FAWN: A Fast Array of Wimpy Nodes

David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, Vijay Vasudevan

Presented By,
Elangovan KN

Introduction

- Storage systems have been growing in importance over the years.
- These storage systems aim to provide fast access to the clients for both read and write operations.
- Most algorithms proposed improves the performance of the storage systems by increasing power consumption.
- Key-Value storage systems is one type which has been growing in terms of both size and importance.

Key-Value Systems

- Key-Value Systems have the following characteristics
  - I/O intensive and not computation intensive.
  - Random access over large datasets.
  - Objects stored are of small size (100s of bytes to around 1KB).
  - Massively parallel.
  - Mostly independent operations.
  - Requires large clusters to support them.
- Key-Value systems have keys that are mapped to particular values.
- Amazon's Dynamo, Facebook 's Memcached, LinkedIn's Voldemort, Google's MapReduce technology and Yahoo's Hadoop-based equivalent are some examples for Key-Value system.

Existing Solutions

- **Conventional disk-based clusters**
  - The traditional storage disk such as the magnetic disk might be used in the clusters.
  - However, the conventional disks perform poorly during random-access workloads.
  - Hence this method becomes inefficient in terms of performance.
- **DRAM-based clusters**
  - DRAMs have high performance even for random access workloads.

Existing Solutions (cntd.)

- However, storage of terabytes of data will be very expensive and power consuming.
- Two 2GB Dual Inline Memory Module (DIMMs) consume the same amount of power as a 1TB hard disk.
- These clusters power consumption can account up to 50% of the total cost.
- In the future, the datacenters that house them may need a dedicated substation to operate properly.

- Hence there is a serious tradeoff in existing solutions between power and performance.

The Problem

- Looking at these problems, the question arises,

  Can a cost-effective cluster be built that reduces the power consumption while meeting the performance requirements of these data intensive applications?
- FAWNs are the solution to the problem faced with these clusters.
- FAWN couples low-power, efficient embedded CPUs with flash storage to provide efficient, fast, and cost-effective access to large, random-access data.
- Flash is cheaper, consumes less power and has good performance for random reads.

Why FAWN?

- **Increasing CPU-I/O Gap:**
  - The gap between CPU performance and I/O bandwidth have grown.
  - Hence in the conventional methods, the bottleneck occurs in the memory bandwidth for the applications involving data intensive workloads.
  - FAWN tries to increase the CPU-I/O gap by reducing the idle times for the I/O operations while maintaining high performance.
  - Since the applications are data intensive and computationally simple, the wimpy processors focus on improving I/O rather than computations.

Why FAWN?

- **CPU power consumption:**
  - Modern processors operate at high frequencies and thus increasing power consumption.
  - They do not improve the speed of basic computation but increase the efficiency by employing branch prediction, caching, etc.
  - These methods further increase the power consumption. Even if the processors don't run at full capacity, they still consume more power.
- FAWN architecture have slower CPUs that have more transistors to improve the basic computation. Hence they execute more instructions per joule.

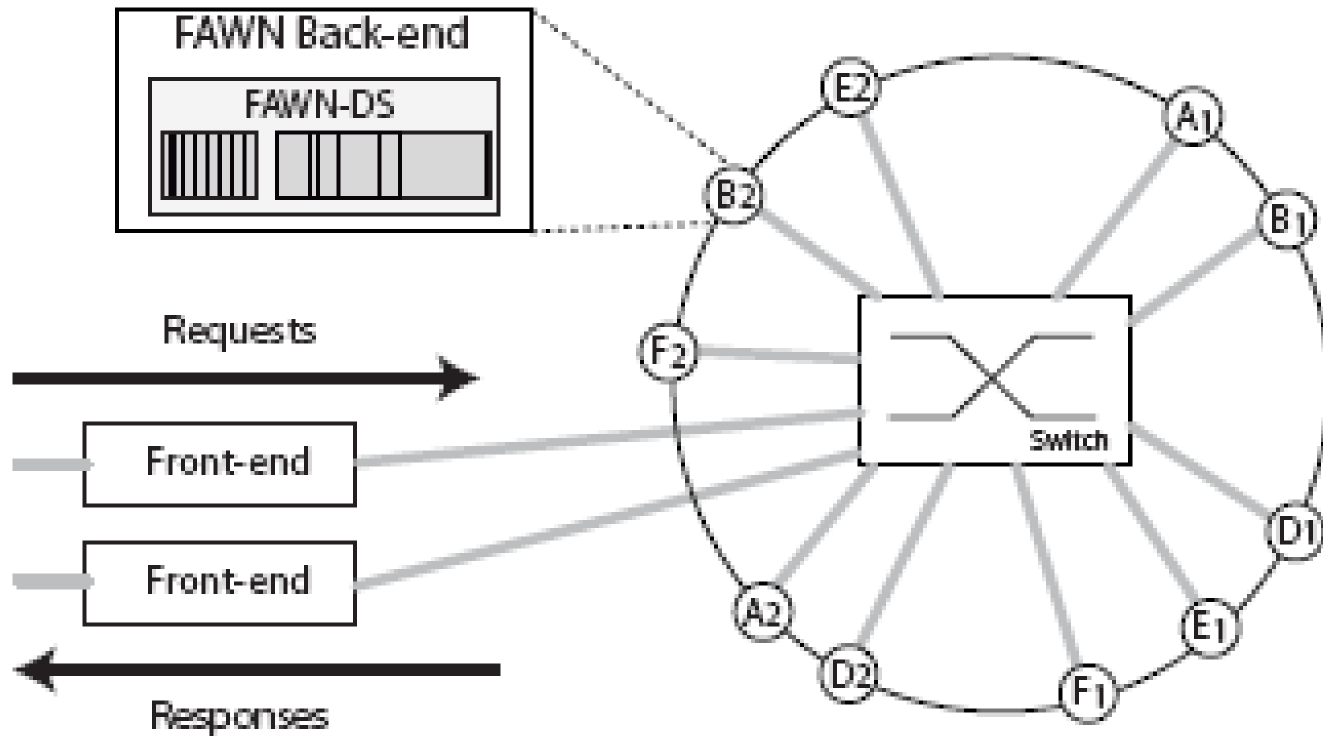
Why FAWN?

- **Dynamic Power Scaling is inefficient**.
  - Dynamic Voltage and Frequency Scaling (DVFS) tends to reduce voltage as the frequency reduces.
  - Modern CPUs are designed to already operate near the minimum voltage even at the highest frequencies.
  - Hence the systems running at 20% capacity still consumes 50% power.
- FAWN approach reduces power consumption proportionally while maintaining efficient performance at 100% load.

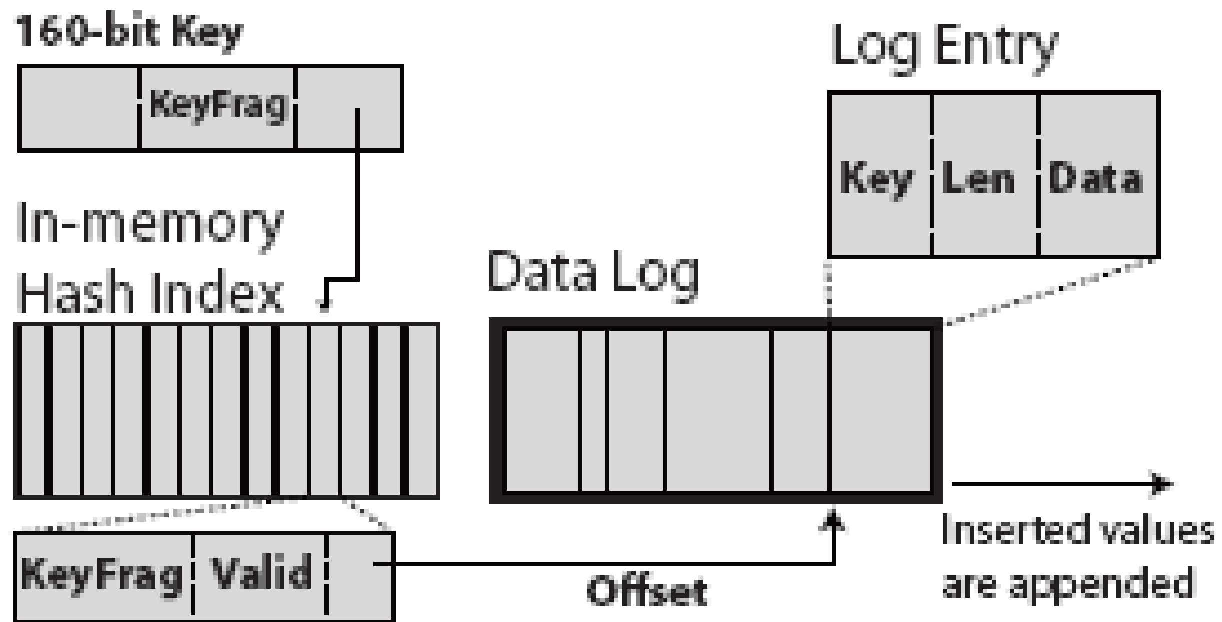
FAWN Design

- FAWN maintains front end where the client requests enter.
- The Front end forwards the request to the back end nodes.
- The back-end serves the request from its FAWN-Data Store (DS) and returns its result to the front end.
- The back-end storage nodes are organized into a ring using consistent hashing.
- Each physical node is divided into V virtual nodes and each virtual node is represented by virtual ID (VID) in the ring. Hence each node represents V key ranges.

FAWN Design



FAWN Data Store



FAWN-DS

- As said before, the FAWN-DS used flash storage memory. Since the flash have poor performance for writes, especially for random writes, the DS uses log-structure key-value store.
- Each store contains the value for the key range associated with one virtual node.
- To further improve the performance, the FAWN has DRAM hash table. This table maintains a part of the key value needed to map keys to the corresponding offset in the data Log.
- The data store is append only store with the deletions occurring during the garbage collection.

Mapping in FAWN

- FAWN uses 160-bit keys. However, the DRAM table stores only a part of the key.
- Each entry in the DRAM is 6 bytes (15 bit key fragment, a valid bit and 4-byte pointer to the DS).
- The client gives a get request using the 160 bit key.
- The FAWN extracts the *i* lower order bits of the key (index bits) and the next 15 lower order bits (key fragment).
- The index bits are used to locate a bucket in the hash index. The hash index contains 2^i hash buckets.

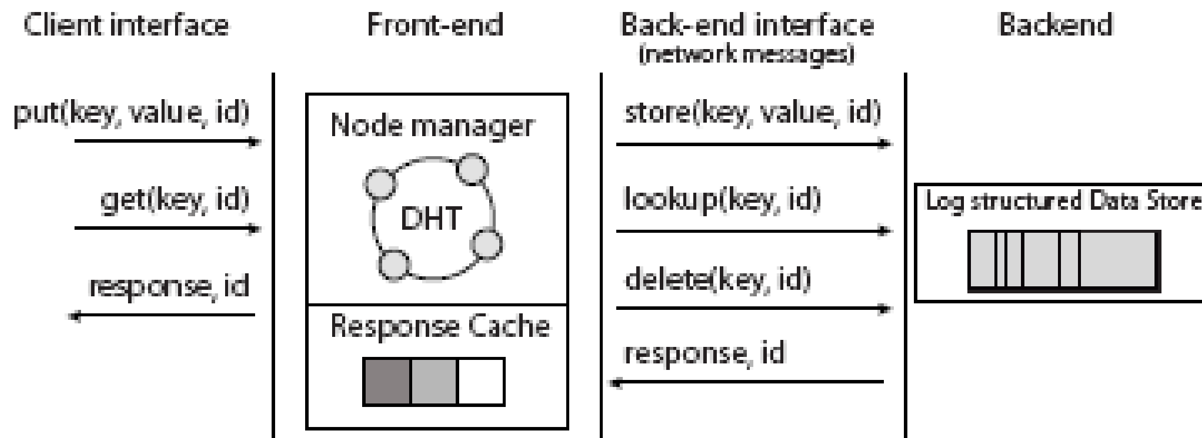
Mapping (cntd.)

- The client's key fragment and the key fragment inside the hash bucket are checked to see if they match.
- Then the 4-byte pointer is used to locate the data log and then the key inside the log is checked with the client's key.
- A mismatch in one of these two cases causes the FAWN to search the rest of the hash table.
- There is also a possibility of the key fragment to point to some other key. This will result in two reads instead of one.
- These cases occur approximately once in 32,768 times .

Pseudocode for Mapping

```
/* KEY = 0x93df7317294b99e3e049, 16 index bits */
INDEX = KEY & 0xffff; /* = 0xe049; */
KEYFRAG = (KEY >> 16) & 0x7fff; /* = 0x19e3; */
for i = 0 to NUM_HASHES do
    bucket = hash[i](INDEX);
    if bucket.valid && bucket.keyfrag==KEYFRAG &&
       readKey(bucket.offset)==KEY then
        return bucket;
    end if
    {Check next chain element...}
end for
return NOT_FOUND;
```

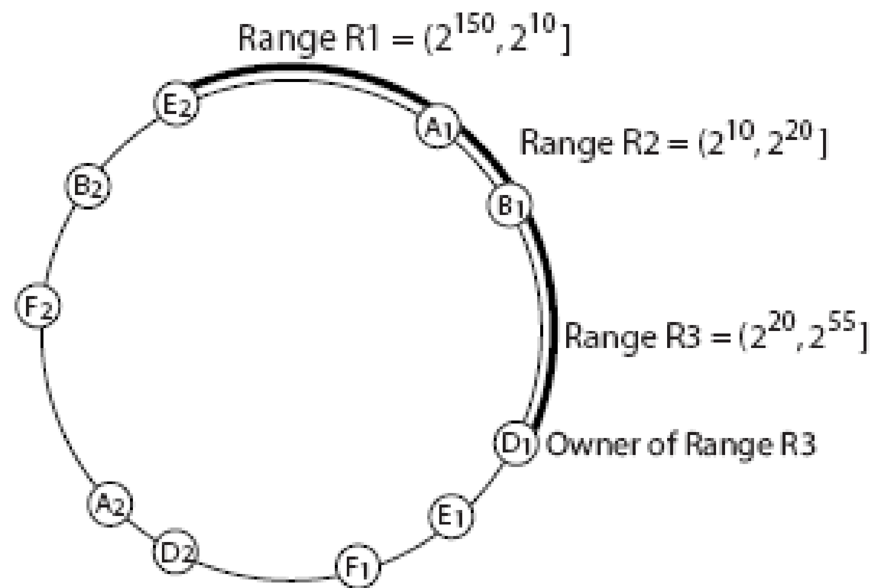
FAWN KV Interface



Front end to Back end mapping

- Each Front end maps to around 80 or more back-ends.
- Each Front end maintains the list of VIDs under it and hence the range of keys for which it has to serve back.
- If the Front end gets a query outside its key range, it forwards the query to the proper node.
- When a new back-end node joins, it gets a list of front-end nodes. Each virtual node uses this list to identify its corresponding front end node and join it.
- This mechanism gives robustness to the architecture in case of front end failures.

Consistent Hashing



Basic Functions

- The basic functions of the DS includes store, lookup and delete.
  - **Store**- appends an entry to the log, updates the hash entry to point to the log and sets the valid bit to true.
  - **Lookup**- retrieves the stored data.
  - **Delete**- changes the valid bit to false and writes a Delete entry to the end of the data. This entry is for fault tolerance. In case of hash table failure, the entry makes sure that the data is deleted when the hash table is being reconstructed.

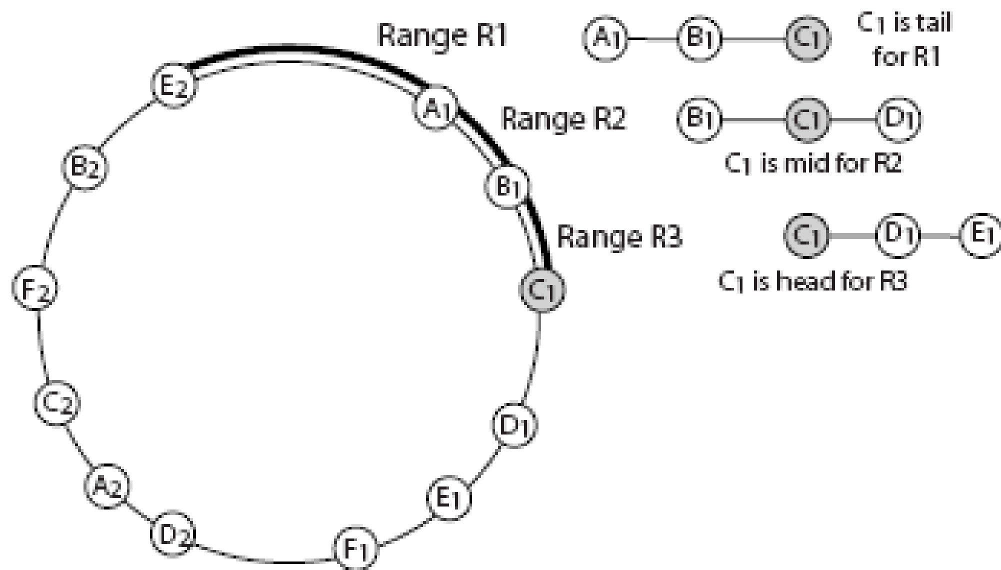
Caching

- The FAWN maintains a two level caching.
- Back end nodes maintain a cache of recently accessed data in their buffer cache.
- While the normal speed is 1300 queries per second, the back end cache can serve up to 85000 queries per second.
- The Front end also maintains a small, high speed cache that helps in serving repeated requests to the same back end.

Reconstruction of Hash table

- Hash table can be reconstructed in case of any failures.
- To facilitate this, the FAWN-DS stores the checkpoints of hash table periodically.
- After a failure, the FAWN refers to the latest checkpoint to reconstruct the hash table.

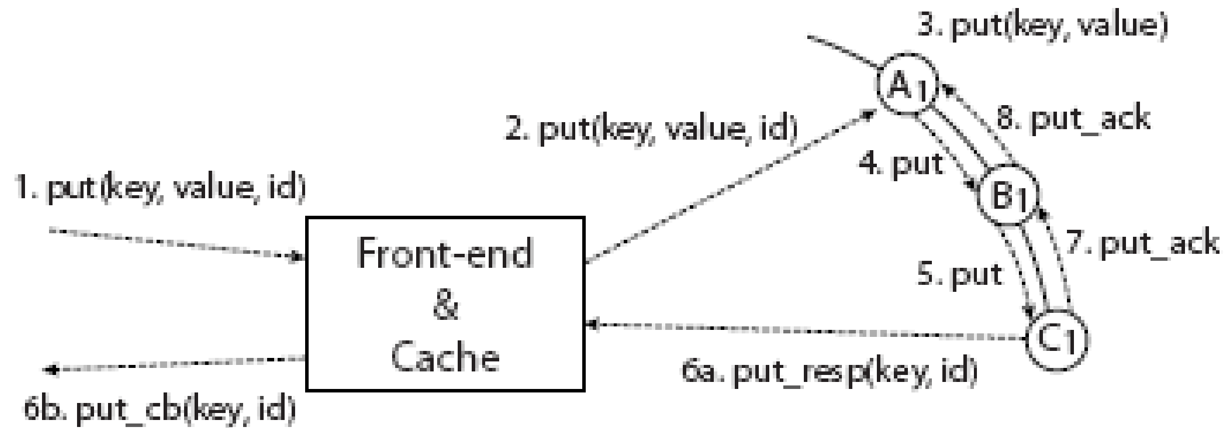
Replication



Replication

- Each Range of keys are replicated and stored in R nodes in the ring.
- The nodes remain as the head for one range, tail for one range and mid for R-2 chains.
- The owner usually remains the head for the chain.
- The put command is sent to the head from where the command is forwarded to each node for that range. The tail sends the response back to front end and each node acknowledges its preceding node for the range.
- Get command is sent to the tail which responds to the request.

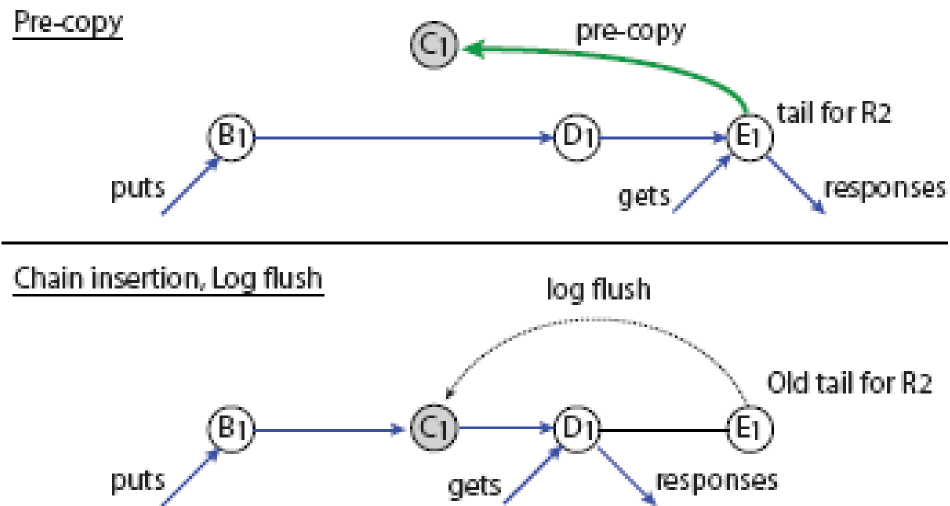
Replication



Join

- When a node joins the FAWN ring,
  - A single key range is split into two.
  - The new node must receive copies R ranges of key
  - The front end should get the information of the node and must start treating it as a head or tail for appropriate ranges.
  - The older nodes that had the copies of the ranges must erase those copies.
- Before the node officially joins, the tail for each range copies their contents into the new node through pre-copy procedure.
- After the pre-copy, the front end sends a chain membership message through each chain to inform the other nodes of the new member.
- The tail node is also changed because of the insertion and all nodes are made aware of it.

Join



Leave

- Here the key ranges have to be merged instead of splitting.
- The process are almost similar to the join.

Maintenance

- The joining of a new node is done by the split operation. Leave is done by the merge operation.
- These operations parse the data log and make the changes to the new data log.
- While these operations are performed, the log is locked to make sure there is no access of outdated datas.

Failure Detection

- Each front end exchanges heartbeat messages with its backend nodes every t_{hb} seconds.
- If a node missed $fd_{threshold}$ heartbeats, the front-end assumes it to have failed and initiates leave protocol.
- A failure similar to this during join will not affect the normal structure as long as the majority of the data has not been transferred.

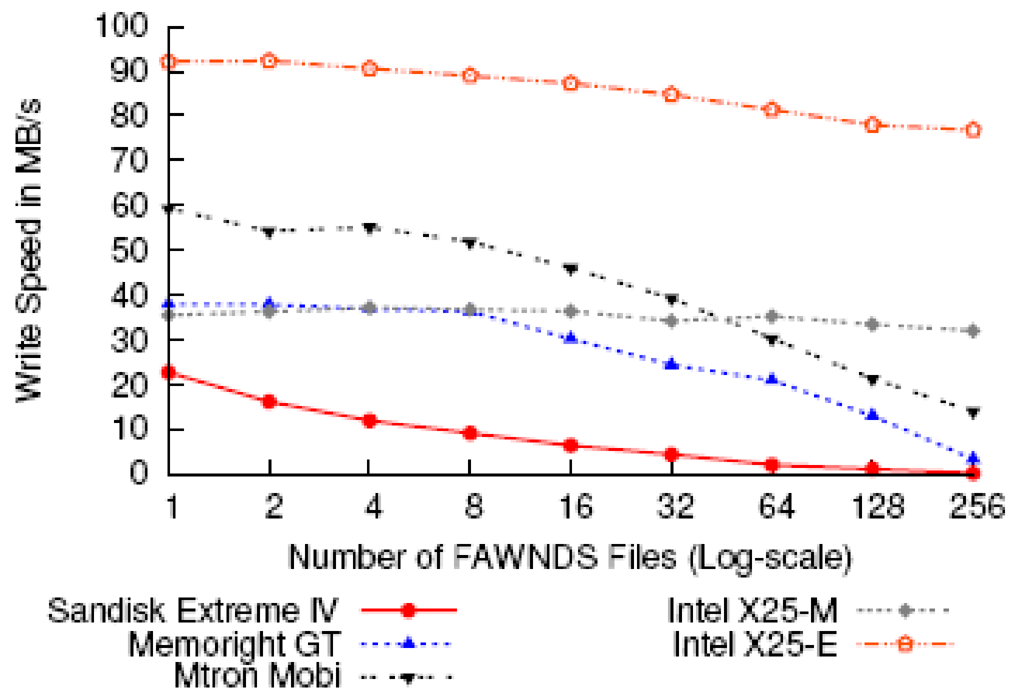
Evaluation – System Model

- Single-core 500 MHz AMD Geode LX processor,
- 256 MB DDR SDRAM operating at 400 MHz,
- 100 Mbps Ethernet.
- 21 back end nodes, each node contains one 4 GB Sandisk Extreme IV Compact Flash device.
- The nodes are connected to each other and to a 27 W Intel Atom-based front-end node using two 16-port Netgear GS116 GigE Ethernet switches.
- The object sizes vary from 256 byte to 1KB.

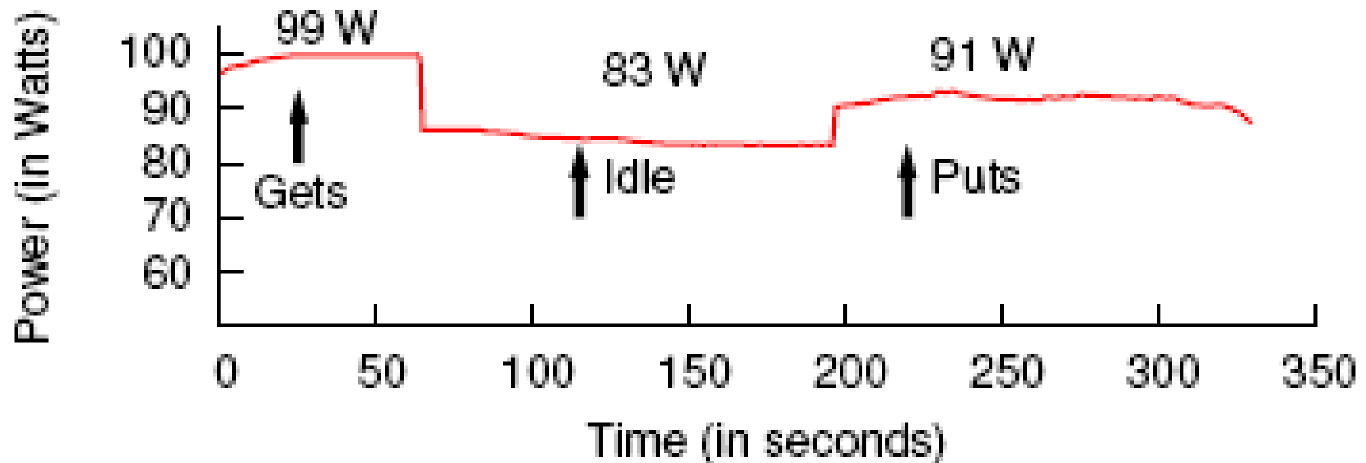
Local random read performance

DS Size	*1 KB Rand Read* (in queries/sec)	*256 B Rand Read* (in queries/sec)
10 KB	72352	85012
125 MB	51968	65412
250 MB	6824	5902
500 MB	2016	2449
1 GB	1595	1964
2 GB	1446	1613
3.5 GB	1150	1298

Semi-Random write performance



Power consumption of the system for 256 B values during Puts/Gets



Traditional and FAWN node statistics

System	Cost	W	QPS	Queries/Joule	GB/Watt	TCO/GB	TCO/QPS
Traditionals:							
5-2TB HD	$2K	250	1500	6	40	0.26	1.77
160GB PCIe SSD	$8K	220	200K	909	0.72	53	0.04
64GB DRAM	$3K	280	1M	3.5K	0.23	59	0.004
FAWNs:							
2TB Disk	$350	20	250	12.5	100	0.20	1.61
32GB SSD	$500	15	35K	2.3K	2.1	16.9	0.015
2GB DRAM	$250	15	100K	6.6K	0.13	134	0.003

Questions?