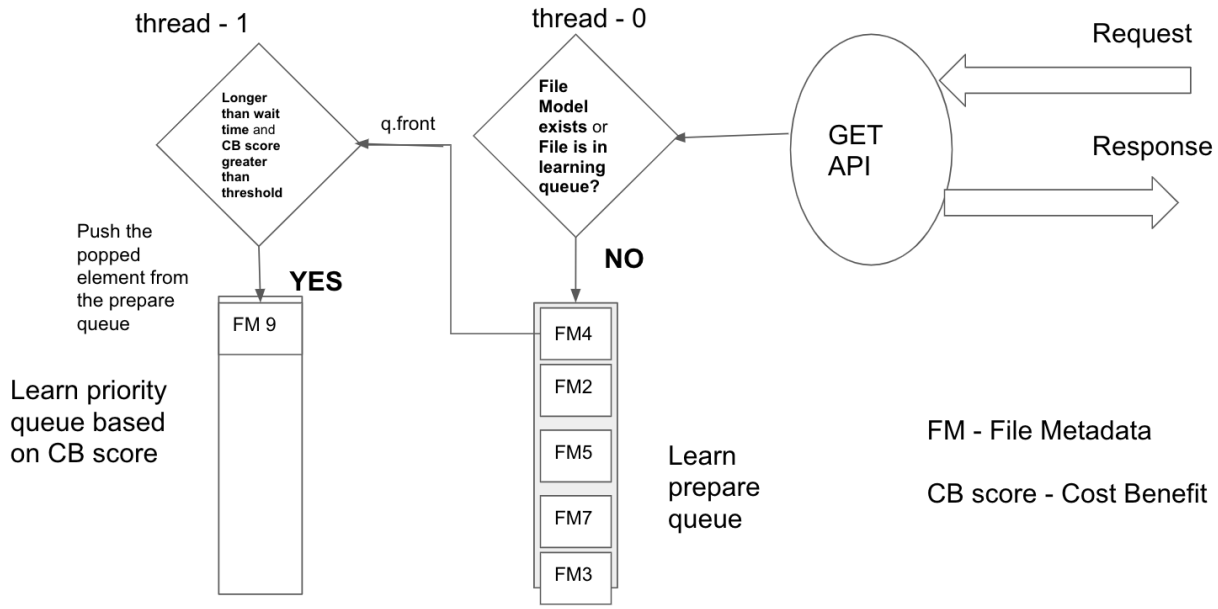

Figure 5: Lazy learning - Modified Learning Flow in Bourbon.

The total learning cost in Bourbon $C_{baseline}$ and the learning cost in lazy learning mode C_{lazy_learn} are:

$$C_{baseline} = C_F \cdot T_{LF}$$

$$C_{lazy\_learn} = C_F \cdot R_{LF}$$

Since C_F is constant, we can say that:

$$C_{lazy\_learn} \leq C_{baseline}$$

The above analysis shows that the learning cost of lazy learning is smaller than or equal to the baseline Bourbon learning cost.

We put a file into the learning queue on the first lookup of the file, i.e., when the first `Get` is invoked on that file. We keep the files in the queue ordered based on the calculated CBA score. This can be made as a new mode/policy and can be enabled only for certain kinds of workloads which would perform better with this policy. Figure 5 summarizes our lazy learning workflow.

4 IMPLEMENTATION

We implemented learned file iterators seeking, paralleled range queries, and lazy learned indexes in Bourbon by adding in a new namespace `projMod` into the system for new functionalities such as learned file iterators and worker thread pools, plus a small amount of in-place modifications of Bourbon to hook up the new functionalities.

Our modifications include around 800 lines of new code and less than 100 lines of in-place modifications. Code is available through the archive file submitted along with this report on Canvas.

4.1 Range Query Method Interface

We added a new method interface called `RangeQuery` to the `DB` class. This method will perform paralleled range query if the parallelization flag is set. If this flag is not set, it will construct learned file iterators for the query if the current operating mode is Bourbon, otherwise fall back to normal LevelDB range queries.

Below is a code snippet example of using this new range query method interface:

```
// Defines a range query on range [start, start+len-1].
projMod::RangeQueryRequest rq(start, len,
                              next_key_func);
bool paralleled = true;
db->RangeQuery(read_options, rq, paralleled);

// Read value of key "179".
std::cout << rq["179"] << std::endl;
```

4.2 Handling Non-existing Keys in Learned File Iterators

For learned file iterators, one tricky issue is that, for a non-existing target key, the original file model does not guarantee the returned error bound to cover the smallest key greater than or equal to the target. This could happen when the target key lies right after the end point of some segment and before the start point of the next segment. Figure 6 is a demonstration of such situation.

We solved this issue by adding an extra linear calculation of the start point of the next segment to the predicted segment. If the prediction result is even greater than the next segment's start point,

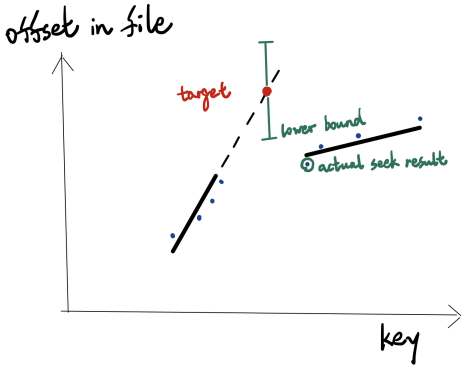

Figure 6: Situation Where PLR Model Prediction Does Not Cover Correct Seek Result.

than do not predict anywhere larger than that offset. Instead, just return the prediction as the next segment's start point.

4.3 Enabling Lazy Learning

To implement the lazy learning strategy, it required only around 100 lines of code. We introduced lazy learning as a separate operating mode in the implementation so that the application can configure Bourbon to use lazy learning if needed.

We started with the database that has only data files with no learned models. We made sure that files are pushed to learning queue only for the files that had positive lookup for a `Get` request. Before pushing the file to the `learning_prepare` queue, we make sure that, (i) the file does not have an already learned model, and (ii) the file is not already present in the learning queue phase.

To check these conditions, we used the `file_learned` state that we get by reading the flag from the corresponding `LearnedIndex` for the file. However, to check if the file is in queue or not, we added a flag to the file meta data structure and we update the flag when the file is pushed into or popped from the queue.

The background compaction thread follows the same existing scheme in Bourbon on which file to pick from the queue and when to pick, using the existing methodology of having a priority queue sorted based on CBA score and also using wait times to avoid learning the index for short-lived files.

5 EVALUATION

We evaluated our range query optimizations and lazy learned indexes implementation on a CloudLab `c220g1` node. Results show that learned file iterators can reduce seek latency by 15%, hence benefiting small range queries. Paralleled range query gets speedup up to 1.41x with an optimal number of worker threads, but is inherently limited by the repeated work of seeking and the overhead of multithreading. Lazy learned indexes can bring down the percentage of files learnt up to 13%. It also helps bring down the time spent learning file models by 29% and minimally impacts the performance of `Get` and `Put` operations.

Results for range query optimizations can be easily reproduced on a CloudLab node of the same type with:

```
cd ~/Bourbon/proj2-scripts
sudo ./run_all_proj2.sh
```

5.1 Hardware Configuration

We ran experiments on a CloudLab `c220g1` type node. The node is equipped with Intel Xeon E5-2630 v3 CPU at 2.40GHz frequency. The processor has 32 cores, 32KB L1 data cache, 32KB L1 instruction cache, 256KB L2 cache, and 20MB L3 cache. The node has 128GB of DDR4 memory at 1866MHz frequency. The storage device is a single Intel DC S3500 480GB SATA SSD. The SSD device has an internal parallelism (queue depth) of 32.

5.2 Learned File Iterators Seeking

To evaluate range query latency performance, we created two database images with two different synthetic datasets: the *uniform* dataset consists of 70M integer keys uniformly distributed across the space $[0, 2^{32} - 1]$; the *linear* dataset consists of 1000 perfect monotonic linear segments, each having a slope chosen from $[1, 64]$, again with a total number of 70M integer keys. The linear dataset suits Bourbon better than the uniform dataset because it is inherently linearly segmented and every learned file model thus has fewer number of segments. We picked a total number of 70M keys in order to give a decent coverage over all levels of LevelDB. Once loaded into a database directory, both datasets produce a database image with the number of files in each level shown in Table 1.

Level	0	1	2	3	4	5
# Files	3	7	75	513	1030	0

Table 1: Number of Files Per Level in 70M Keys DB Image.

We first compare the latency improvements of offline learned file iterators against baseline WiscKey mode on range queries of different lengths. Figure 7 shows the result. Notice that the lengths mean the size of the range, not the number of actual existing keys in the database within range. Since the two datasets both have an average interval of around 32 between keys, the number of existing keys within range can be approximated as dividing the range length by 32. Range length 128 (the leftmost group) covers only 3 to 5 existing keys in average, which means that its performance is very close to a single Seek operation. Baseline WiscKey delivers a Seek in 6.6 μs in average, while our learned file iterators reduce this latency down to 5.6 μs, showing an improvement of seek performance by 15%. Both versions deliver a `Next` operation in around 0.4 μs. As expected, as the length of range queries growing up, the relative speedup goes down, because the number of `Next` operations grow linearly and learned file iterators are only improving the seek operation that is done once at the beginning of every range query.

Table 2 compares the latency performance of learned file iterators on the two different datasets. As expected, Bourbon file models perform better on the *linear* dataset because of its segmented nature and fewer number of segments per file.

5.3 Range Query Parallelization

We measured the latency performance improvement of range query parallelization with different numbers of worker threads on the *linear* dataset image using range queries of length 64K. Every such

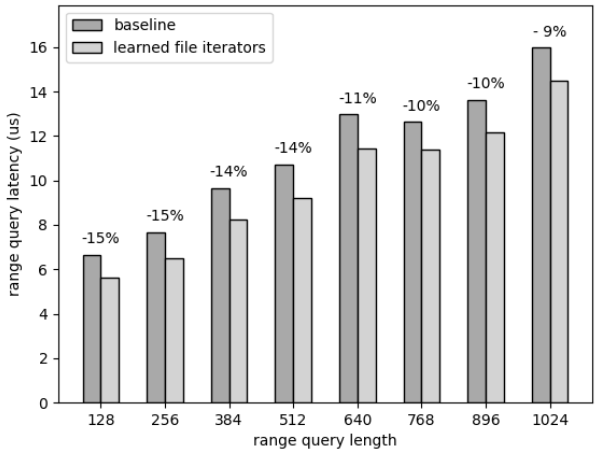

Figure 7: Range Query Latency with Learned File Iterators against Baseline WiscKey on Different Range Lengths. The database is loaded the *linear* dataset with a total number of 70M keys. The *x*-axis labels the range length, not the number of existing keys in the database within range. Average interval between keys in our image is around 32, hence, the 128 range length covers approximately 4 keys, etc.

Dataset	Baseline	Learned Seeking	Improvement
uniform	8.35 μs	7.55 μs	-9.6%
linear	7.68 μs	6.52 μs	-15.1%

Table 2: Learned File Iterators Latency Improvement on Different Datasets. Both versions are running range queries of length 256.

range query covers approximately 2048 existing keys. The worker threads partition the range evenly by the key space and each constructs its own LevelDB iterator on its assigned range. Figure 8 shows the result.

There are three observations we can get from the result. First, the optimal number of worker threads indeed match the internal parallelism of underlying SSD devices, as we can see the shortest actual work time comes from 24 to 32 worker threads. Second, the amount of multithreading overhead steadily grows with the number of threads, because work division and dispatching takes longer. Third, the relative speedup ratio is bad and far from linear speedup. Even in the best case of 24 threads, we are not getting better than 1.5x speedup. One possible reason is that when the range query is partitioned onto multiple threads, each thread must perform its own `Seek` operation to the start of its partition. Hence, the overall work is greater than performing a single `Seek` followed by `Nexts` on a single thread. Another possible reason is that the workload is not evenly distributed across threads, because it is partitioned by the logical key space. Some threads may get a region of more closely positioned keys than others, making them the perform bottleneck. *Work stealing* can be applied in this situation, to let early finishing threads help remaining threads instead of barely waiting.

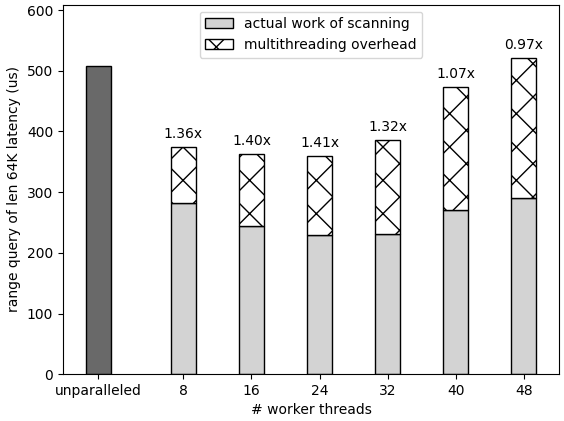

Figure 8: Paralleled Large Range Query Latency with Different Number of Worker Threads. The database is loaded the *linear* dataset with a total number of 70M keys. Latency numbers are measured from range queries of length 64K, covering approximately 2048 existing keys each. The *actual work* part measures the LevelDB iterator loop and the *multithreading overhead* part covers work dispatching and synchronization overheads across threads. Internal queue depth of the Intel DC S3500 SATA SSD device is 32.

5.4 Lazy Learned Indexes

In this section, we try to empirically prove the claim that we made theoretically about the lazy learning (LL) strategy in section 3.3. To evaluate the lazy learning strategy, we used the YCSB [1] workloads, since they give numerous controls over generating different distributions of traces.

To stress the property of lazy learning, we generated a custom workload B' with these characteristics with 10M records and 10M total operations on it: a read-heavy workload (95% read and 5% write), with hot spots on 1% of the data which serves 90% of the operations. The request distribution that we followed for this experiment is *zipfian*. Also, we compared against the default workload *B*.

We chose this workload for the following reasons: (i) to avoid disruption caused by write-heavy workloads which our implementation does not handle well at the moment (however, will be included in the future work for a complete study); (ii) to show the advantage of lazy learning, we wanted to create hot spots so that reads are focused only on a particular dataset and *zipfian* distribution to aid it further. To do the experiments, we first load the YCSB-load traces and, based on the configuration, if offline learning is enabled, we allow models to be built. Subsequently, we run the YCSB-run traces to measure performance.

Dataset	Bourbon (off+on)	LL (on)	LL (off+on)
Workload-B	298	283	280
Workload-B (with hotspots)	294	285	255

Table 3: Number of File Models Learned in Each Operating Mode. LL - Lazy Learning, off - Offline, on - Online.

Operation	Bourbon (off+on)	LL (on)	LL (off+on)
Get	8.72 μs	8.61 μs	8.61 μs
Put	1.6 μs	1.6 μs	1.6 μs

Table 4: Get and Put Latency for Different Modes of Learning in Workload-B (with hotspots). LL - Lazy Learning, off - Offline, on - Online.

Dataset	Bourbon (off+on)	LL (on)	LL (off+on)
Workload-B (with hotspots)	0.7 s	2.9 s	0.5 s

Table 5: Model Learning Time for Different Modes of Learning. LL - Lazy Learning, off - Offline, on - Online.

In table 3, we could find that lazy learning performs slightly better than the baseline Bourbon, as the number of files learned during lazy learning is smaller. Lazy learning (offline+online) mode means that file models for the data loaded into the database already exist, and we do online learning during the run of the database based on the new files created). The Lazy learning (fully online) mode means to learn file models based on the incoming requests.

We can see from the table 3 that the gain in the custom workload is slightly better than the default workload because of the hot spots which makes fewer files to be referenced again a larger number of times, so there's a higher probability for files to not be referenced, thus saving us from learning models for those files. From table 4, we can see that latency of serving requests is not affected by lazy learning for both put and get requests, since learning is done in background when there is a positive lookup on the file. Since in lazy learning, only few requests will be served in the baseline path, and the model for the file will be learned asynchronously as soon as there comes the first reference to the file, we did not see much degradation in the `Get` requests. Since `Put` requests do not get affected by these, setup continues to remain unaffected.

In table 5, we can see that during the run phase (i.e., excluding the initial data load phase of YCSB), the Lazy learn (online) mode has spent more time on model learning compared to the other modes, since there are no pre-existing models in the database. Also, we can observe that lazy learning (offline + online) has shorter model learning time compared to the baseline Bourbon as expected.

However, performance improvements brought by lazy learning is not as great as what we expected. We feel that there might be in the distribution that we used to think of having hot spots. It is possible that a large number files actually get referenced at least once, though the majority of them are referenced only a few times due to the configuration of hot spots. To study and improve this, we can create a benchmark to simulate a real life workload having requests to only a small fraction of keys, where the keys are spread over a small fraction of the total number of files. We can also make our algorithm more robust, which we discuss in section 6.

6 FUTURE WORK

Besides the optimizations we have implemented and evaluated, we believe there are abundant possibilities of further optimizations to Bourbon. First, for learned iterators, it is possible to replace lower level concatenating iterators with a full Bourbon level model, instead of a sorted collection of learned file models. This could further improve seek performance. Second, for large range queries, every worker threads' iterator seek operation could adopt learned seeking. Third, work stealing could be applied to paralleled range queries to better balance workload distribution among threads.

Finally, to improve the results for lazy learning, we could pursue a new compaction and sorting technique so that logically-related key-value entries can be placed in the same file, thus creating more hot spots and further reducing the number of models getting learned. Also, another method that we could experiment is, instead of learning the model for the file on the first request, we make a more sophisticated decision by learning the model after some k lookups on the same file.

7 CONCLUSION

We enabled three optimizations to Bourbon: *learned file iterators* to speed up small range queries, multithreading parallelization to speed up large range queries, and lazy learned indexes to avoid unnecessary file learning for write-intensive workloads. We reduced small range query latency with learned iterators seeking by up to 15%. We speed up large range queries with multithreading by up to 1.41x. Also, with our further improvement of lazy learning, we made the model learning procedure more efficient by not learning the model that will not be used at all. With workloads with hot spots, the lazy learning strategy reduces the number of file models learnt by 13% with minimal impact on the latency of `Get` and `Put` operations.

REFERENCES

[1] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[2] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[3] Google. 2019. LevelDB. https://github.com/google/leveldb.

[4] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/3064176.3064187

[5] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu