Workload Analysis of Key-Value Stores on Non-Volatile Media

Vishal Verma (Performance Engineer, Intel)
Tushar Gohad (Cloud Software Architect, Intel)
Outline

- KV Stores – What and Why
- Data Structures for KV Stores
- Design Choices and Trade-offs
- Performance on Non-Volatile Media
- Key Takeaways
Outline

- KV Stores – What and Why
- Data Structures for KV Stores
- Design Choices and Trade-offs
- Performance on Non-Volatile Media
- Key Takeaways
What are KV Stores

- Type of NoSQL database that uses simple key/value pair mechanism to store data
- Alternative to limitations of traditional relational databases (DB):
  - Data structured and schema pre-defined
  - Mismatch with today’s workloads.
  - Data growth in large and unstructured
  - Lots of random writes and reads.
- NoSQL brings flexibility as application has complete control over what is stored inside the value
What are KV Stores

- Key in a key-value pair must (or at least, *should*) be unique. Values identified via a key, and stored values can be numbers, strings, images, videos etc
- API operations: `get(key)` for reading data, `put(key, value)` for writing data and `delete(key)` for deleting keys.
- **Phone Directory** example:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>(123) 456-7890</td>
</tr>
<tr>
<td>Kyle</td>
<td>(245) 675-8888</td>
</tr>
<tr>
<td>Richard</td>
<td>(787) 122-2212</td>
</tr>
</tbody>
</table>
KV Stores: Benefits

- **High performance**: Enable fast location of object rather than searching through columns or tables to find an object in traditional relational DB.

- **Highly scalable**: Can scale over several machines or devices by several orders of magnitude, without the need for significant redesign.

- **Flexible**: No enforcement of any structure to data.

- **Low TCO**: Simplify operations of adding or removing capacity as needed. Any hardware or network failure do not create downtime.
Outline

- KV Stores – What and Why
- Design Choices and Trade-offs
- Performance on Non-Volatile Media
- Key Takeaways
Design Choice: B-Tree

- Internal nodes (green) – pivot keys
- Leaf nodes (blue) – data records (KV)
- Query time proportional to height
  - Logarithmic time
- Insertion / Deletion
  - Many random I/Os to disk
  - May incur rebalance – read and write amplification
  - Full leaf nodes may be split – space fragmentation
B-Tree: Trade-offs

- Example DB engines: BerkeleyDB, MySQL InnoDB, MongoDB, WiredTiger
- Good design choice for read-intensive workloads
- Trades read performance for increased Read / Write / Space Amplification
  - Nodes of B-Trees are on-disk blocks – aligned IO = space-amp
  - Compression not effective – 16KB block size, 10KB data compressed to 5KB will still occupy 16KB
  - Byte-size updates also end up in page size read / writes
## Performance Test Configuration

<table>
<thead>
<tr>
<th>System Config</th>
<th>Non-Volatile Storage</th>
<th>OS Config</th>
<th>Key-Value Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Intel(R) Xeon(R) CPU E5-2618L v4, 2.20GHz, HT disabled, 20 cores</td>
<td>Intel ® P4500 SSD (4TB)</td>
<td>Distro: Ubuntu 16.04.1</td>
<td>WiredTiger 3.0.0</td>
</tr>
<tr>
<td>Memory: 64GB</td>
<td></td>
<td>Kernel: 4.12.4</td>
<td>RocksDB 5.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arch: x86_64</td>
<td>TokuDB 4.6.119</td>
</tr>
</tbody>
</table>

### Dataset:
- 500GB - 1TB (500 million - 1 billion records)
- Dataset size higher (> 3:1 DRAM size)
- Compression Off

- Key_size: 16 Bytes
- Value_size: 1000 Bytes
- Cache size: 32GB

### db_bench:
- Source: https://github.com/wiredtiger/leveldb.git
- Test Duration: 30 minutes

### Workloads:
- ReadWrite (16 Readers, 1 Writer)

### Linux Kernel 4.12.0 Tuning parameters
- Drop page cache after every test run
- XFS filesystem, agcount=32, mount with discard
Readwrite: WiredTiger B-Tree

500 million rows

WiredTiger BTree Writes
(30 min run)

WiredTiger BTree Reads
(30 min run)

Read plot represents single Reader.

Ops Completed: 1 thread Write 2.15 million, 16 thread Read 27.36 million
Readwrite: WiredTiger B-Tree

500 million rows

Test: Single Writer, 16 Readers. Read plot represents single Reader latency.
Design Choice: LSM Tree

- Log-Structured Merge-tree
- Two or more Tree-like components
  - In-memory Tree (RocksDB: memtable)
  - One or more Trees on persistent store (RocksDB: SST files)
- Transforms random writes into few sequential writes
  - Write-Ahead Log (WAL) – append-only journal
  - In-memory store – inexpensive writes to memory as the first level write. Flushed sequentially to first level in persistent store.
- Compaction – Merge sort like background operation
  - Few sequential writes (better than random IOs in B-Tree case)
  - Trims duplicate data – minimal space amplification
- Example DB engines: RocksDB, WiredTiger, Cassandra, CouchBase, LevelDB
LSM Trees – Operation

- N-level Merge Tree
- Transform random writes into sequential writes using WAL and In-memory tables
- Optimized for insertions by buffering
- Key value items in the store are sorted to support faster lookups
Design Choice: LSM Tree

- Writes – Append-only constructs
  - No read-modify-write, no double write
  - Reduce fragmentation but sort/merges to multiple levels cause Write Amplification
- Reads – Expensive point, order by and range queries
  - Might call for compaction
  - Might need to scan all levels
  - Read Amplification
  - Can be optimized with Bloom Filters (RocksDB)
- Deletes – Defers deletes via tombstones
  - Tombstones scanned during queries
  - Tombstones don’t disappear until compaction
LSM Trees: Trade-offs

- **Read / Write / Space Amplification Summary**
  - Read amp: 1 up to number of levels
  - Write amp: 1 + 1 + fan-out

- **Most useful in applications**
  - Tiered storage with varying price / performance points
    - RAM, Persistent Memory, NVMe SSDs etc
  - Large dataset where inserts / deletes are more common than reads / searches

- **Better suited for Write-intensive workloads**
  - Better compression: page/block alignment overhead small compared to size of persistent trees (SST files in RocksDB)
  - Leveled LSMs have lower write and space amplification compared to B-Tress
LSM Tree Example: RocksDB

1 billion rows

RocksDB LSM Write Throughput (30 min run)

RocksDB LSM Read Throughput (30 min run)

Read plot represents single Reader.

Ops Completed: 1 thread Write 5.2 million, 16 thread Read 6.3 million
LSM Tree Example: RocksDB

1 billion rows

RocksDB LSM Write Latency (30 min run)

RocksDB LSM Read Latency (30 min run)

Test: Single Writer, 16 Readers. Read plot represents single Reader Latency.
Design Choice: Fractal Trees

- Merges features from B-trees with LSM trees
- A Fractal Tree index has buffers at each node, which allow insertions, deletions and other changes to be stored in intermediate locations
- Fast writes slower reads and updates
- Better than LSM for range reads on cold cache, but the same on warm cache
- Commercialized in DBs by Percona (Tokutek)
Maintains a B tree in which each internal node contains a buffer, shaded RED.

To insert a data record, simply insert it into the buffer at the root of the tree vs. traversing entire tree (B-Tree).

When root buffer full, move inserted record down one level until they reach leaves and get stored in leaf node like B-Tree.

Several leaf nodes are grouped to create sequential IO (~1-4MB).

Insert(31, ...) moves 41, 43 into the next level.
Fractal Trees: Trade-offs

- Read / Write / Space Amplification Summary
  - Write Amplification similar to leveled LSM (RocksDB)
  - Better than B-Tree and LSM (in theory)
- Most useful in applications
  - Tiered storage with varying price / performance points
    - RAM, Persistent Memory, NVMe SSDs etc
  - Large dataset where inserts / deletes are more common than reads / searches
- Better suited for Write-intensive workloads
  - Lower write and space amplification compared to B-Tress
Fractal Tree Example: TokuDB
1 billion rows

Test: Single Writer, 16 Readers. Read plot represents single Reader.
Fractal Tree Example: TokuDB

1 billion rows

Test: Single Writer, 16 Readers. Read plot represents single Reader.
Outline

- KV Stores – What and Why
- Design Choices and Trade-offs
- Performance Comparisons
- Key Takeaways
# Performance Test Configuration

<table>
<thead>
<tr>
<th>System Config</th>
<th>Non-Volatile Storage</th>
<th>OS Config</th>
<th>Key-Value Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Intel(R) Xeon(R) CPU E5-2618L v4, 2.20GHz, HT disabled, 20 cores</td>
<td>Intel ® P4500 SSD (4TB)</td>
<td>Distro: Ubuntu 16.04.1</td>
<td>WiredTiger 3.0.0</td>
</tr>
<tr>
<td>Memory: 64GB</td>
<td></td>
<td>Kernel: 4.12.4</td>
<td>RocksDB 5.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arch: x86_64</td>
<td>TokuDB 4.6.119</td>
</tr>
</tbody>
</table>

**Dataset:**
- Dataset size higher (> 3:1 DRAM size)
- Compression Off

**Key_size:** 16 Bytes
**Value_size:** 1000 Bytes
**Cache size:** 32GB

**db_bench:**
- Source: https://github.com/wiredtiger/leveldb.git
- Test Duration: 30 minutes

**Workloads:**
- ReadWrite (16 Readers, 1 Writer)
- Overwrite (4 Writers)

**Linux Kernel 4.12.0 Tuning parameters**
- Drop page cache after every test run
- XFS filesystem, agcount=32, mount with discard
## KV Workload#0: Fill (Insert)
### 1 Writer, Space and Write Amplification

<table>
<thead>
<tr>
<th></th>
<th>App Writes (GB)</th>
<th>Disk Usage (GB)</th>
<th>Space Amplification</th>
<th>Data Bytes Written (per iostat, GB)</th>
<th>Write Amplification (per iostat)</th>
<th>SSD Writes* includes NAND GC (GB)</th>
<th>Total Write Amplification (including NAND GC)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WiredTiger</strong></td>
<td>~190</td>
<td>223</td>
<td>1.17</td>
<td>223</td>
<td>1.07</td>
<td>239</td>
<td>1.3</td>
</tr>
<tr>
<td>(B-Tree)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RocksDB</strong></td>
<td>~190</td>
<td>197</td>
<td>1.04</td>
<td>395</td>
<td>1.11</td>
<td>438</td>
<td>2.3</td>
</tr>
<tr>
<td>(Leveled LSM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TokuDB</strong></td>
<td>~190</td>
<td>196</td>
<td>1.03</td>
<td>605</td>
<td>1.14</td>
<td>688</td>
<td>3.6</td>
</tr>
<tr>
<td>(Fractal-Tree)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Space Amplification** is the amount of disk space consumed relative to the total size of KV write operation.

**Write Amplification (per iostat)** is the amount of data written relative to the total size of KV operation.

**Write Amplification (including NAND GC)** is the work done by the storage media relative to the total size of KV operation.

* - SSD Device Writes obtained with nvme-cli from SMART stats for Intel® P4500
KV Workload#1: Readwhilewriting

1 Writer, 16 Readers

**ReadWrite: KV Store Write Throughput**

- TokuDB: 900 OPs/sec
- RocksDB: 4580 OPs/sec
- WiredTiger (B-Tree): 652 OPs/sec

**ReadWrite: KV Store Read Throughput**

- TokuDB: 1966 OPs/sec
- RocksDB: 2444 OPs/sec
- WiredTiger (B-Tree): 6894 OPs/sec

**99.99pct Write Latency (s)**

- WiredTiger (B-Tree): 10
- RocksDB: 30
- TokuDB: 35

2017 Storage Developer Conference
KV Workload#2: Overwrite

4 Writers

OverWrite: KV Store Write Throughput

<table>
<thead>
<tr>
<th>KV Store</th>
<th>Operations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiredTiger (B-Tree)</td>
<td>4204</td>
</tr>
<tr>
<td>RocksDB</td>
<td>20793</td>
</tr>
<tr>
<td>TokuDB</td>
<td>21602</td>
</tr>
</tbody>
</table>

99.99pct Write Latency (s)

- TokuDB: 120
- RocksDB: 20
- WiredTiger (B-Tree): 5000

2017 Storage Developer Conference
Outline

- KV Stores – What and Why
- Design Choices and Trade-offs
- Performance on Non-Volatile Media
- Key Takeaways
Key Takeaways

- Key-Value stores important for unstructured data
- Choice of an SSD-backed Key Value store is workload-dependent
  - Performance vs Space / Write Amplification trade-off key decision factor
- Traditional B-Tree implementations
  - Great for read-intensive workloads
  - Better write amplification compared to alternatives
  - Poor space amplification
- LSM and Fractal-Trees
  - Well-suited for write-intensive workloads
  - Better space amplification compared to B-Tree
Thank you!

Comments / Questions?

Vishal Verma (vishal.verma@intel.com)