Cheap and Large CAMs for High Performance Data-Intensive Networked Systems- The Bufferhash KV Store

Presented by
Akhila Nookala
M.S EE Wayne State University
ECE7650 Scalable and Secure Internet Services and Architecture, Winter 2016
Introduction

• The goal of the paper is to build cheap and large CAMs, or CLAMs, using a combination of DRAM and flash memory.

• These are targeted at emerging data-intensive networked systems that require massive hash tables running into a hundred GB or more, with items being inserted, updated and looked up at a rapid rate.
Problem Statement

• For such systems, using DRAM to maintain hash tables is quite expensive, while on-disk approaches are too slow.

• In contrast, CLAMs cost nearly the same as using existing on-disk approaches but offer orders of magnitude better performance.

• The design leverages an efficient flash-oriented data-structure called Buffer Hash that significantly lowers the amortized cost of random hash insertions and updates on flash. Buffer Hash also supports flexible CLAM eviction policies.
Buffer Hash KV store

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

• BufferHash consists of multiple super tables.
• Each super table has three main components: a buffer, an incarnation, table, and a set of Bloom filters.
• Components in the higher level are maintained in DRAM, while those in the lower level are maintained in flash.
The approach: Buffering insertions

- Maintain small hash table (buffer) in memory
- When the memory buffer gets full, write it to flash
- We call in-flash buffer, incarnation of buffer
- Incarnation: In-flash hash table Buffer: In-memory hash table DRAM Flash SSD
- Multiple insertions in Lazy batched manner
Buffer: In-memory hash table

DRAM

Flash SSD

Incarnation: In-flash hash table
A Super Table

Figure 1: *A Super Table*
Buffer

- Buffer. This is an in-memory hash table where all newly inserted hash values are stored.

- A buffer can hold a fixed maximum number of items, determined by its size and the desired upper bound of hash collisions.

- When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash, after which the buffer is re-initialized for inserting new keys. The buffers flushed to flash are called incarnations.
Incarnation table.

- This is an in-flash table that contains old and flushed incarnations of the in-memory buffer.
- The table contains $k$ incarnations, where $k$ denotes the ratio of the size of the incarnation table and the buffer.
- The table is organized as a circular list, where a new incarnation is sequentially written at the list-head.
- To make space for a new incarnation, the oldest incarnation, at the tail of the circular list, is evicted from the table.
- Depending on application’s eviction policy, some times in an evicted incarnation may need to be retained and are re-inserted into the buffer.
Bloom filters.

• Since the incarnation table contains a sequence of incarnations, the value for a given hash key may reside in any of the incarnations depending on its insertion time. A naive lookup algorithm for an item would examine all incarnations, which would require reading all incarnations from flash.

• The Bloom filter for an incarnation is a compact signature built on the hash keys in that incarnation. To search for a particular hash key, we first test the Bloom filters for all incarnations.
Bloom filters

- If any Bloom filter matches, then the corresponding incarnation is retrieved from flash and looked up for the desired key. Bloom filter-based lookups may result in false positive thus, a match could be indicated even though there is none, resulting in unnecessary flash I/O.

- As the filter size increases, the false positive rate drops, resulting in lower I/O overhead.
Super Table Operations

• Insert. To insert a (key, value) pair, the value is inserted in the hash table in the buffer. If the buffer does not have space to accommodate the key, the buffer is flushed and written as a new incarnation in the incarnation table.

• The incarnation table may need to evict an old incarnation to make space.
Lookup.

- A key is first looked up in the buffer. If found, the corresponding value is returned. Otherwise, in-flash incarnations are examined in the order of their age until the key is found.

- To examine an incarnation, first its Bloom filter is checked to see if the incarnation might include the key. If the Bloom filter matches, the incarnation is read from flash, and checked if it really contains the key. Note that since each incarnation is in fact a hash table, to lookup a key in an incarnation, only the relevant part of the incarnation (e.g., a flash page) can be read directly.
Bloom filters for optimizing lookups

Lookup key →

In-memory lookups
False positive!

Configure carefully!

DRAM
Buffer
Bloom filters
Flash

Incarnation table
Update

• As mentioned earlier, flash does not support small updates/deletions efficiently hence, we support them in a lazy manner.

• Suppose a super table contains an item \((k, v)\), and later, the item needs to be updated with the item \((k, v')\).

• In a traditional hash table, the item \((k, v)\) is immediately replaced with \((k, v')\).

• If \((k, v)\) is still in the buffer when \((k, v')\) is inserted, we do the same. However, if \((k, v)\) has already been written to flash, replacing \((k, v)\) will be expensive.
Update

• Hence, we simply insert \((k, v')\) without doing anything to \((k, v)\).
• Lazy update wastes space on flash, as outdated items are left on flash; the space is reclaimed during incarnation eviction.
Delete

• For deleting a key k, a super table does not delete the corresponding item unless it is still in the buffer; rather the deleted key is kept in a separate list (or, a small in-memory hash table), which is Consulted before lookup—if the key is in the delete list, it is assumed to be deleted even though it is present in some incarnation.
Weakness of Buffer Hash

• Excessively large number of (incarnations) levels makes BF less effective.
• Searching in individual incarnations is not efficient.
Question and Answers

(1) “A key idea behind BufferHash is that instead of performing individual random insertions directly on flash, DRAM can be used to buffer multiple insertions and writes to flash can happen in a batch.” Very briefly explain the difference between the ways of FAWN and BufferHash in which they locate a KV pair written on the flash?

- Buffer hash uses bloom filters to locate KV pair on Flash, FAWN-DS maintains an in-DRAM hash table (Hash Index) that maps keys to an offset in the append-only Data Log on flash.

- Buffer hash has an index containing the location of the KV pair but the FAWN looks up in the Flash memory in a sequential manner making the memory overhead high.
(2) “BufferHash consists of multiple super tables. Each super table has three main components: a buffer, an incarnation table, and a set of Bloom filters.” Use Figure 1 to describe Buffer Hash’s data structure.

- **Two Level Heirarchy** – Components in the higher level are maintained in the DRAM and those in lower level are in Flash.
- **Buffer** – An In Memory hash table where all newly inserted hash values are stored. When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash.
- **Incarnation Table**: An Inflash table that contain old and flushed incarnations of the in memory buffer. The table is arranged in a circular way that the oldest incarnation is at the tail of the circular list and the new one at the list head.
- **Set of Bloom Filters**: The Bloom filters are indexed to provide for the lookup operations.

![Figure 1: A Super Table](image-url)
Question and Answers

(3) “This is an in-flash table that contains old and flushed incarnations of the in-memory buffer.” Please explain the relationship between the buffer and the incarnation

• Buffer is an In Memory hash table where all newly inserted hash values are stored. When the number of items in the buffer reaches its capacity, the entire buffer is flushed to flash to form Incarnations.

4) “Since the incarnation table contains a sequence of incarnations, the value for a given hash key may reside in any of the incarnations depending on its insertion time.” Please explain why Bloom filters are needed

• A normal lookup algorithm for an item would examine all incarnations, which would require reading all incarnations from flash. To avoid this excessive I/O cost, a super table maintains a set of in-memory Bloom filters one per incarnation.
(5) “A super table supports all standard hash table operations” Describe the steps involved in insert, lookup, update/delete operations

- **Insert**: To insert a (key, value) pair, the value is inserted in the hash table in the buffer. If the buffer does not have space to accommodate the key, the buffer is flushed and written as a new incarnation in the incarnation table.
- **Lookup**: A key is first looked up in the buffer. If found, the corresponding value is returned. Otherwise, in-flash incarnations are examined in the order of their age until the key is found. Bloom filters are used to check for in-flash lookups.
- **Update/Delete**: Flash does not support small updates/deletions efficiently; hence, we support them in a lazy manner. The updates are done when the hash tables are flushed into the Flash memory.
(6) “If the Bloom filter matches, the incarnation is read from flash, and checked if it really contains the key. Note that since each incarnation is in fact a hash table,....”. Could you describe the structure of the hash table? Could we hold all incarnations’ hash tables in the memory? Why?

• No, we cannot hold all incarnations’ hash tables in the memory. Because the incarnations are arranged by age and the oldest incarnation needs to be completely evicted or partly evicted(according to application policy) to provide the memory for the latest incarnation when the memory gets filled.

• Hash Table is an implementation of Hash function and it consists of KV Pairs
References

• Anand et al., Cheap and Large CAMs for High Performance Data-Intensive Networked Systems in NSDI’10
• http://www.ece.eng.wayne.edu/~sjiang/