FAWN: A Fast Array of Wimpy Nodes

Presenter: Murali Pavan V
fs0952
Outline

- Motivation
- FAWN
- Why FAWN?
- Design Overview
- Understanding Flash Storage
- FAWN Data Store
- Basic Functions
Motivation

- Growing importance of High-Performance, Data Intensive applications.
  - Key Value storage systems (Amazon, Facebook, LinkedIn)
  - Small objects, random access over large data sets.

- Power costs dominate the total cost of operations.
  - 50% of total cost, Data Centers are being constructed near stations

- Lack of Power Efficient solution for Data Intensive Computing
  - Disk based solution – Slow random access, High Power Consumption
  - DRAM based solution – Expensive, High Power Consumption.
Q1. “The workloads these systems support share several characteristics: they are I/O, not computation, intensive, requiring random access over large datasets, ..., and the size of objects stored is typically small.” Read the above statement, indicate why workloads of these characteristics represent a challenge to the system design?

- Small-object random-access workloads are ill-served by conventional disk-based clusters and memory-based clusters.
- When the object becomes small, the respective metadata size becomes more compared to the size of the data.
- The poor seek performance of the disks make them inefficient in terms of both system performance and performance per watt.
- Clusters which also include the DRAM are expensive and consume a surprising amount of power, because they require a lot of storage space and which become expensive.
FAWN

- Cost effective/ Power Efficient Cluster for Data Intensive workloads
  - Low Power embedded CPUs + Flash Storage (Wimpy Node)

- FAWN-KV (Key-Value) Design and Implementation
  - Log Structured Data-store (FAWN-DS)
Why FAWN?

- CPU I/O gap
  - Modern processors are so efficient that a lot of time is spent idle

- CPU power consumption scales linearly
  - Increased caches to keep the superscalar

- Dynamic Voltage Frequency Switching (DVFS) is inefficient
  - CPU still operates generally at 50% power consumption
Design

Figure 1: FAWN-KV Architecture.
Understanding the Flash Storage

- Fast Random Reads
  - (<< 1ms), up to 175 times faster than that of magnetic disks
- Efficient I/O
  - Less than 1Watt even under heavy load
- Slow Random Writes
  - Small writes on flash are expensive
  - Updating a single byte of data is as expensive as writing an entire block of pages
Q2. “The key design choice in FAWN-KV is the use of a log structured per-node data-store called FAWN-DS that provides high performance reads and writes using flash memory.” “These performance problems motivate log-structured techniques for flash file systems and data structures” What key benefit does a log structured data organization bring to the KV store?

- The key benefit of the log structured data organization may be the high write throughput that can be obtained since all the updates to the data and the metadata are written in a sequential order in the log. The GET Requests can also be served very fast.
FAWN-DS

- It is a log-structured key value store.
6 Byte Memory Entry
- Key Fragment (15bit) + Valid Bit (1bit) + Pointer (4byte)
- Only a fragment of key stored in Hash (Limited DRAM)
Q3. “To provide this property, FAWN-DS maintains an in-DRAM hash table (Hash Index) that maps keys to an offset in the append-only Data Log on flash.” What are potential issues of the design?

- Every KV pair has an in-memory metadata, the cost can be too high.
- Once the in-memory metadata is lost due to power failure, some of the info can be lost since data is in the memory and needs to be recovered.
Q4. "It stores only a fragment of the actual key in memory to find a location in the log;" Is there correction concern in this design?

After storing the fragment of the key to Memory, the key fragment is searched in the Flash and then, the rest of the key which is stored on the flash is compared. If the key matches there would not be any issue, but if the key is not matched then the key fragment is searched again.

Mostly, there would not be an issue of searching twice. It happens only once in $2^{15}$ times. And also the flash is very fast in reading (up to 2000 reads per second).
Basic Functions
Store, Lookup, Delete

- **Store**: Appends an entry to the log, updates the corresponding hash table entry to point to this offset within the data log and sets the valid bit true.

- **Lookup**: Retrieves the hash entry containing the offset, indexes into Data log and returns the data blob.

- **Delete**: Invalidates the hash entry corresponding to the key by clearing valid flag and writing a Delete entry to the end of the data file. The delete entry is necessary for fault-tolerance.
Q5. “Basic functions: Store, Lookup, Delete” Use Figure 2(a) to explain how these basic functions are executed?
Q6. “As an optimization, FAWN-DS periodically checkpoints the index by writing the Hash Index and a pointer to the last log entry to flash.”. Why does this check-pointing help with the recovery efficiency? Why is a Delete entry needed in the log for a correct recovery

- If there is any failure or any kind of data loss, all of the Hash Index must be recovered totally from the Data log, which takes a lot of time when the Data log is huge.
- So, it checkpoints the index by writing the Hash Index and a pointer to the last log entry to the flash, which can then use the checkpoint as a starting point to reconstruct the in-memory Hash Index quickly.
- The invalidated hash table entry is not immediately committed to non-volatile storage to avoid random writes, so a failure following a delete requires a log to ensure that recovery will delete entry upon reconstruction. Hence the Delete entry is required.
Thank You