Question and Answer Report:

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

Subject:
ECE7650, Scalable and Secure Internet services and Architecture

Term:
Winter2016

Submitted by:
Akhila Nookala
fs4352@wayne.edu
M.S in EE, Department of ECE, Wayne State University.

Under the guidance of :
Dr. Song Jiang
Associate Professor, Department of ECE, Wayne State University.
Cheap and Large CAMs for High Performance Data-Intensive Networked Systems-The Bufferhash KV Store

Introduction and Problem Statement

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. 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.
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, which increases the memory overhead.

(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 Hierarchy – 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 In flash 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.
(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. To search for a particular hash key, we first test the Bloom filters for all incarnations; 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. For many incarnations arranged vertically (forming a shape of vertical table), the performance of the bloom filter is poor. This is due to the fact that the false positive rate of the bloom filter per incarnation is one percent i.e it can cheat only
once and false positive can be correct only once for an incarnation and for many levels of incarnations, the false positive rate would vary, (it would be more than one percent), the probability of false positive being correct can be more than 1%, which would result in the poor performance of the bloom filter and hence the Buffer hash KVstore

The below diagram explains how the incarnations are arranged in vertical pattern in flash and the performance of bloom filter being affected due to many levels of incarnations.
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. The incarnation table may need to evict an old incarnation to make space. The below diagram explains the principle of insertion

**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.

The below diagram explains the principle of lookups in bloom filter.

**Bloom filters for optimizing lookups**

![Diagram showing bloom filters and in-memory lookups.]

Update/Delete: Flash does not support small updates/deletions efficiently; hence, we support them in a lazy manner.

Update: From the design proposed in the paper, we understand 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. Hence, we simply insert \((k, v')\) without doing anything to \((k, v)\).

Since the incarnations are examined in order of their age during lookup, if the same key is inserted with multiple updated values, the latest value (in this example, \(v'\)) is returned by a lookup.
The below diagram explains the principle of Update (Specifically Lazy update).

**Lazy updates**

![Diagram showing the principle of Update](image)

**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.

(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?

Hash Table is an implementation of Hash function and it consists of KV pairs. In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.
No, we cannot hold all incarnations’ hash tables in the memory. As per the design, in the flash the incarnations are arranged by the age and the oldest incarnation needs to be completely evicted or partly evicted in order to provide the memory for the latest incarnation formed or created when the buffer in DRAM is full (i.e., when the memory gets filled).

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/