ECE 7650 Scalable and Secure Internet Services and Architecture
---- A Systems Perspective

Part II: Software Infrastructure in Data Centers:
Key-Value Data Management Systems
Key-Value Store

Clients

PUT(key, value)
value = GET(key)
DELETE(key)

- Dynamo at Amazon
- BigTable (LevelDB) at Google
- Redis at GitHub, Digg, and Blizzard Interactive
- Memcached at Facebook, Zynga and Twitter
- Voldemort at Linkedin
NoSQL DB and KV Store

- A NoSQL or Not Only SQL database stores and organizes data differently from the tabular relations used in relational databases.

- Why NoSQL?
  - Simplicity of design, horizontal scaling, and finer control over availability.

- It can be classified as column, document, key-value, and graph store based on their data models.

- The key-value model is one of the simplest non-trivial data models, and richer data models are often implemented on top of it.
  - Applications store their data in a schema-less way.

- Array, linked list, binary search trees, B+ tree, or hash table …?
The First Example: Memcached
Memcached

- Memcached is a high-performance, distributed in-memory object caching system, generic in nature.
- It is a key-based cache daemon that stores data (or objects) wherever dedicated or spare RAM is available for very quick access.
- It is a distributed hash table. It doesn’t provide redundancy, failover or authentication. If needed the client has to handle that.
Memcached

- To reduce the load on the database by caching data BEFORE it hits the database
- Can be used for more than just holding database results (objects) and improve the entire application’s response time
- Feel the need for speed
  - Memcache is in RAM - much faster then hitting the disk or the database
A Typical Use Case of Memcached
A Typical Use Case of Memcached

Web Servers

Database

Memcached Servers

Web Servers request data from the Database. The Memcached Servers store the data in key-value pairs. When the request is made, the Memcached Servers retrieve the data from their memory, which is much faster than accessing the Database.

SQL Query:

```sql
SELECT `name` FROM `users` WHERE `id` = 1;
```

Key:

```
MD5("SELECT `name` FROM `users` WHERE `id` = 1;")
```
A Typical Use Case of Memcached

**Web Servers**

**Database**

**Memcached Servers**

Key = MD5("SELECT `name` FROM `users` WHERE `id` = 1;")

GET(key)

Hash (Key) ➔ Server ID

Miss
A Typical Use Case of Memcached

Web Servers

SELECT `name` FROM `users` WHERE `id` = 1;

Database

Memcached Servers
A Typical Use Case of Memcached

Web Servers

Database

Memcached Servers
A Typical Use Case of Memcached

Key = MD5("SELECT `name` FROM `users` WHERE `id` = 1;")
Value = Query Result

SET(key, Value)
A Typical Use Case of Memcached

Web Servers

Database

Memcached Servers
Memcached

Distributed memory caching system as a cluster

Key

Hash Function

Memcached Servers
Slab-based memory allocation: memory is divided into 1 MB slabs consisting of fixed-length chunks. E.g., slab class 1 is 72 bytes and each slab of this class has 14563 chunks; slab class 43 is 1 MB and each slab has only one chunk. Least-Recently-Used (LRU) algorithm to select the items for replacement in each slab-class queue. (why slab?)

Lock has to be used for integrity of the structure, which is very expensive (why?)
Memcached Workload Pattern at Facebook

Figure 1. Request rates at different (a) days (USR) and (b) times of day (ETC, in Coordinated Universal Time [UTC]). Each data point counts the total number of requests in the preceding second.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In Proceedings of the SIGMETRICS'12,
Memcached Workload Pattern at Facebook

Figure 2. Key and value size distributions for all traces: (a) key size cumulative distribution function (CDF) by appearance, up to Memcached’s limit of 250 bytes (not shown); (b) value size CDF by appearance; and (c) value size CDF by total amount of data used in the cache. For example, values under 320 bytes or so in pool SYS use virtually no space in the cache; 320-byte values weigh around 8 percent of the data; and values close to 500 bytes take up nearly 80 percent of the entire cache’s allocation.
Memcached Workload Pattern at Facebook

Figure 3. Distribution of request types per pool, over exactly seven days: (a) USR, n = 60.7 billion requests; (b) APP, n = 39.5 billion requests; (c) ETC, 30 billion requests; (d) VAR, 44.6 billion requests; and (e) SYS, 4.4 billion requests. UPDATE commands aggregate all non-DELETE writing operations.

Why do update requests represent more expensive operations in memcached?
A Well-known One: Google’s BigTable and LevelDB

- A multi-layered LSM-tree structure
- Progressively sort data for small memory demand
- Small number of levels for effective Bloom Filter use.

Chang, et al., **Bigtable: A Distributed Storage System for Structured Data** in OSDI’06.
Google’s BigTable and LevelDB to Scale Data Store

Scale Problem

- Lots of data
- Millions of machines
- Different project/applications
- Hundreds of millions of users

Storage for (semi-)structured data

No commercial system big enough

- Couldn’t afford if there was one

Low-level storage optimization helps performance significantly

  Much harder to do when running on top of a database layer
Bigtable

Fault-tolerant, persistent

Scalable

- Thousands of servers
- Terabytes of in-memory data
- Petabyte of disk-based data
- Millions of reads/writes per second, efficient scans

Self-managing

- Servers can be added/removed dynamically
- Servers adjust to load imbalance
Data model: a big map

- `<Row, Column, Timestamp>` triple for key
  - Each value is an uninterpreted array of bytes
- Arbitrary "columns" on a row-by-row basis
  - Column family: qualifier. A small number of families and large number of columns
- Lookup, insert, delete API

Each read or write of data under a single row key is atomic
SSTable

Immutable, sorted file of key-value pairs

Chunks of data plus an index
  • Index is of block ranges, not values
  • Index loaded into memory when SSTable is opened
  • Lookup is a single disk seek

Alternatively, client can load SSTable into memory

![Diagram of SSTable with 64K block and index]

64K block 64K block 64K block

SSTable

Index
**Tablet**

Contains some range of rows of the table  
Unit of distribution & load balance  
Built out of multiple SSTables
Multiple tablets make up the table

Tablet
aardvark

Tablet
apple

Tablet
apple_two_E

SSTable
SSTable

SSTable
SSTable

boat
Finding a tablet

- Client library caches tablet locations
Servers

Tablet servers manage tablets, multiple tablets per server. Each tablet is 100-200 MBs

- Each tablet lives at only one server
- Tablet server splits tablets that get too big

Master responsible for load balancing and fault tolerance

- Use Chubby to monitor health of tablet servers, restart failed servers
- GFS replicates data. prefer to start tablet server on same machine that the data is already at
Editing/Reading a table

**Mutations** are committed to a commit log (in GFS)
Then applied to an in-memory version (memtable)
**Reads** applied to merged view of SSTables & memtable
Reads & writes continue during tablet split or merge
Bigtable Tablet

LevelDB is similar to a single Bigtable tablet

Figure 5: Tablet Representation
Compactions

Minor compaction – convert a full memtable into an SSTable, and start a new memtable

- Reduce memory usage
- Reduce log traffic on restart

Major compaction

- Combining a number of SSTables into possibly smaller number of SSTables.
- No deletion records, only live data
Management of SSTable -- LSM-tree

Log-Structured Merge-Tree (LSM-tree)

- Optimized for fast random updates, inserts and deletes with moderate read performance.

- Convert the random writes to sequential writes
  - Accumulate recent updates in memory
  - Flush the changes to disks sequentially in batches
  - Merge on-disk components periodically

- At the expense of read performance
Google’s LevelDB: Progressively Increasing Level Size

- **MemTable**
- **Write**
- **Immutable MemTable**
- **Memory**
- **Disk**
- **Dump**

- **Level 0**
  - SSTable
  - Compaction

- **Level 1**
  - 10MB
  - SSTable
  - …
  - SSTable

- **Level 2**
  - 100 MB
  - …
  - SSTable

- **Log**
- **Manifest**
- **Current**

- **K1 V1**
- **K2 V2**
- **K3 V3**
LevelDB Write Flow
LevelDB Read Flow

Bloom Filter
LevelDB Compact (L0/L1)
LevelDB Compact (L0/L1 Move)
Background on Bloom Filter

- Data structure proposed by Burton Bloom
- Randomized data structure
  - Strings are stored using multiple hash functions
  - It can be queried to check the presence of a string
- Membership queries result in rare false positives but never false negatives
- Originally used for UNIX spell check
- Modern applications include:
  - Content Networks
  - Summary Caches
  - route trace-back
  - Network measurements
  - Intrusion Detection
Programming a Bloom Filter
Querying a Bloom Filter

![Diagram of a Bloom Filter](image)
Querying a Bloom Filter (False Positive)
Optimal Parameters of a Bloom Filter

Bloom filter computes k hash functions on input

- \( n \) : number of strings to be stored
- \( k \) : number of hash functions
- \( m \) : the size of the bit-array (memory)

- The false positive probability
  \[
  f = \left( 1 - \left( 1 - \frac{1}{m} \right)^{kn} \right)^k \approx (1 - e^{-kn/m})^k.
  \]

- The optimal value of hash functions, \( k \), is
  \[
  k = \ln 2 \times m/n = 0.693 \times m/n
  \]

Key Point : false positive rate decreases exponentially with linear increase in number of bits per string (item)
A Use Scenario of KV store:
Data-intensive Networked Systems

- Object store (~4 TB)
- HasTable (~32 GB)
- Chunk pointer
- Key (20 B)
- Chunks (4 KB)
- Look up
- Data center
- Branch office
- WAN optimizers
- WAN

- High speed (~10 K/sec) inserts and evictions
- High speed (~10K/sec) lookups for 500 Mbps link
- Large hash tables (32 GB)
### Candidate options

<table>
<thead>
<tr>
<th></th>
<th>Random reads/sec</th>
<th>Random writes/sec</th>
<th>Cost (128 GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>250</td>
<td>250</td>
<td>$30^+</td>
</tr>
<tr>
<td>DRAM</td>
<td>300K</td>
<td>300K</td>
<td>$120K^+</td>
</tr>
<tr>
<td>Flash-SSD</td>
<td>10K*</td>
<td>5K*</td>
<td>$225^+</td>
</tr>
</tbody>
</table>

+Price statistics from 2008-09

Too slow

Too expensive

2.5 ops/sec/$

Slow writes

* Derived from latencies on Intel M-18 SSD in experiments

---

How to deal with slow writes of Flash SSD?
Flash/SSD primer

Random writes are expensive

Avoid random page writes

Reads and writes happen at the granularity of a flash page

I/O smaller than page should be avoided, if possible
Flash Solid State Drive (SSD)

File System

Read pages

Write pages

Block Interface

Controller (FTL)

RAM

Flash Memory
Basics of NAND Flash Memory

Three operations: read, write, erase

Reads and writes are done at the granularity of a page (2KB or 4KB)

Erases are done at the granularity of a block

- Block: A collection of physically contiguous pages (64 or 128)
- Block erase is the slowest operation requiring about 2ms

 Writes can only be done on erased pages
Out-of-Place Updates

Over-writes on the same location (page) are expensive

Updates are written to a free page

OOB area

- Keeps valid/free/invalid status
- Stores LPN, used to reconstruct mapping table upon power failure

Flash Mapping Table

<table>
<thead>
<tr>
<th>LPN</th>
<th>PPN (PBN, Offset)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(0, 3)</td>
</tr>
<tr>
<td>B</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>C</td>
<td>(0, 2)</td>
</tr>
</tbody>
</table>
Flash Translation Layer (FTL)

Flash Translation Layer
- Emulates a normal block device interface
- Hides the presence of erase operation/erase-before-write
- Address translation, garbage collection, and wear-leveling

Address Translation
- Mapping table present in small RAM within the flash device
A Naïve Design: Hash Table on Flash/SSD

Cannot move entire hash table to the SSD:
*Keys are likely to hash to random locations*

SSDs: FTL handles random writes to some extent;
But garbage collection overhead is high

~200 lookups/sec and ~200 inserts/sec with WAN optimizer workload, << 10 K/s and 5 K/s
Can’t assume locality in requests – DRAM as cache won’t work
Three Metrics to Minimize

Memory overhead = Index size per entry
• Ideally 0 (no memory overhead)

Read amplification = Flash reads per query
• Limits query throughput
• Ideally 1 (no wasted flash reads)

Write amplification = Flash writes per entry
• Limits insert throughput
• Also reduces flash life expectancy
  • Must be small enough for flash to last a few years
Design I: FAWN for Fast Read and Fast Write

FAWN (Fast Array of Wimpy Nodes)
- A Key-Value Storage System
  - I/O intensive, not computation-intensive
  - Massive, concurrent, random, small-sized data access
- A new low-power cluster architecture
  - Nodes equipped with embedded CPUs + FLASH
  - A tenth of the power compared to conventional architecture

David G. Andersen, et al, FAWN: a Fast Array of Wimpy Nodes, on SOPS’09
FAWN System

- Hash Index to map 160-bit keys to a value stored in the data log;
- It stores only a fragment of the actual key in memory to find a location in the log => reduce memory use.
- How to store, lookup, update, and delete?
- How to do garbage collection?
- How to reconstruct after a crash, and how to speed up the reconstruction?
- Why is a Delete entry necessary? (hint: fault tolerance)
FAWN DataStore

<table>
<thead>
<tr>
<th>Constrained DRAM</th>
<th>Avoid Random Write in Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only maintain Hash Table in DRAM</td>
<td>Append-only log-structured filesystem</td>
</tr>
</tbody>
</table>

Chained hash entries in each bucket
DRAM Must be Used Efficiently

DRAM used for index (locate) items on flash
1 TB of data to store on flash
4 bytes of DRAM for key-value pair (previous state-of-the-art)

- 32 B: Data deduplication => 125 GB!
- 168 B: Tweet => 24 GB
- 1 KB: Small image => 4 GB
Design II: SkimpyStash for Small Memory Demand

- Memory use is not proportional to the KV item count.
- Place the hash table buckets, including the links, to the flash.
- Only the first pointers to buckets (Hash table directory), are in memory.

Debnath et al., SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage in SIGMOD’11
Use in-RAM write buffer to enable batched writes (timeout threshold to bound response time and concurrent write and flush)

Basic operations:
- lookup: HT directory in memory $\rightarrow$ bucket on the flash
- insert: buffer in memory $\rightarrow$ batched write to the flash as a log and linked into HT directory.
- delete: write a NULL entry
SkimpyStash Architecture
Garbage collection in the log:
- Start from the log tail (other than the currently written end).
- Do page by page,
- Cannot update predecessor’s pointer in the bucket.
- Compact and relocate whole bucket → leave orphans for garbage collection.

The cost of write/delete/lookup
In the worst case how many flash reads are needed for one lookup?

The consequence of unbalanced buckets
Exceptionally long buckets → unacceptably long lookup time!

Solution: two-choice-based hashing: each key would be hashed to two candidate HT directory buckets, using two hash functions h1 and h2, and inserted into the one that has currently fewer elements

How to know in which bucket to search for a lookup?
Use of Bloom filter for each bucket. The filter is dimensioned one byte per key and assume average number of items in each bucket.
The consequence of spreading a chain of entries in a bucket across pages.

Use compaction to ameliorate the issue.
The Weaknesses of SkimpyStach

- Long/Unpredictable/unbounded lookup time.
Design III: BufferHash using Equally-sized Levels

- Move entire hash tables to the disk/flash
- The store consists of multiple levels and each is organized as a hash table.

Anand et al., Cheap and Large CAMs for High Performance Data-Intensive Networked Systems in NSDI’10
The approach: Buffering insertions

Control the impact of random writes
Maintain small hash table (buffer) in memory
As in-memory buffer gets full, write it to flash
  • We call in-flash buffer, incarnation of buffer
Two-level memory hierarchy

Net hash table is: buffer + all incarnations
Lookups are impacted due to buffers

Multiple in-flash lookups. Can we limit to only one?

⇒ Use **Bloom Filters**
Bloom filters for optimizing lookups

In-memory lookups
False positive!

Configure carefully!

2 GB Bloom filters for 32 GB Flash for false positive rate < 0.01!
Update: naïve approach

Update key

Expensive random writes

Discard this naïve approach
Lazy updates

Update key

Insert key

Key, new value

Key, old value

Lookups check latest incarnations first

DRAM

Buffer

Bloom filters

Flash

Incarnation table

Update key →

Insert key →

4 3 2 1
**BufferHash is just a KV Cache: Eviction for streaming apps**

Eviction policies may depend on application
- LRU, FIFO, Priority based eviction, etc.

Two BufferHash primitives
- Full Discard: evict all items
  - Naturally implements FIFO
- Partial Discard: retain few items
  - Priority based eviction by retaining high priority items

BufferHash best suited for FIFO
- Incarnations arranged by age
- Other useful policies at some additional cost

Question: how to retain items of priority?
Issues with using one buffer

Single buffer in DRAM
  • For all operations and eviction policies

High worst case insert latency
  • Few seconds for 1 GB buffer
  • New lookups stall
Sharding: Partitioning buffers

Partition buffers
- Based on first few bits of key space
- Size > page
  - Avoid i/o less than page
- Size >= block
  - Avoid random page writes

Reduces worst case latency

Eviction policies apply per buffer

Incarnation table

DRAM

Flash

0

1
BufferHash: Putting it all together

Multiple buffers in memory
Multiple incarnations per buffer in flash
One in-memory Bloom filter per incarnation

Net hash table = all buffers + all incarnations
Weaknesses of BufferHash

Excessively large number of (incarnations) levels makes BF less effective.

Searching in individual incarnations is not efficient.

<table>
<thead>
<tr>
<th>bits/key</th>
<th>50 Levels</th>
<th>100 Levels</th>
<th>150 Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40.95%</td>
<td>81.90%</td>
<td>122.85%</td>
</tr>
<tr>
<td>12</td>
<td>15.70%</td>
<td>31.40%</td>
<td>47.10%</td>
</tr>
<tr>
<td>14</td>
<td>6.00%</td>
<td>12.00%</td>
<td>18.00%</td>
</tr>
<tr>
<td>16</td>
<td>2.30%</td>
<td>4.59%</td>
<td>6.89%</td>
</tr>
<tr>
<td>18</td>
<td>0.88%</td>
<td>1.76%</td>
<td>2.64%</td>
</tr>
</tbody>
</table>

Table 1: Bloom-filter false-positive rate.
Design IV: SILT with Levels of Dramatically-different Sizes

Hyeontaek Lim et al, **SILT: A Memory-Efficient, High-Performance Key-Value Store**, in SOSP’11.
Seesaw Game?

How can we improve?

Memory efficiency

High performance

SkimpyStash

FAWN-DS
BufferHash
Solution Preview: (1) Three Stores with (2) New Index Data Structures

Queries look up stores in sequence (from new to old)

Inserts only go to Log

Data are moved in background

SILT Sorted Index (Memory efficient)

SILT Filter

SILT Log Index (Write friendly)
LogStore: No Control over Data Layout

Naive Hashtable (48+ B/entry)
SILT Log Index (6.5+ B/entry)

Still need pointers: size ≥ log N bits/entry

Memory overhead
6.5+ bytes/entry

Write amplification
1
How to find the alternative slot for displacement by storing hash index in the tag?
HashStore: Remove in-memory HT (or the index)

HashStore saves memory over LogStore by eliminating the index and reordering the on-flash (key,value) pairs from insertion order to hash order.

Figure 4: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as the in-memory filter.
SortedStore: Space-Optimized Layout

To merge HashStore entries into the SortedStore, SILT must generate a new SortedStore.
SILT’s Design (Recap)

- **<SortedStore>**
  - SILT Sorted Index
  - On-flash sorted array

- **<HashStore>**
  - SILT Filter
  - On-flash hash tables

- **<LogStore>**
  - SILT Log Index
  - On-flash log

**Memory overhead**
- 0.7 bytes/entry

**Read amplification**
- 1.01

**Write amplification**
- 5.4
Any Issue with SILT?

- SILT provides both memory-efficient and high-performance key-value store
  - Multi-store approach
  - Entropy-coded tries
  - Partial-key cuckoo hashing

- The weakness: Write amplification is way too high!