A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases

By Carl Otto and Rickard
Content

- Motivation
- Background
- Secondary indexing taxonomy
- Methods for testing
- Results
- Conclusion
Motivation

- Many applications require queries on non-primary attributes
- No head-to-head comparison of various secondary indexing methods exist
- To provide a guideline for choosing a secondary index
- In regards to Big Data, we need efficient systems that can scale and handle a huge amount of read and write requests
Background

- NoSQL
- Log-Structured Merge-Tree (LSM)
- Secondary index
NOSQL

- Response to new challenges and opportunities
  - cheap storage, high volume
- Different types
- Fast throughput, fast lookup on primary key
Log-Structured Merge-Tree

- key-value data structure optimized for write-heavy workloads
- consists of an in-memory component C0 and several on disks components C1,C2,....Cn.

<table>
<thead>
<tr>
<th>In-memory components</th>
<th>On disk components</th>
</tr>
</thead>
<tbody>
<tr>
<td>new records and deletes inserted here</td>
<td>updates are performed out-of-place -&gt; sequential writes</td>
</tr>
<tr>
<td>updates are performed in-place</td>
<td>implemented as MemTables in LevelDB</td>
</tr>
<tr>
<td>implemented as MemTables in LevelDB</td>
<td>Implemented as SSTables in LevelDB</td>
</tr>
</tbody>
</table>
Log-Structured Merge-Tree 2.

- The Rolling Merge Process
Secondary Indexing

- Need for queries on information not found explicitly by primary key
- A mapping from a specific value of a secondary attribute to a list of primary keys whose entries contain that value.
- Similar to inverse index found in information retrieval.
- Example: given a set of tweets with unique tweetID (primary key) and a number of different users (secondary attribute): index will store (UID1 -> {TID2, TID5,...}, UID2 -> {TID1, TID3,...}, ...)
- Secondary indexing structures must be maintained during writes -> consistency between secondary indexes and data tables
Stand Alone Index

- Used by most NoSQL architectures
- fast query response time
- LSM table
- Support Secondary Indexing by having a separate table for indices
- Different types
  - Eager
  - Lazy
  - Composite

(a) TweetsData - **Primary table**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>{&quot;UserID&quot;: u1, &quot;text&quot;: &quot;t1 text&quot;}</td>
</tr>
<tr>
<td>t2</td>
<td>{&quot;UserID&quot;: u1, &quot;text&quot;: &quot;t2 text&quot;}</td>
</tr>
<tr>
<td>t3</td>
<td>{&quot;UserID&quot;: u2, &quot;text&quot;: &quot;t3 text&quot;}</td>
</tr>
<tr>
<td>t4</td>
<td>{&quot;UserID&quot;: u2, &quot;text&quot;: &quot;t4 text&quot;}</td>
</tr>
</tbody>
</table>

(b) UserIndex

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>u1</td>
<td>[t2, t1]</td>
</tr>
<tr>
<td>u2</td>
<td>[t4, t3]</td>
</tr>
</tbody>
</table>
Stand Alone Index

- **Eager**
  - Each write updates index table/tree
  - Write becomes slower
    - Requires read + write
  - Eager Index shows exponential write costs due to updating the posting list for every write. Needs to read existing posting list every time, then it writes and invalidates the previous one.
  - Consumes less space than Lazy
  - Used in MongoDB, CouchDB, DynamoDB, Riak, Aerospike
Stand Alone Index
- Lazy
- Just appends new index
- A drawback of the lazy update strategy is that reads on index tables become slower, because they have to merge the posting lists at query time
- Used in Cassandra
Stand Alone Index
- Composite

- Each entry consists of secondary index and primary index
- Prefix search on secondary key
- Writes and compactions are faster than “Lazy,” but secondary attribute lookup may be slower as it needs to perform a range scan on the index table
- Used in AsterixDB, Spanner
- Secondary attribute lookup is slower
  - Needs to perform a range scan on the index table
Embedded Secondary Indexes

- No separate secondary index structure
- Secondary attribute information is stored inside the data blocks
- Superior write throughput and are more space efficient
- Made out of two indexes
  - Bloom filter
  - Zone map

(a) Bloom filters

(b) Zone maps

(b) Zone maps and bloom filters in Embedded Index
Bloom filters

- Independent hash-functions used to indicate existence of key.
- Does not give false negatives.
- Gives false positives: UID9 might appear positive in block with TID11, even though User9 did not send Tweet11.
- Drawback of bloom filter index is that it can only support lookup queries
- Range queries is not supported.

Zone Map

- Typically store min- and max-value of column
- Used for both range and look-up queries
- only useful when the incoming data is time-correlated
- An attribute is called time-correlated if its value for a record is highly correlated with the record’s insertion timestamp
- Used in Oracle, Netezza, Spanner and AsterixDB
Results
Evaluation Method

- Performed a study on secondary indexing techniques for NoSQL databases
- Used a synthetic data (tweet) generator
  - Based on real-world seed data
- Static workload
  - Data first inserted, then queried
- Mixed workload
  - PUT-operations are updates on existing values

(a) Static Workload

<table>
<thead>
<tr>
<th>Operation</th>
<th>PUT</th>
<th>GET</th>
<th>LOOKUP</th>
<th>RANGE LOOKUP</th>
<th>UserID (no of users)</th>
<th>Creation Time (in minutes)</th>
<th>Top-K</th>
</tr>
</thead>
<tbody>
<tr>
<td>180 million</td>
<td>100K</td>
<td>15K (each index)</td>
<td>50K (each index)</td>
<td>10, 100</td>
<td>1, 100</td>
<td>10, 100, No Limit</td>
<td></td>
</tr>
</tbody>
</table>

(b) Mixed Workloads

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>Operation Frequency Ratios</th>
<th>n</th>
<th>Top-K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write heavy</td>
<td>80% GET 15% LOOKUP 5% Update 0%</td>
<td>50 million</td>
<td>10</td>
</tr>
<tr>
<td>Read heavy</td>
<td>20% GET 70% LOOKUP 10% Update 0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update heavy</td>
<td>40% GET 15% LOOKUP 5% Update 40%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results 2: different work-loads

(a) Write Heavy Workload

(b) Read Heavy Workload

(c) Update Heavy Workload
Conclusions

● Eager Index bad.

Guideline for selecting secondary index:

1. **Index Attribute**
   - Time-Correlated?
     - Yes: **Embedded Index**
       - e.g., Index on timestamp/date for any application
     - No: **Space Constraint?**
       - Yes: **Embedded Index**
         - e.g., mobile software which maintains a key-value store
       - No: **High ratio of writes to lookups?**
         - Yes: **Embedded Index**
           - e.g., Wireless Sensor Network [9, 16]
         - No: **Sensitive to Top-K?**
           - Yes: **Stand-alone Index With Lazy Updates**
             - e.g., Facebook or Twitter to store user posts [12, 17]
           - No: **Stand Alone Composite Index**
             - e.g., IBM/Teradata type any General Data Analytics Platform [25, 29].

Questions?