From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees

Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau
Data Lookup

Data lookup is important in systems

How do we perform a lookup given an array of data?

  Linear search

What if the array is sorted?

  Binary search

What if the data is huge?
Data Structures to Facilitate Lookups

Assume sorted data

Traditional solution: build specific data structures for lookups
   B-Tree, for example
   Record the position of the data

What if we know the data beforehand?
Bring Learning to Indexing

Lookups can be faster if we know the distribution
  The model \( f(\cdot) \) learns the distribution

Leaned Indexes

Time Complexity – \( O(1) \) for lookups

Space Complexity – \( O(1) \)
  Only 2 floating points – slope + intercept

\[ f(x) = 0.5x - 50 \]

\[ x = 100 \rightarrow f(x) = 0 \]

---

Kraska et al. The Case for Learned Index Structures. 2018
Challenges to Learned Indexes

How to efficiently support insertions/updates?
  Data distribution changed
  Need re-training, or lowered model accuracy
How to integrate into production systems?

Key
\[ f(x) = 0.5x - 50 \]

\[
\begin{array}{ccccccc}
100 & 101 & 102 & 103 & 104 & 106 & ... \\
200 & 202 & 204 & 206 & ... \\
300 & 302 & 304 & 306 & 350 & 400 \\
\end{array}
\]
Bourbon

A Learned index for LSM-trees
Built into production system (WiscKey)
Handle writes easily

LSM-tree fits learned indexes well
Immutable SSTables with no in-place updates

Learning guidelines
How and when to learn the SSTables

Cost-Benefit Analyzer
Predict if a learning is beneficial during runtime

Performance improvement
1.23x – 1.78x for read-only and read-heavy workloads
~1.1x for write-heavy workloads
**LevelDB**

Key-value store based on LSM
- 2 in-memory tables
- 7 levels of on-disk SSTables (files)

**Update/Insertion procedure**
- Buffered in MemTables
- Merging compaction
- From upper to lower levels
- No in-place updates to SSTables

**Lookup procedure**
- From upper to lower levels
- Positive/Negative internal lookups

**Diagram**
- Memory
  - L0 (8M)
  - L1 (10M)
  - L2 (100M)
  - L3 (1G)
  - ...... (up to L6 (1T))
- SSTables
- MemTables
  - K_{min}
  - K_{max}
Learning Guidelines

Learning at SSTable granularity
   No need to update models
   Models keep a fixed accuracy

Factors to consider before learning:
   1. Lifetime of SSTables
      How long a model can be useful
   2. Number of Lookups into SSTables
      How often a model can be useful
Learning Guidelines

1. Lifetime of SSTables
   How long a model can be useful

Experimental results
   Under 15Kops/s and 50% writes
   Average lifetime of L0 tables: 10 seconds
   Average lifetime of L4 tables: 1 hour
   A few very short-lived tables: < 1 second

Learning guideline 1: Favor lower level tables
   Lower level files live longer

Learning guideline 2: Wait shortly before learning
   Avoid learning extremely short-lived tables
Learning Guidelines

2. Number of Lookups into SSTables
   How often a model can be useful
   Affected by various factors
      Depending on workload distribution, load order, etc.
      Higher level files may serve more internal lookups

Learning guideline 3: Do not neglect higher level tables
   Models for them may be more often used

Learning guideline 4: Be workload- and data-aware
   Number of internal lookups affected by various factors
Learning Algorithm: Greedy-PLR

Greedy Piecewise Linear Regression
From Dataset $D$
Multiple linear segments $f(\cdot)$
$\forall (x, y) \in D, |f(x) - y| < error$
$error$ is specified beforehand
In bourbon, we set $error = 8$

Train complexity: $O(n)$
Typically ~40ms

Inference complexity: $O(\log \#\text{seg})$
Typically <1$\mu$s

Xie et al. Maximum error-bounded piecewise linear representation for online stream approximation. 2014
Bourbon Design

Bourbon: Build upon WiscKey
   WiscKey: key-value separation built upon LevelDB
   (Key, value_addr) pair in the LSM-tree
   A separate value log

Why WiscKey?
   Help handle large and variable sized values
   Constant-sized KV pairs in the LSM-tree
   Prediction much easier
Bourbon Design

Find File → Load Index Block → Model Lookup → Load & Search Chunk → Read Value

IB | DB | DB | DB | … | DB

SSTable

L0          L1          L2

WiscKey (Baseline) path ~4μs

Bourbon (model) path 2~3μs
Goal: Minimize total CPU time
   A balance between always-learn and no-learn

Learn!

Estimated benefit
   Baseline path lookup time
   Model path lookup time
   Number of lookups served

Estimated cost
   Table size
Effectiveness of Cost-Benefit Analyzer

Learn most/all new tables at low write percentages
  • Reach a better foreground latency than offline learning

Limit learning at high write percentages
  • Reduce learning time and have a good foreground latency

Minimal total CPU cost in all scenarios
Evaluation

Various micro and macro benchmarks

- Dataset
- Load order
- Request distribution
- Range queries
- YCSB
- SOSD
- On-disk database

Database resides in memory
- Reduce data access time
- Better show benefits in indexing time
- Come back to this condition later
Can Bourbon adapt to different datasets?

Micro benchmark: datasets

- 4 synthetic datasets: linear, normal, seg1%, and seg10%
- 2 real-world datasets: AmazonReviews and OpenStreetMapNY

Uniform random read-only workloads

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Data</th>
<th>#Seg</th>
<th>%Seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>64M</td>
<td>900</td>
<td>0%</td>
</tr>
<tr>
<td>Seg1%</td>
<td>64M</td>
<td>640K</td>
<td>1%</td>
</tr>
<tr>
<td>Normal</td>
<td>64M</td>
<td>705K</td>
<td>1.1%</td>
</tr>
<tr>
<td>Seg10%</td>
<td>64M</td>
<td>6.4M</td>
<td>10%</td>
</tr>
<tr>
<td>AR</td>
<td>33M</td>
<td>129K</td>
<td>0.39%</td>
</tr>
<tr>
<td>OSM</td>
<td>22M</td>
<td>295K</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

Bourbon performs better with lower number of segments
Reach 1.6x gain for two real-world datasets with 1% segments
Performance with different request distributions?

Micro benchmark: request distribution

Read-only workloads
Sequential, zipfian, hotspot, exponential, uniform, and latest

Bourbon improves performance by ~1.6x
Regardless of request distributions
Can Bourbon perform well on real benchmarks?

Macro benchmark: YCSB

6 core workloads on YCSB default dataset

Bourbon Improves reads without affecting writes

Bourbon’s gain holds on real benchmarks
Bourbon improves reads without affecting writes
Is Bourbon beneficial when data is on storage?

Performance on fast storage
Data resides on an Intel Optane SSD
5 YCSB core workloads on YCSB default dataset

Bourbon can still offer benefits when data is on storage
Will be better with emerging storage technologies
Conclusion

Bourbon

Integrates learned indexes into a production LSM system
Beneficial on various workloads
Learning guidelines on how and when to learn
Cost-Benefit Analyzer on whether a learning is worthwhile

How will ML change computer system mechanisms?
Not just policies
Bourbon improves the lookup process with learned indexes
What other mechanisms can ML replace or improve?
Careful study and deep understanding are required
Thank You for Watching!

The ADvanced Systems Laboratory (ADSL)
https://research.cs.wisc.edu/wind/

Microsoft Gray Systems Laboratory
https://azuredata.microsoft.com/