Machine Learning for Data Management Systems

Learned LSMs

Amol Deshpande
February 9, 2023

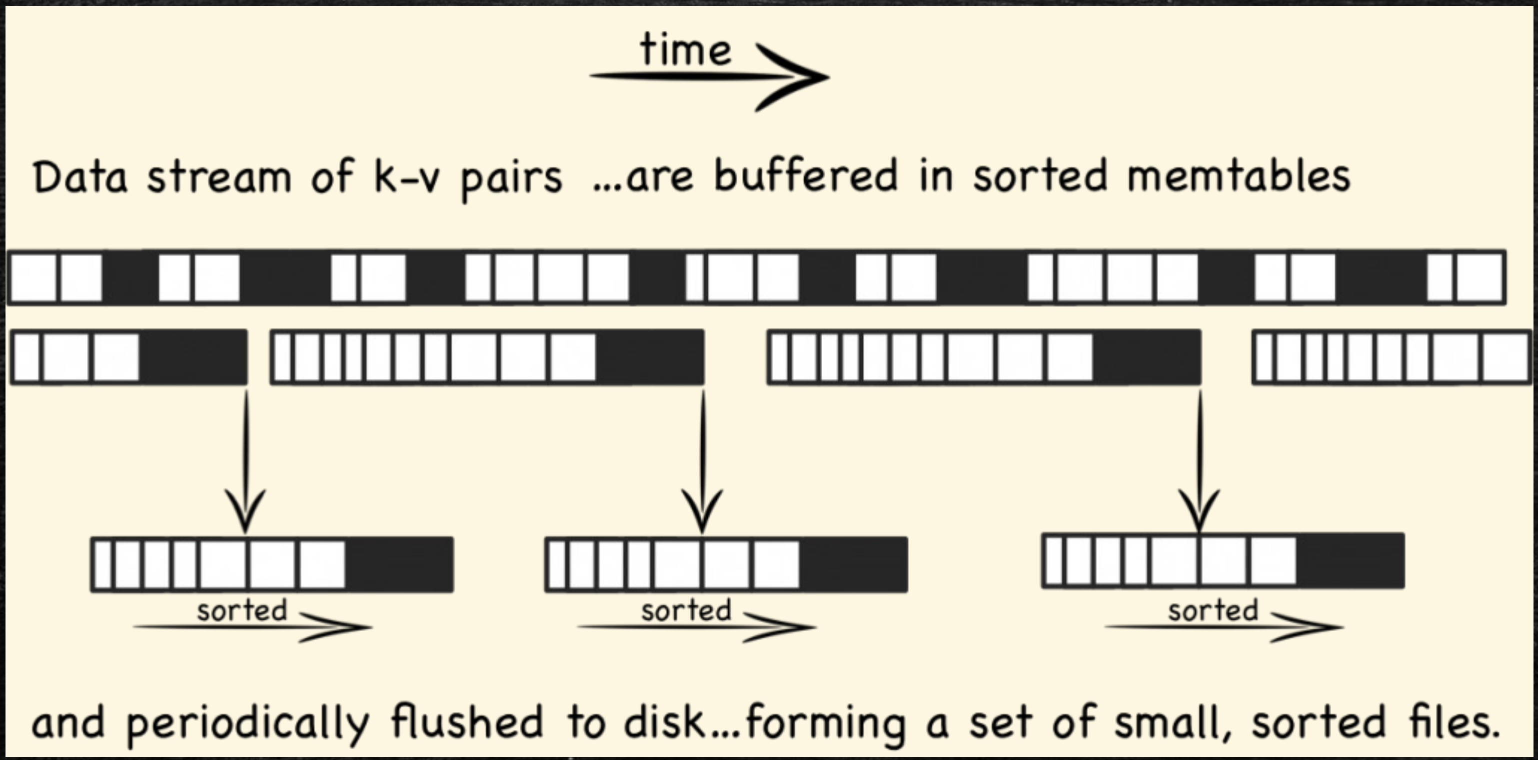
Outline

- **Log-structured Merge Trees**
- WiscKey → Bourbon
- Discussion

Log-structured Merge Trees

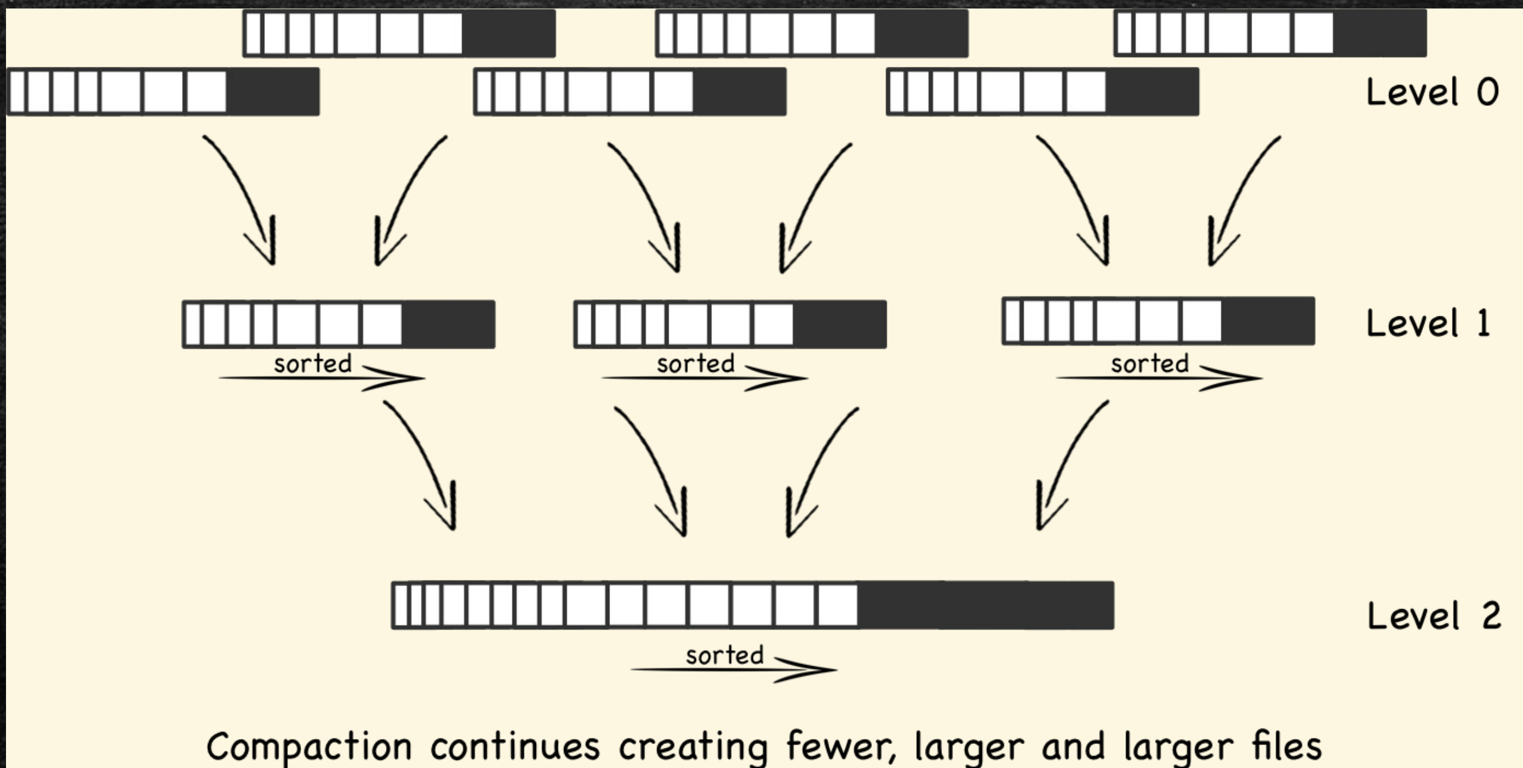
- Very widely used today, in most modern systems
  - RocksDB, Cassandra, LevelDB, InfluxDB, Bigtable, …
- Key insight:
  - B+-trees/Hash index etc., require ”in-place” random updates
    - Have to do reorganizations when updates are made
  - Not a good idea as the gap between random and sequential increases
  - Also, not a good idea for SSDs which don’t like small writes
  - Instead
    - Keep all the data sorted in memory and build red/black tree or binary tree on it
    - As you run out of memory, write out the sorted “run” to disk and never modify it again (except see below)
    - When “searching”, you have to search all of ”runs” -- so periodically compact them to reduce the number of runs

Log-structured Merge Trees



http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

Log-structured Merge Trees



http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

Log-structured Merge Trees

- Benefits:
  - Large sequential writes
  - Compaction is done in a batched fashion and exploits the sorted nature → very fast and sequential writes as well

- Drawbacks:
  - Searching is expensive
  - Same "key" is present in many different files
  - Can use "bloom filters" to reduce the number of files touched
    - Read up if interested

WiscKey

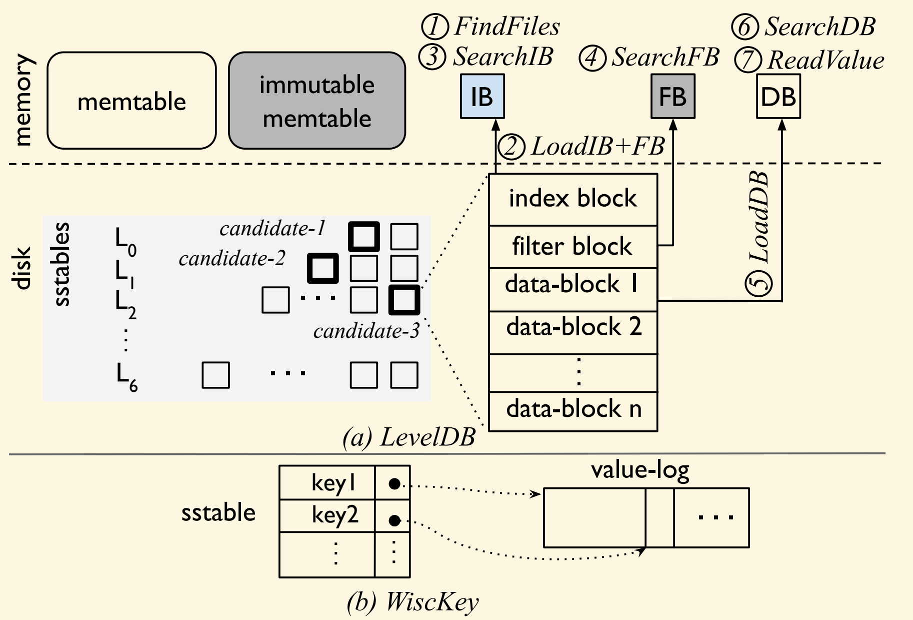


Figure 1: **LevelDB and WiscKey.** *(a) shows how data is organized in LevelDB and how a lookup is processed. The search in in-memory tables is not shown. The candidate sstables are shown in bold boxes. (b) shows how keys and values are separated in WiscKey.*

WiscKey

- Main Differences
  - "Values" kept in a separate log so that the SSTables/MemTables are small
    - SSTables now contain a pointer to the value
    - Improves compaction times
  - Requires an extra read at the end (once you have a location of the value)
  - Range queries are more expensive (values are not sorted)
  - Bloomfilter for each "block" within a run?
    - The Bourbon paper suggests this, but the original paper says a BF for each run

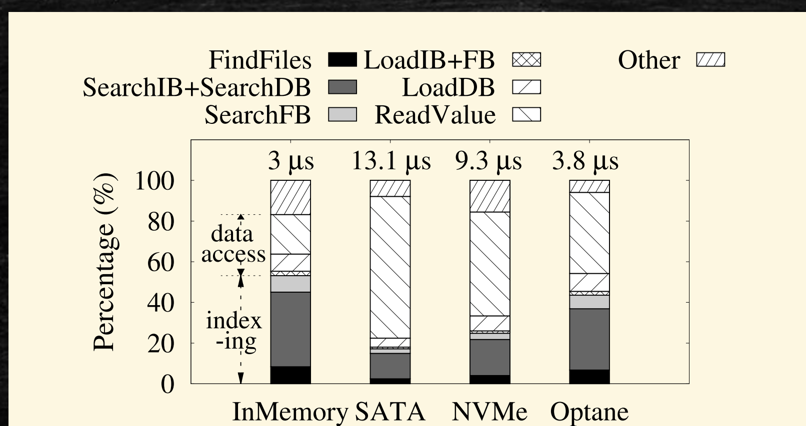
may be present in all L_0 files (because of overlapping ranges) and within one file at each successive level. ② *LoadIB+FB*: In each candidate sstable, an index block and a bloom-filter block are first loaded from the disk. ③ *SearchIB*: The index block is binary searched to find the data block that 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

Outline

- Log-structured Merge Trees
- **WiscKey → Bourbon**
- Discussion

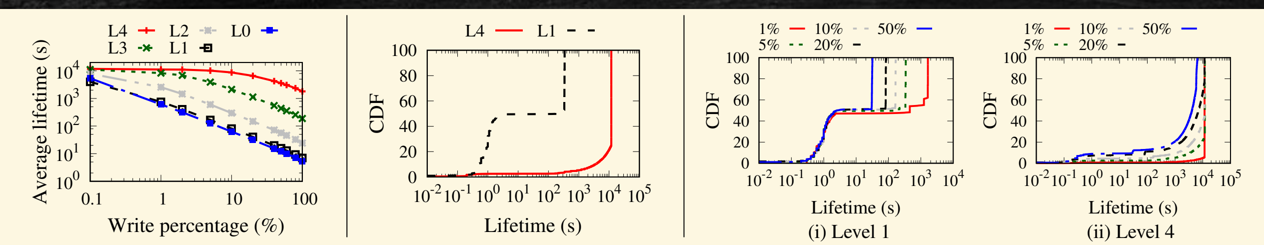
Are Learned Indexes a Good Match?

- How big is the indexing cost compared to overall cost?
- Indexing costs high if the data is in-memory, but not if on an SSD
  - Access costs will dominate, so not much benefit to improving indexing (about 20% assuming no cost to indexing)
  - Amdahl's law



Are Learned Indexes a Good Match?

- Is there any point in building expensive indexes if there are too many write?
- Let's look at how often SSTables are deleted
  - Lower levels have decent lifetimes, but not upper levels -- especially for high write ratios
  - Interestingly: a significant fraction of SSTables live for a very short time even in lower levels
    - Compaction often propagates all the way down



(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.*

Are Learned Indexes a Good Match?

- How often will the models be used if we build them?
- Compute #lookups by levels
  - High #lookups at internal levels, but many of them are negative
  - Higher fraction of positive lookups at lower levels

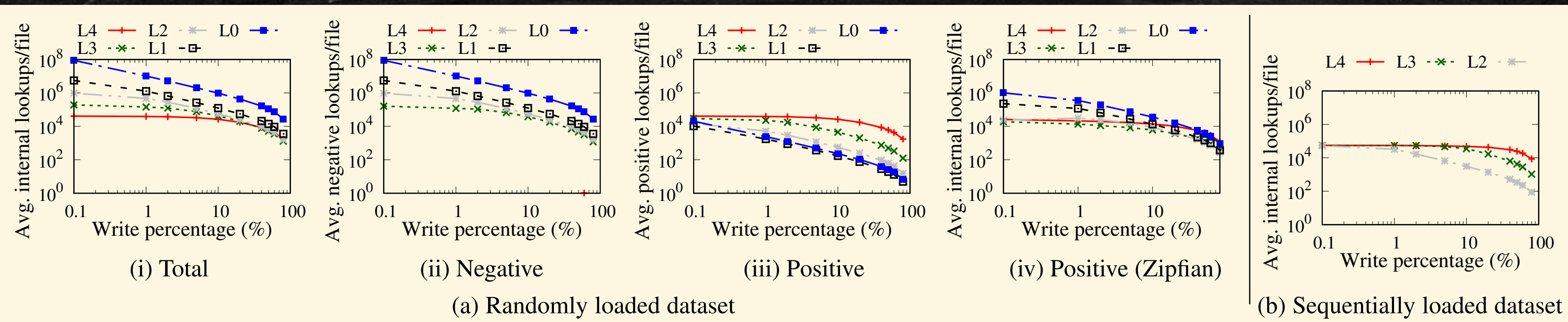
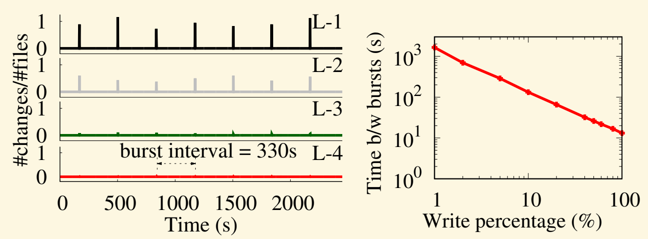


Figure 4: **Number of Internal Lookups Per File.** *(a)(i) shows the average internal lookups per file at each level for a randomly loaded dataset. (b) shows the same for sequentially loaded dataset. (a)(ii) and (a)(iii) show the negative and positive internal lookups for the randomly loaded case. (a)(iv)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

Are Learned Indexes a Good Match?

- Does it make sense to build models for an entire level (i.e., all SSTables in a level)?
- Changes happen in bursts
- Fewer changes on lower levels, but still too many for write-heavy workloads



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

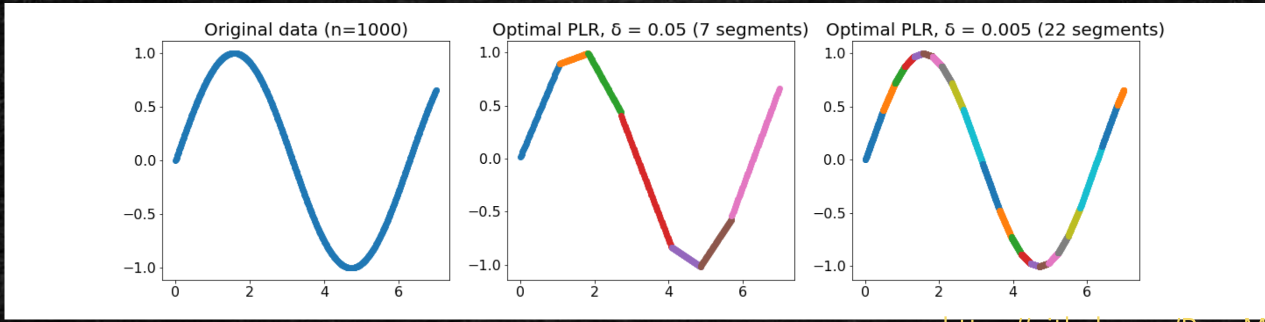
Figure 5: **Changes at Levels.** *(a) shows the timeline of file creations and deletions at different levels. Note that #changes/#files is higher than 1 in L_1 as there are more creations and deletions than the number of files. (b) shows the time between bursts for L4 for different write percentages.*

Learning Guidelines

- Favor learning files at lower levels
- Wait before learning a file
  - To ensure it is not transient
- Don't neglect higher levels
- Be workload- and data-aware
- Do not learn levels for write-heavy workloads

Bourbon Design

- Use Piecewise Linear Regression (PLR)
  - Greedy-PLR: add one point at a time and switch to a new line if error too high
  - Using the model: binary search to find the line segment, followed by a calculation for the offset, and search around the offset
    - (Radix Spline Index might work better here)
- Helps that WiscKey stores values separately
  - Otherwise offsetting wouldn't work as well



https://github.com/RyanMarcus/plr

Bourbon Design

- Level vs file learning
  - Level learning only beneficial if the workload has no writes

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.*

Bourbon Design

- How long to wait before learning?
  - Time to learn a file (approx. 4MB) around 40ms
  - Wait for 50ms (is 2-competitive compared to the optimal)

- Cost-benefit analysis to decide whether to learn
  - Lower levels may not have enough lookups to make it worthwhile
  - Use statistics to make a decision

Bourbon Design

- Putting it all together
  - It does model lookup before bloom filter
  - Unclear why -- LSMs do BFs before (in general) searching IB (AFAIK)

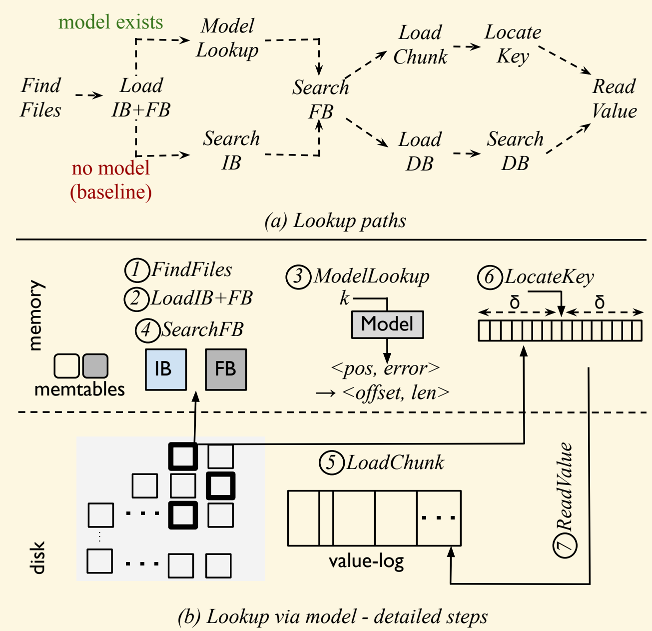
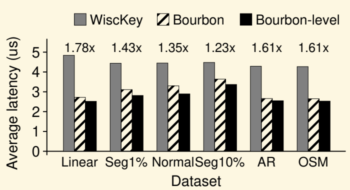


Figure 6: **Bourbon Lookups.** *(a) shows that lookups can take two different paths: when the model is available (shown at the top), and when the model is not learned yet and so lookups take the baseline path (bottom); some steps are common to both paths. (b) shows the detailed steps for a lookup via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

Evaluation

- Questions to answer
  - 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)
  - How does Bourbon perform with range queries? (§5.3)
  - In the presence of writes, how does Bourbon's cost-benefit analyzer perform compared to other approaches that always or never re-learn? (§5.4)
  - Does Bourbon perform well on real benchmarks? (§5.5)
  - Is Bourbon beneficial when data is on storage? (§5.6)
  - Is Bourbon beneficial with limited memory? (§5.7)
  - What are the error and space tradeoffs of Bourbon? (§5.8)

- 1.23x to 1.78x performance improvements across the datasets



(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.

Outline

- Log-structured Merge Trees
- WiscKey → Bourbon
- **Discussion**