BigData Algorithms: LSMTree

Alfred Thompson

Nov 15, 2016

Origins

- Promulgated in the paper "The log-structured merge-tree (LSM tree)"
  - by Patrick O'Neil et al (1996 Acta Infomratica, vol 33, issue 4)
- LSM trees show up on multiple BigData/FastData platforms
- Need a way to optimize read use cases on databases sustaining heavy write loads
- The classic RDBMS approach of adding an index to speed up reads makes writes more expensive
- Can we use the memory hierarchy in a different way to get faster reads and writes?

Use Case

- History log of events from channels sustaining heavy write activity
- Need to select activity streams for specific channels

Scenario

- Transactions log of financial activity over all accounts
- Need to present financial history to account holders (eg, online banking)
- Need to searches for fraud indicators for a subset of accounts or merchants

More Usage

This sort of query is usually enabled with indexes in the relational model

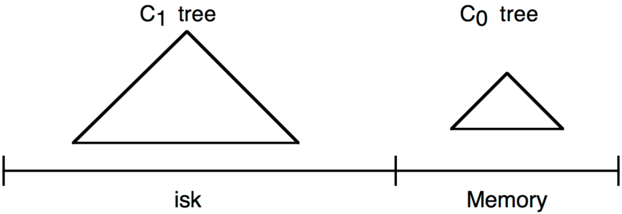
```
SELECT * FROM history
WHERE history.account_id = %custacctid
AND history.timestamp > %custdatetime
```

- Without an index the query requires a direct search of all rows
- Index of `account_id` column provides nice speedup, but the query may benefit from concatenated index of `(account_id, timestamp)`
- Reads get faster but at a huge cost to writes
- In a write heavy use case financial transactions log, this is untenable

Ops

- `CRUD`
  - favor `CUD` over `R` (but keep `R` plenty fast)
- Store keys and values as byte arrays
- key-value store has 2 (or more) levels:
  - C_0 is an ordered map of key-values, entirely memory resident
  - C_1 a tree of key-values, residing on disk
- Other things
  - C_0 is assumed to be much faster, and much smaller than C_1
  - Frequently accessed ("hot") pages will be stored in memory ("buffer cache")
  - In [modern] practice, C_1 is distributed on a grid

Implementation



Implementation Detail

Writes

- Write to serial log (eg, HBase WAL)
- Write value to C_0

Reads

- Check for value in C_0
- If not found, look in C_1
  - use a bloom filter (Burton Howard Bloom, 1970) to determine which files in C_1 to look in
  - probabilistic approach gives "maybe" answer using a bitset
  - answers = possibly in set, definitely not in set

Spills

- When C_0 fills, asynchronously dump segments of C_0 to C_1
- External file merge new segments into C_1

More Implementation

Crashes

- Play back the serial log to cache, while suspending reads
- Handle writes as usual
- All done, turn reads back on

Bulk load

- Generate the C1 files to avoid thrashing cache on bulk load
  - LONG operation

Variation

LDAP is almost the inverse of the financial log use case

- Read heavy
- Only occasional CUD

Lightning Memory-mapped Database (LMDB)

- Pluggable with BerkeleyDB
- Very small, very fast transactional database

Real World Implementations

- BigTable
- HBase
- RocksDB
- LevelDB
- Indeed LSMTree
- WireShark (MongoDB)

Summary

The LSMTree is a classic algorithm that efficiently supports write heavy workloads while also supporting fast reads

Use Cases

- financial activity
- generic history logs
- distributed session management
- anything else where writes predominate over reads

The implementation is tweaked in modern implementations by replacing the in memory tree with a map

- Approximates O(1) lookup for in memory store
  - Actually O(log_2 n)

Many robust open source implementations that scale across a range of use cases:

- BigData/FastData
- Embedded/Mobile
- Distributed Systems