Outline

• Briefly discuss Learned Indexes

• LSM Trees
Learned Index Structures

• What are the arguments in favor of learned index structures?

• Why is an index a “model”?

• What does Neumann’s blog say?
Learned Index Structures

• What are the arguments in favor of learned index structures?
  – B+ trees, hash tables: distribution agnostic
  – GPU/TPU: efficient for regression model

• Why is an index a “model”?

• What does Neumann’s blog say?
Learned Index Structures

• What are the arguments in favor of learned index structures?
  – B+ trees, hash tables: distribution agnostic
  – GPU/TPU: efficient for regression model

• Why is an index a “model”?
  – Index maps key value to position
  – Regression model does the same

• What does Neumann’s blog say?
Learned Index Structures

• What are the arguments in favor of learned index structures?
  – B+ trees, hash tables: distribution agnostic
  – GPU/TPU: efficient for regression model

• Why is an index a “model”?
  – Index maps key value to position
  – Regression model does the same

• What does Neumann’s blog say?
  – Use a simple regression model
Learned Index Structures

Figure 2: Indexes as CDFs
Discussion

(in class)
Outline

• Briefly discuss Learned Indexes

• LSM Trees

Slides based on
Monkey: Optimal Navigable Key-Value Store,
Dayan, Athanassouli, Idreos,
SIGMOD’2017
Reading for Monday!!
Motivation

• Sorted arrays = best for reads
• Unsorted log file = best for writes
• B+ trees = good for read, so-so for write

• LSM trees = optimize the writes
• Notice:
  – Primary (clustered) index only
  – Key/value stores, but also relational DBs
More Motivation

Index for one attribute:
Person.name

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
<th>Carl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More Motivation

Index for one attribute:
Person.name

Index for entire table:
Person(name, age, city)
More Motivation

Index for one attribute: Person.name

Index for entire table: Person(name,age,city)

Index for entire db: Person, Dept, Project, ...

<table>
<thead>
<tr>
<th>Person</th>
<th>Accounting</th>
<th>4th floor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Sales</td>
<td>2nd floor</td>
</tr>
<tr>
<td>Dept</td>
<td>Alice</td>
<td>22 Seattle</td>
</tr>
<tr>
<td>Dept</td>
<td>Bob</td>
<td>53 Kent</td>
</tr>
<tr>
<td>Dept</td>
<td>Carl</td>
<td>37 Pasco</td>
</tr>
<tr>
<td>Project</td>
<td>Compiler</td>
<td>$55000</td>
</tr>
<tr>
<td>Project</td>
<td>Database</td>
<td>$77000</td>
</tr>
<tr>
<td>Project</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
More Motivation

Index for one attribute: Person.name

Index for entire table: Person(name, age, city)

Index for entire db: Person, Dept, Project,…

E.g. MySQL on RocksDB using MyRocks
Three Main Ideas for Writes

1. Store writes in a buffer in main memory
   When full: spill to disk

2. Spilling to disk (instead of a B+ tree):
   Sort and write to a sorted file.

3. When too many sorted files:
   Merge them to a larger sorted file
LSM Trees

Files on disk

Write buffer

Main memory
LSM Trees

Main memory

Write buffer

Level 1

Files on disk
LSM Trees

Main memory

Write buffer

Files on disk

Level 1

Level 2
LSM Trees

Main memory

Write buffer

Files on disk

Level 1

Level 2

Level 3
LSM Trees

Files on disk

Level 1

Level 2

Level 3

... Level L
LSM Trees

Level 0

Write buffer

Main memory

Files on disk

Level 1
Level 2
Level 3

…

Level L
LSM Trees

Write buffer

Main memory

Level 0

Files on disk

Level 1

Level 2

Level 3

... 

Level L

T = size ratio between levels
Discussion

• Spilling to next level is a bulk operation; inserts a large number of values

• Better amortized cost than inserting those values one by one into a B+ tree

• Typically done by offline process
Read

• To read a key, we need to search it at all levels

• Cost is worse than B+ tree

• Three ideas to speedup reads (next)
Three Main Ideas for Reads

1. Bloom filter for each level

2. Fenceposts in main memory for each level

3. Read single block for each level, do binary search
Reading

<table>
<thead>
<tr>
<th>Level</th>
<th>Main Memory</th>
<th>Secondary Storage</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>buffer</td>
<td></td>
<td>$P \cdot B \cdot T^0$</td>
</tr>
<tr>
<td>1</td>
<td>filters</td>
<td></td>
<td>$P \cdot B \cdot T^1$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>$P \cdot B \cdot T^2$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>fence pointers</td>
<td></td>
<td>$P \cdot B \cdot T^L = N \cdot \frac{T-1}{T}$</td>
</tr>
</tbody>
</table>

$$M = M_{buffer} + M_{filters} + M_{pointers}$$

Total = $N$
Updates, Deletes

• Never!

• Instead, invalidate the record, and insert a new record if update
Next

• How do we optimize the main memory:
  – Write buffer
  – Bloom filters
  – Fence pointers

• Merge policy
  – Tiering or
  – Leveling
Optimizing Bloom Filters

Most memory used by Bloom filters

• Common practice:
  – Ensure the same FPR for all levels
  – FPR constant, space m increases by factor T

\[ \text{FPR} = e^{-\frac{m}{n} \cdot (\ln^2 2)} \quad , \quad n = \# \text{items at given level} \]
Optimizing Bloom Filters

Most memory used by Bloom filters

• Common practice:
  – Ensure the same FPR for all levels
  – FPR constant, space m increases by factor T

• Paper observes:
  – Cost per level is the same: reading 1 block
  – Space increases but benefit is constant!
  – Keep space constant, FPR increases by factor T
Merge Policy

• Tiering (write optimized)
  – Flush main memory buffer sorted to disk
  – Accumulate multiple sorted files/level
  – When more than T sorted files: merge them and add 1 file to the next level
Merge Policy

• **Tiering (write optimized)**
  – Flush main memory buffer sorted to disk
  – Accumulate multiple sorted files/level
  – When more than T sorted files: merge them and add 1 file to the next level

• **Leveling (read-optimized)**
  – Merge-sort main memory with level 1
  – When a level becomes too large, move it to the next level by sorting
Size Ratio: $T = 3$

 Merge Policies

**Tiering**

```
  □ □ □
  □ □ □
  ...  ...
```

**Leveling**

```
  □
  □
  □
  □
  ...  ...
```
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling

merge
Merge Policies

Size Ratio: \( T = 3 \)
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling

merge

...
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling

\[ T = 3 \]
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling

merge
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling
Size Ratio: $T = 3$

**Merge Policies**

**Tiering**

**Leveling**

...
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling
Size Ratio: $T = 3$

Merge Policies

Tiering

Leveling

What happens when $T \to \infty$?
Merge Policies

Tiering

... Leveling

Then \( L = 1 \)

What happens when \( T \to \infty \) ?
Then $L = 1$

What happens when $T \to \infty$?
Merge Policies

Tiering

Leveling

...  

A log file!

A sorted file!

Then $L = 1$

What happens when $T \to \infty$?
Recap: Merge-Sort

- Problem: Sort a file of size $B$ with memory $M$
- Will discuss only 2-pass sorting, for when $B \leq M^2$
Merge-Sort: Step 1

• Phase one: load M pages in memory, sort
Merge-Sort: Step 2

- Merge $M - 1$ runs into a new run
- Result: runs of length $M (M - 1) \approx M^2$

Assuming $B \leq M^2$, we are done
Merge-Sort

• Cost:
  – Read+write+read = 3B(R)
  – Assumption: B(R) <= M^2

• Other considerations
  – In general, a lot of optimizations are possible
Summary

• LSM trees: optimized for write-intensive applications
• Three ideas for writes:
  – Memory buffer, spill to disk, multiple levels
• Three ideas for reads:
  – Bloom filters, fence posts, binary search
• When T is very large: log or sorted file