DB indexing

COS 518: Advanced Computer Systems
Lecture 7
Michael Freedman

Basic row-based storage

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Id</td>
<td>NAME</td>
<td>Age</td>
<td>Gender</td>
<td>Birthday</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>CHAR(32)</td>
<td>INT</td>
<td>SMALLINT</td>
<td>DATE</td>
<td></td>
</tr>
</tbody>
</table>

= 50

How to efficiently find data?

• Types of queries
  – Exact match: id = 139856151
  – Predicate scans: All entries with age > 80

• Option 1: Full scan
• Option 2: Use an index!
Use a balanced binary tree?

- Easy to do in memory
- Except how do you store on disk?
  - Let’s say 1M entries
  - 20 levels of a binary tree
  - Don’t want 20 random disk seeks (say, ~200ms)

Strawman: Leverage locality on disk

- Can collocate many pointers on disk
  - Databases typically read data in “page” granularities (4-16KB per page)
- But keeping a binary tree “balanced” requires many rotations (e.g., red-black tree rotations)

B-Tree: Disk-aware lookup tree

- Each tree node sized for disk page
- Internal nodes maintain pointers (to subtrees) and keys
- Keys serve as bookends to subtrees
- Keys also include pointer to underlying row in DB
- Typically maintain sparse nodes (say, 50% empty) for cheaper insertion/deletion

Challenge for randomized workloads

- What if we could just store 2 pages in memory?
- Insert new keys in order: 10, 19, 11, 20, 13, 22, …
**Insert throughput as F(table size)**

Insert batch size: 1, Cache: 4 GB memory

<table>
<thead>
<tr>
<th>Dataset size (millions of rows)</th>
<th>Insert rate [rows / second]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10,000</td>
</tr>
<tr>
<td>2</td>
<td>15,000</td>
</tr>
<tr>
<td>3</td>
<td>20,000</td>
</tr>
<tr>
<td>4</td>
<td>20,000</td>
</tr>
<tr>
<td>5</td>
<td>20,000</td>
</tr>
</tbody>
</table>

**LSM Trees + LevelDB**

- Goal #1: High-throughput writes (inserts/updates/deletes)
- Goal #2: Support index size >> memory size
- Observation: Writes to disk should be batched for high throughput (either SSD or HDD)
- Observation: Sorting/indexing in memory is fast
- Main insight: Don’t try to maintain single data structure with in-place ordering
Collect writes and batch in memory

- Collect writes in memory
  - Can maintain sorted list in memory
  - Can update in-place (overwrite, delete, etc.)

- Disk is immutable
  - Once written to disk, not modified in-place
  - Queries will need to find all records and merge
  - Deletes are simply “tombstone records”

Spill From Memory To Disk

- As memory budget fills up, spill them to disk
  - Write out entire sorted string table
  - Write out a subtree, then remove and prune it in memory

- Each dump forms a “run” ordered by write time

LSM Trees

- SSTable: set of arbitrary, sorted key-value pairs

- LSM Trees: Write to memory, then flush to disk

LSM Trees

- LSM Trees: Write to memory, then flush to disk
Problem & solution

• Index lookups can traverse many SSIndexes
  – Esp. with range scan vs. exact-match lookup
  – Other optimizations trade-off memory for additional disk lookups, e.g., bloom filter vs. SSIndex

• Idea
  – Merge and compact SSTables in background

Log-structured MERGE Tree

• Merge updates disk data structures in background
  – Input: K overlapping sorted SSTables from A-Z at level L
  – Output: Disjoint sorted SSTables files at level L+1

• Compaction also occurs as part of the merges
  – Merges multiple updates into one
  – Deletes tombstoned records
  – Recovers storage from merged updates and deleted values

• Can occur very efficiently by sequential reads from each SSTable and writing to output files (why? Hint: all sorted)

LSM Tree in Level DB