DB indexing

COS 518: *Advanced Computer Systems*
Lecture 7

Michael Freedman

Basic row-based storage

8	32	4	2	4	= 50
Id BIGINT	Name CHAR(32)	Age INT	Gender SMALLINT	Birthday DATE	

2

Basic row-based storage

	8	32	4	2	4	= 50
0	Id BIGINT	Name CHAR(32)	Age INT	Gender SMALLINT	Birthday DATE	
50	Id BIGINT	Name CHAR(32)	Age INT	Gender SMALLINT	Birthday DATE	
100	Id BIGINT	Name CHAR(32)	Age INT	Gender SMALLINT	Birthday DATE	
150	Id BIGINT	Name CHAR(32)	Age INT	Gender SMALLINT	Birthday DATE	

3

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!

4

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)

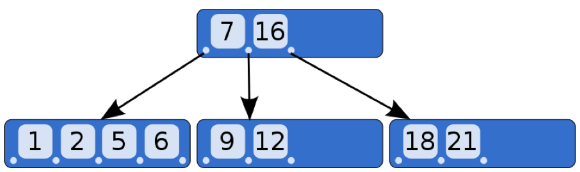
5

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)

6

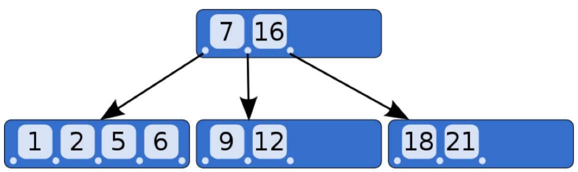
B-Tree: Disk-aware lookup tree



- 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

7

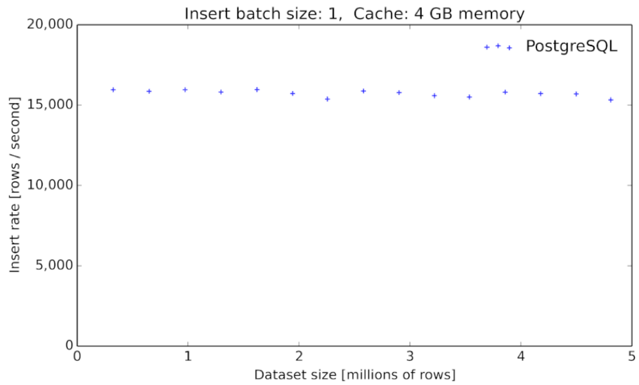
Challenge for randomized workloads



- What if we could just store 2 pages in memory?
- Insert new keys in order: 10, 19, 11, 20, 13, 22, …

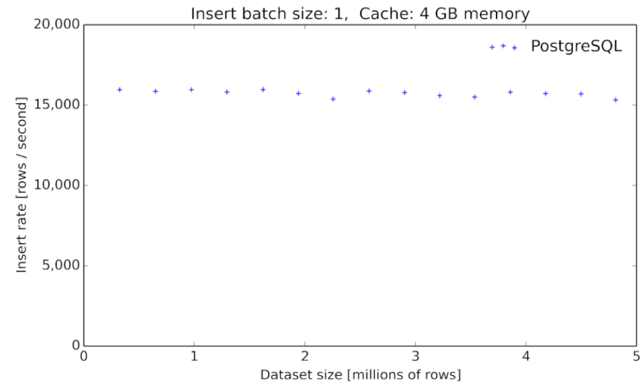
8

Insert throughput as F(table size)



9

Insert throughput as F(table size)



10

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

11

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

12

Collect writes and batch in memory

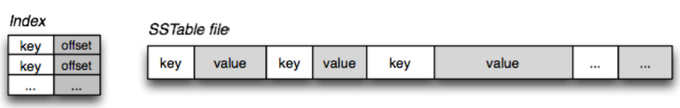
- Collect writes in memory
  - Can maintain sorted list in memory
  - Can update in-place (overwrite, delete, etc.)
- Disk is immutable
  - Once written to disk, not modified in-place
  - Queries will need to find all records and merge
  - Deletes are simply “tombstone records”

Spill From Memory To Disk

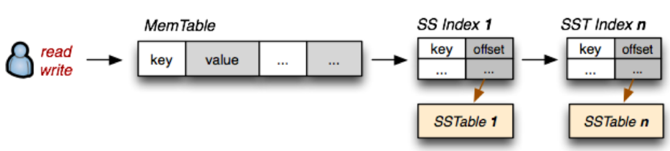
- As memory budget fills up, spill them to disk
  - Write out entire sorted string table
  - Write out a subtree, then remove and prune it in memory
- Each dump forms a “run” ordered by write time

LSM Trees

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



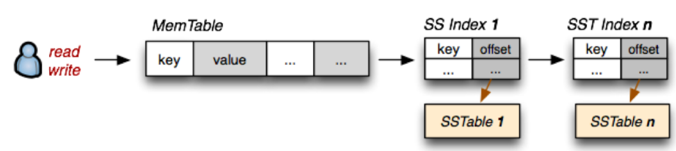
- LSM Trees: Write to memory, then flush to disk



LSM Trees

1. **On-disk `SSTable` indexes are always loaded into memory**
2. **All writes go directly to the `MemTable` index**
3. **Reads check the MemTable first and then the SSTable indexes**
4. Periodically, the MemTable is flushed to disk as an SSTable

- 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