SILT: A MEMORY-EFFICIENT, HIGH-PERFORMANCE KEY-VALUE STORE

PRESENTED BY
PRIYA SRIDHAR
AGENDA

- INTRODUCTION
- Why SILT?
- MOTIVATION
- SILT KV STORAGE SYSTEM
  - LOW READ AMPLIFICATION
  - CONTROLLABLE WRITE AMPLIFICATION
- ARCHITECTURE OF SILT
- BASIC STORE DESIGN
  - LOG STORE
  - HASH STORE
  - SORTED STORE
- IN MEMORY-CUCKOO HASH TABLE
INTRODUCTION

- SILT (Small Index Large Table) is a memory-efficient, high performance key-value store system based on flash storage that scales to serve billions of key-value items on a single node.

- This paper presents a new flash-based key-value storage system, called SILT (Small Index Large Table), that significantly reduces per-key memory consumption with predictable system performance and lifetime.

- This paper makes three main contributions:
  - The design of three basic key-value stores each with a different emphasis on memory-efficiency and write-friendliness
  - Synthesis of the Basic Store Designs.
  - An analytical model for tuning system parameters carefully to meet the needs of different workloads.
WHY SILT?

- Key-value storage systems have become a critical building block for today's large-scale, high-performance data-intensive applications.
- High-performance key-value stores have therefore received substantial attention in a variety of domains, both commercial and academic.
- SILT requires approximately 0.7 bytes of DRAM per key-value entry and uses on average only 1.01 flash reads to handle lookups.
Q1) “Figure 1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.” Explain the positions of FAWN-DS, SkimpyStash, BufferHash, and SILT on the graph.
MOTIVATION

- **FAWN-DS** - Retrieves the hash entry containing the offset, indexes data log and returns the data blob.

- **SKIMPYSTASH** - First looks up the RAM write buffer, if not found there it will look up in the hash table directory in RAM and will search the chained key value pair records on flash in the respective bucket.

- **BUFFERHASH** – Uses buffer, incarnations(in Flash) and bloom filters.

- **SILT** – It has three stores, logstore, hashstore, sortedstore. Since the data is indexed lookup is very fast and since SILT uses only 0.7 bytes/entry it has a low memory overhead.
Q2) Two design goals of SILT are low read amplification and low write amplification. Use any KV store we have studied as an example to show how these amplifications are produced.
SILT-KV STORAGE SYSTEM

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
Q3) Describe SILT’s structure using Figure 2 (Architecture of SILT). Compared with LevelDB, SILT has only three levels. What’s concern with a multi-level KV store when it has too few levels?
ARCHITECTURE

- **Conventional Single-Store Approach**: A common approach to building high-performance key-value stores on flash uses three components:
  1. an in-memory filter to efficiently test whether a given key is stored in this store before accessing flash
  2. an in-memory index to locate the data on flash for a given key
  3. an on-flash data layout to store all key-value pairs persistently.

- **Multi-Store Approach**: Multiple stores (usually DRAM and FLASH) are used for write amplification or inexpensive reads.

- **SILT**: uses a series of basic key-value stores, each optimized for a different purpose.
ARCHITECTURE OF SILT
BASIC STORE DESIGN

- **LogStore**
  - Write friendly key-value store (similar to conventional storage)
  - In-memory hash table to map contents in flash
  - Modified Cuckoo hashing is used

- **HashStore**
  - Immutable hash store
  - No need to in-memory index to locate entries

- **SortedStore**
  - Key-value storage on flash
  - Extremely compact index representation (0.4 bytes/key)
SILT’s Design

<SortedStore>
SILT Sorted Index
On-flash sorted array

<HashStore>
SILT Filter
On-flash hashtables

<LogStore>
SILT Log Index
On-flash log

Merge
Conversion

Memory overhead 0.7 bytes/entry
Read amplification 1.01
Write amplification 5.4
Q4) Use Figure 3 (*Design of LogStore: an in-memory cuckoo hash table (index and filter)*) to describe how a PUT request and a GET request is served in a LogStore. In particular, explain how the tag is used in a LogStore.
DESIGN OF LOGSTORE

- The LogStore writes PUTs and DELETEs sequentially to flash to achieve high write throughput.

- Partial-Key Cuckoo Hashing: Each key is “tagged” to reduce the memory. The tag is used for lookup and if it matches, the full key is retrieved from the log.

- Associativity: Each bucket of the table is of capacity four (i.e., it contains up to 4 entries).

- Hash Functions: Each hash table entry is 6 bytes, consisting of a 15-bit tag, a single valid bit, and a 4-byte offset pointer.
QUESTIONS