Key-Value Stores

Michael Kaminsky

David Andersen, Bin Fan, Jason Franklin, Hyeontaek Lim,
Amar Phanishayee, Lawrence Tan, Vijay Vasudevan

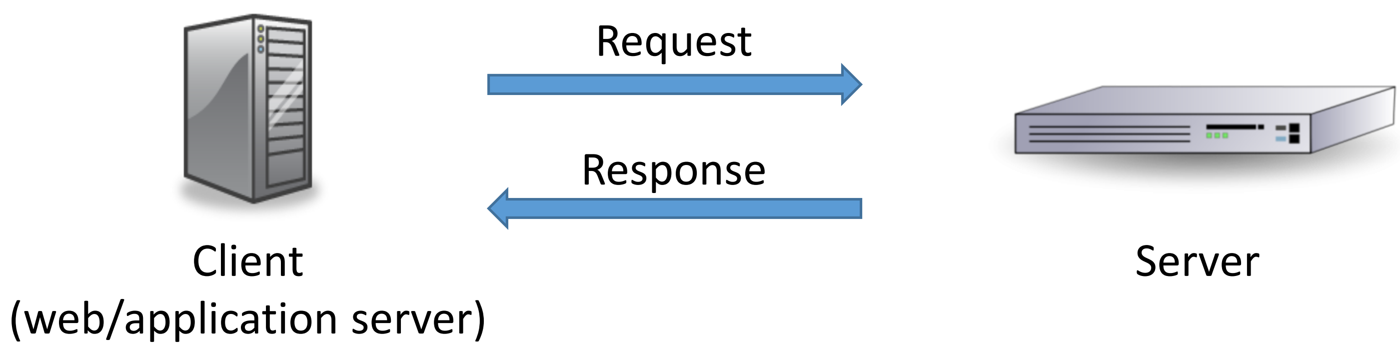
Carnegie Mellon University
Advanced Cloud Computing (15-719)
February 27, 2017

What is a Key-Value store?

- At the simplest level:

  val = GET(*key*)

  PUT(*key*, *val*)



What is a Key-Value store?

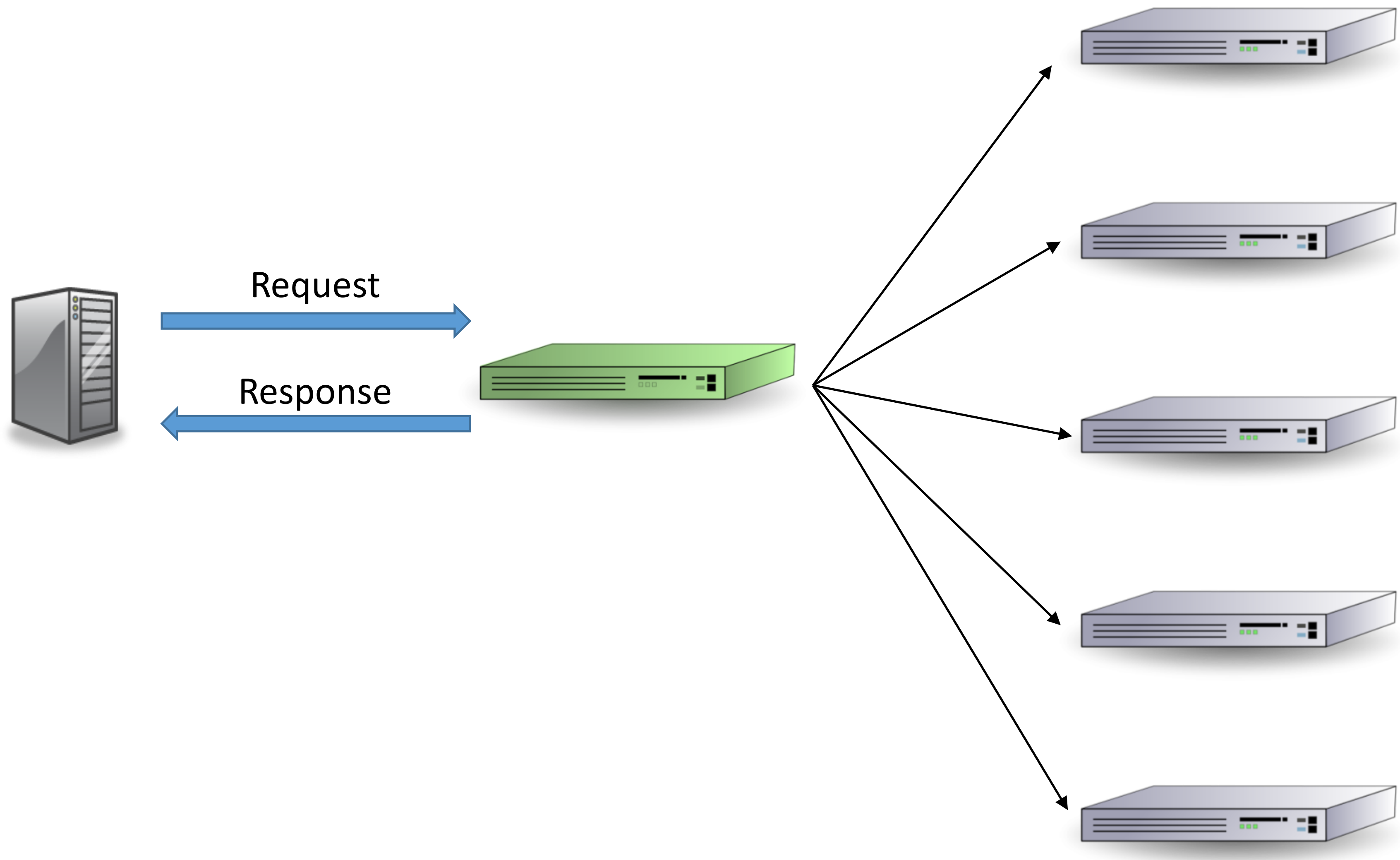
- Can have more complicated interfaces
  - DELETE()
  - INCREMENT()
  - COMPARE_AND_SET()
  - Range Queries
  - MultiGET(), MultiPUT()
  - UPSERT(key, lambda…)
  - …

Where are KV stores used?

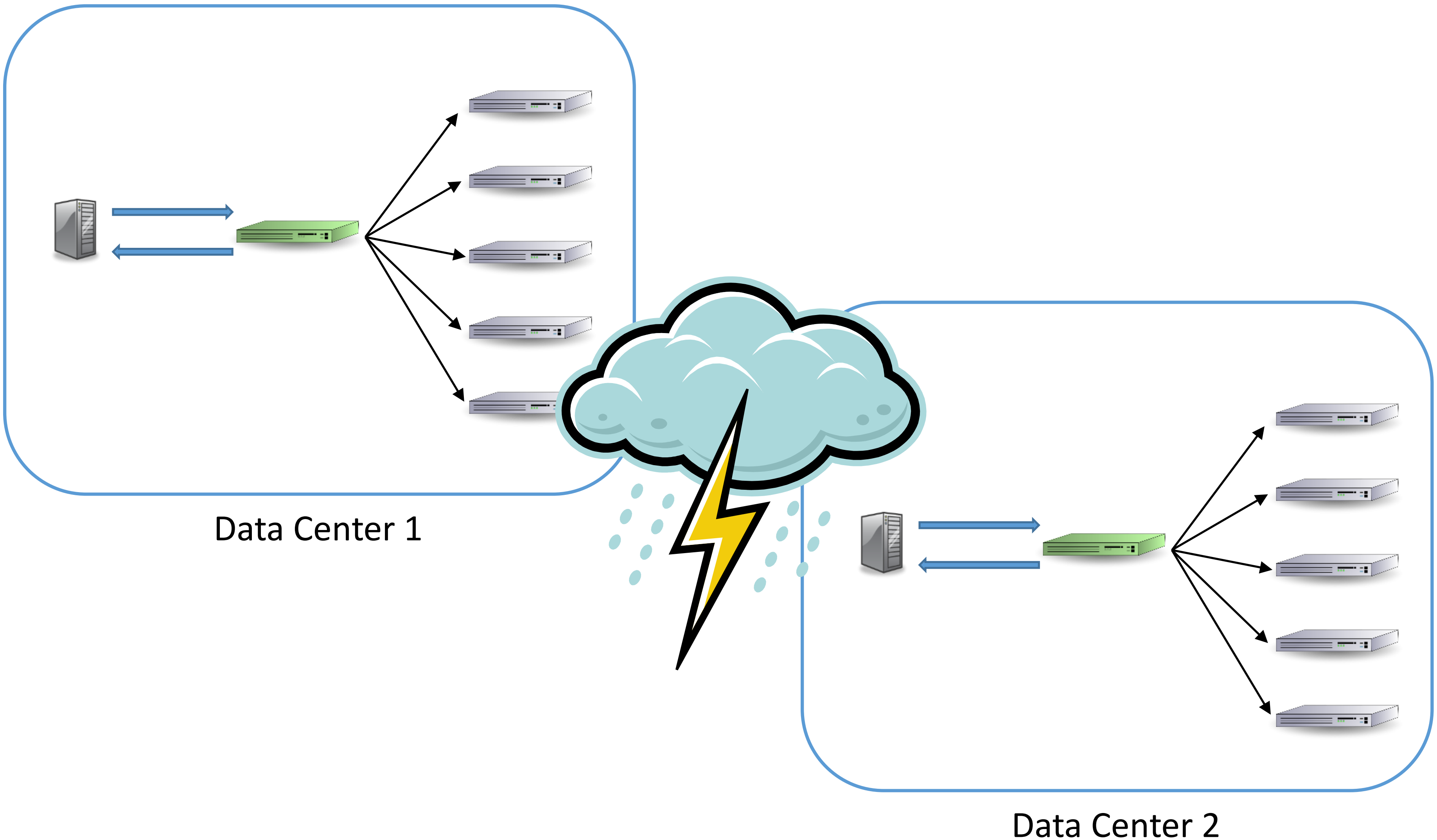
- Everywhere!
  - Amazon – Dynamo → ElastiCache (memcached/redis)
  - Facebook – memcached
  - Google – LevelDB
  - Twitter

Rendering a page requires hundreds of KV queries!

Multiple-node KV-store



Geographically distributed KV-store



What do keys look like?

- Plain text "kaminsky"
- Hashed 0x6337dfad…

- What are the tradeoffs?
  - Plain text keys provide
    - Potential for range queries
    - Sorted vs unsorted
  - Hash keys provide:
    - Potentially smaller/fixed-size keys
    - Load balancing

What do values look like?

- Usually opaque blob (e.g., memcached)
  - Fixed vs. variable-length
  - Could consider having serialized objs; client manipulates
- Might have limited semantic meaning
  - E.g., for INCREMENT()
  - E.g., in Redis, values can be lists, sets, tables, etc.
- How big are KV Pairs?
  - Usually small: 64 bytes, 1K, etc.
  - Overhead matters

How do KV stores fit into the landscape?

- **Typical file systems**
  - Hierarchical directory structure
  - Aimed at much bigger objects (e.g., files)
  - Often, allow modifications of partial objects
- **Relational Databases (RDBMS)**
  - More sophisticated data model; schemas
  - Interface: SQL, joins, etc.
  - Cross-key operations, locking, *transactions*, secondary indices
- **Other data models / NoSQL data stores**
  - Document-oriented (e.g., CouchDB, MongoDB)
  - Column-store (e.g., BigTable, Cassandra, HBase)
  - Provide more capability as the expense of complexity/performance

The lines are very blurry

Today's Focus

- Values are opaque blob
- Small objects
- High throughput / low latency
  - Comes from: simplicity and specialization

- Using three examples
  - An in-memory KV cache: Memcached
  - An on-flash KV storage: FAWN-DS
  - A (local area) distributed KV storage: FAWN-KV

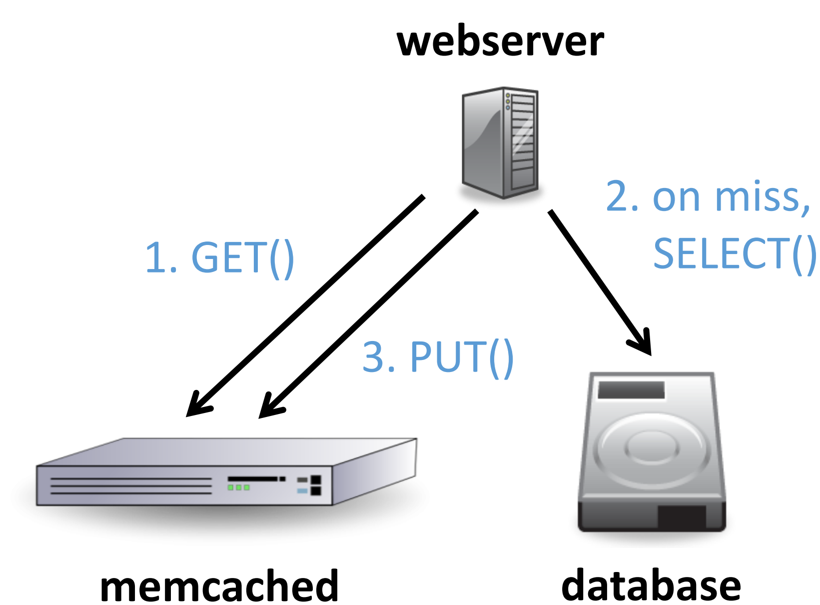
Single-node KV – considerations

Storage Semantics / Key Order	**Unordered (hash table)**	**Ordered (tree)**
Cache (eviction)	Memcached [DRAM]	
Durable Store	FAWN-DS [Flash] RAMCloud [DRAM->disk] Dynamo [disk]	LevelDB [disk] Masstree [DRAM->disk]

- **DRAM**: Low latency/high throughput (SLOs); smaller capacity; high cost/byte
- **Disk**: Persistence; large capacity; low cost/byte
- **Flash**: between DRAM and Disk; different kind of beast
- **Next Gen NVM (e.g., PCM)**: between DRAM and Flash. Coming soon… ?

Example: Memcached

- Very popular single node, in-memory KV store
  - Originally developed for LiveJournal
  - YouTube, Reddit, Facebook, Twitter, Wikipedia, …
  - Often used to cache database queries
    - Key = hash of the SQL
      Val = data returned from backend RDBMS
  - Or, e.g., online status:
    - Key = username
      Value = available, busy, …

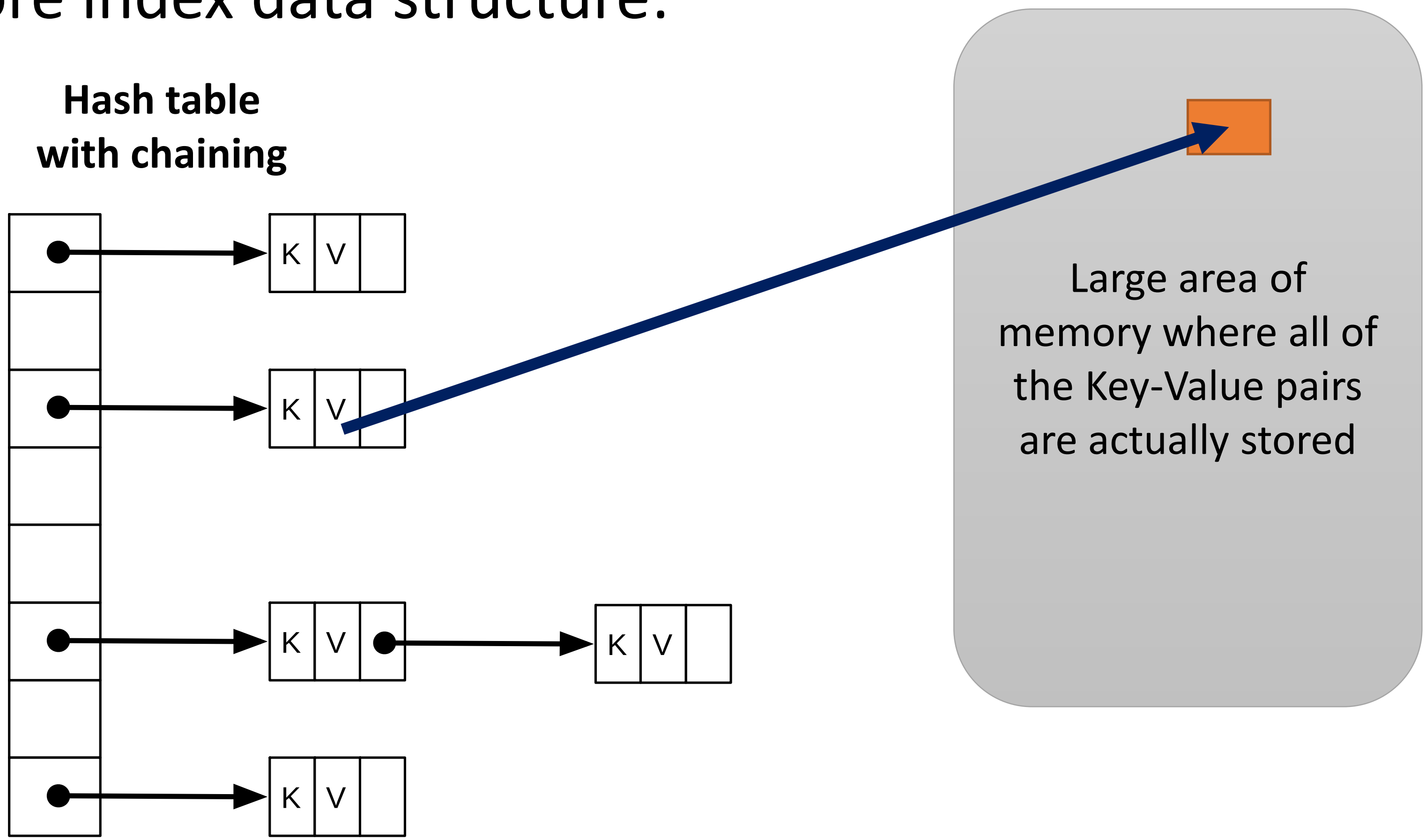


Typical Memcached use cases

- Often used for small objects (FB[Atikoglu12])
  - 90% keys < 31 bytes
  - Some apps only use 2-byte values
- Tens of millions of queries per second for large memcached clusters (FB[Nishtala13])
- Read-mostly workloads

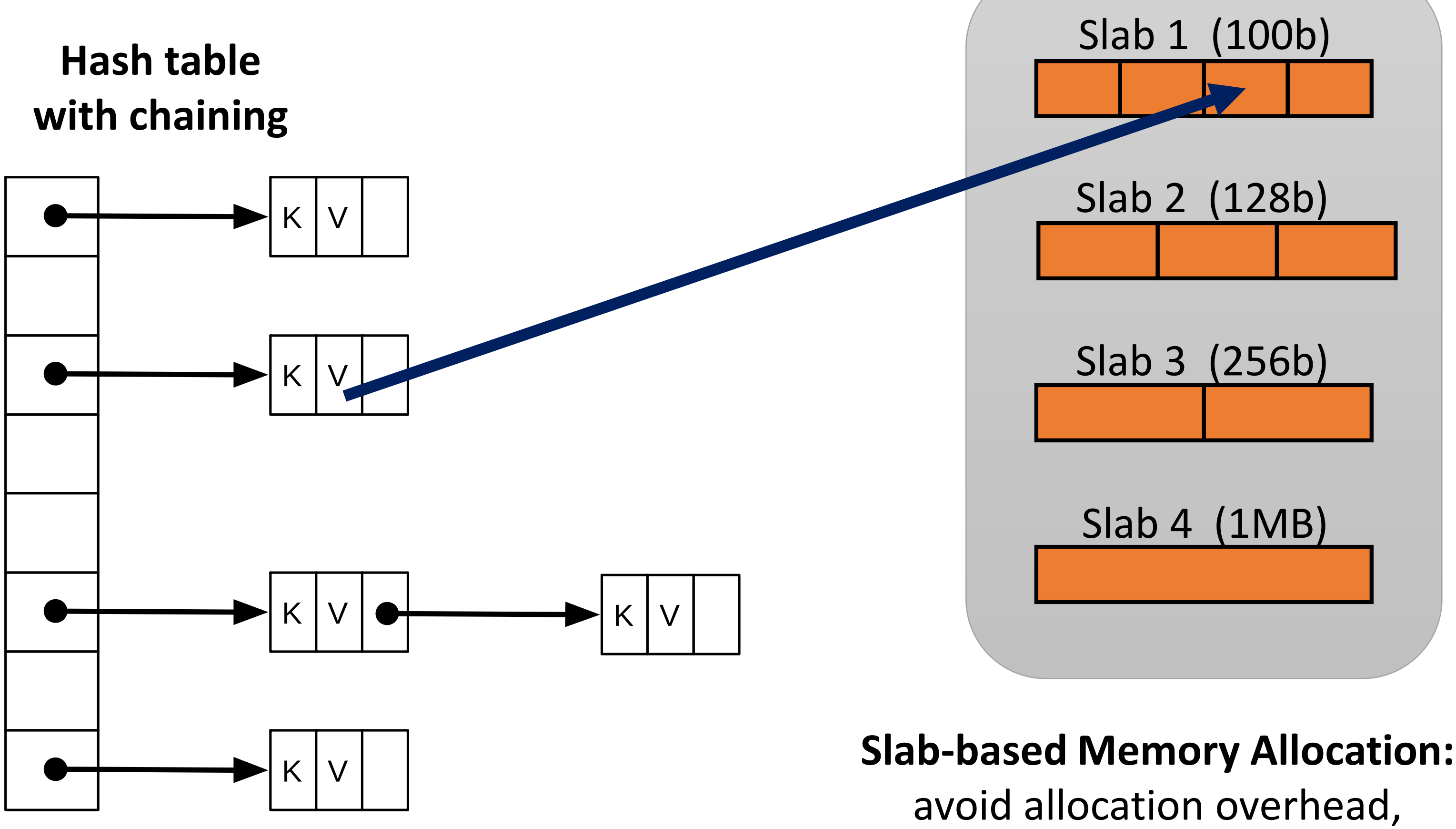
Memcached Design

- Core index data structure:



Memcached Memory Management

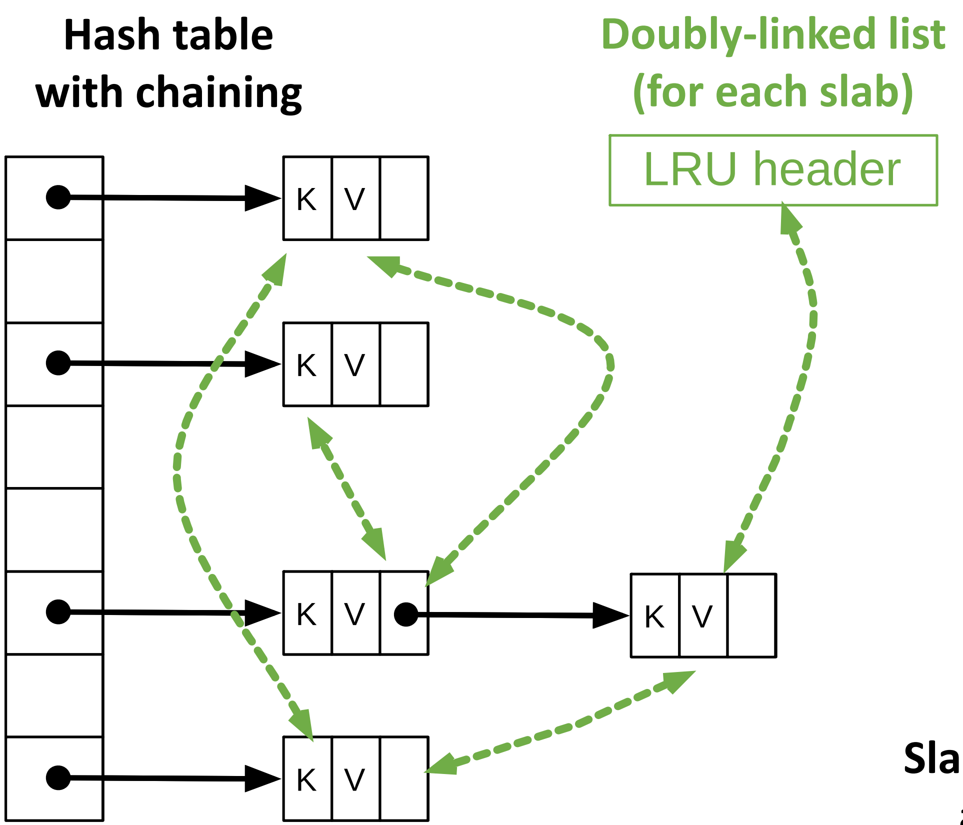
- Core index data structure:

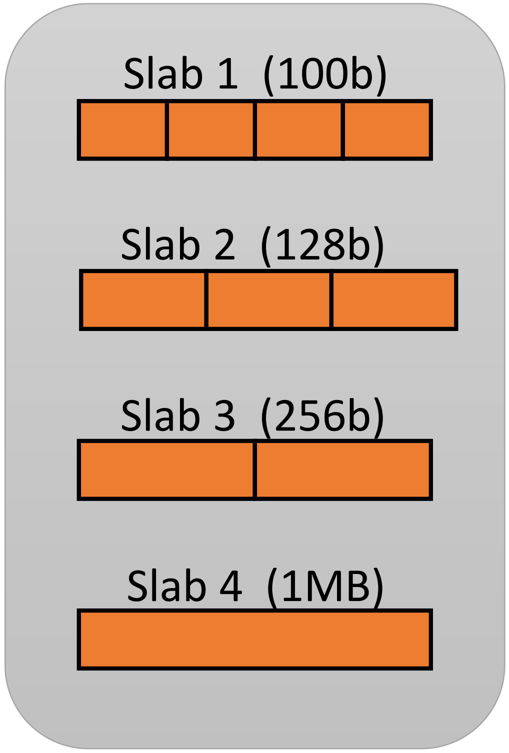


Slab-based Memory Allocation:
avoid allocation overhead,
reduce fragmentation, re-use memory

Memcached Eviction

- Core index data structure:





Slab-based Memory Allocation:
avoid allocation overhead,
reduce fragmentation, re-use memory

Problems with Memcached design

- Single-node scalability and performance
  - Poor use of multiple threads
  - Global locks serialize access to hash table and LRU list
    - Every *read* updates the LRU list
  - Lots of sequential pointer chasing

- Space overhead – affects # items stored & cost
  - 56-byte header per object
    - Including 3 pointers and 1 refcount
    - For a 100B object, overhead > 50%
  - Poor hash table occupancy

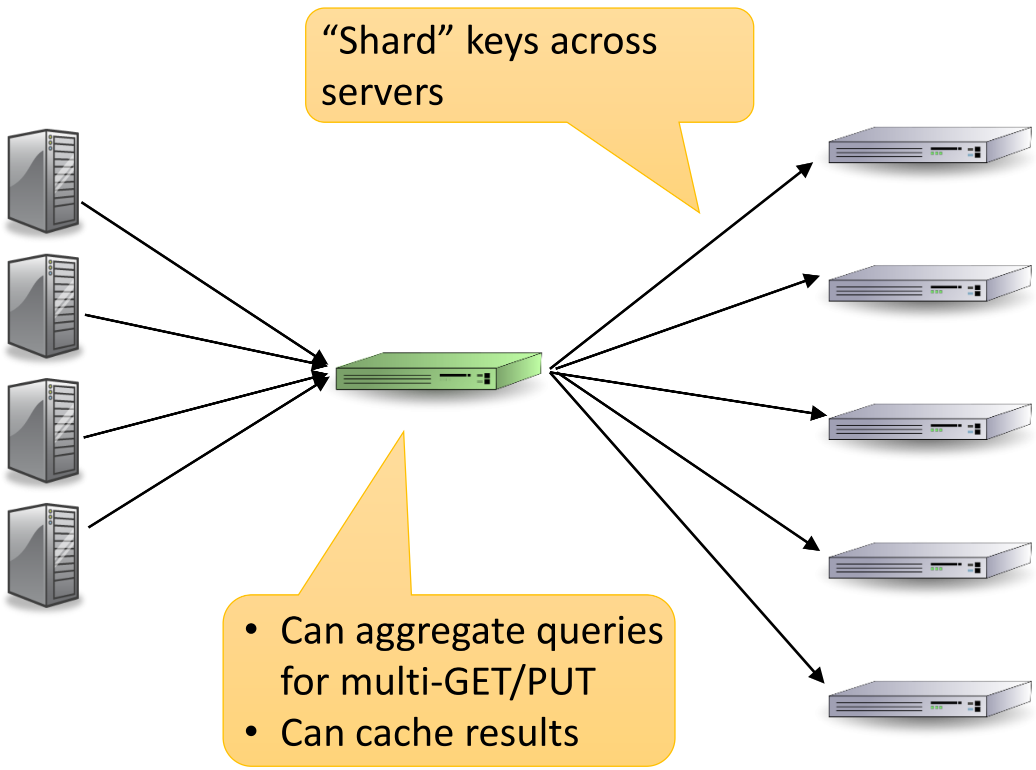
MemC3 [Fan, NSDI'13]

- Core hash table uses *optimistic cuckoo hashing*
  - Higher concurrency:
    - single-writer/multi-reader
    - Lookups can be parallelized
  - Better memory efficiency:
    - No pointers
    - 95% hash table occupancy
- CLOCK-based eviction (approximates LRU)
  - Better space efficiency and concurrency

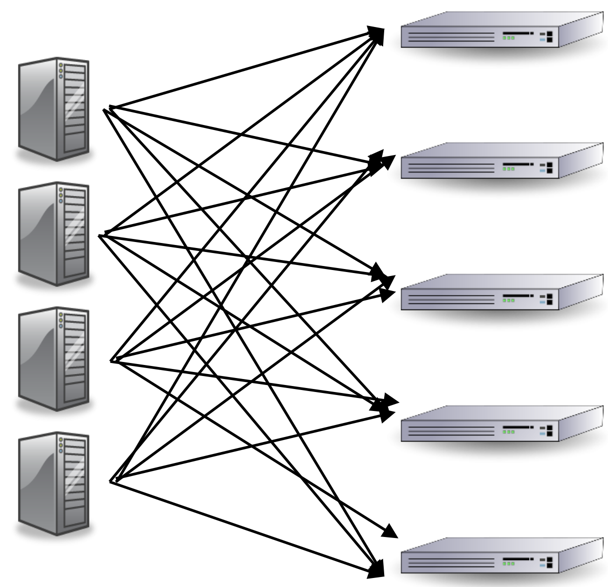
Further reading about single-node KV stores:

- Concurrent Cuckoo Hashing [Li, EuroSys'14]
- MICA [Lim, NSDI'14/ISCA'15]
- Masstree [Mao, EuroSys'12]
- HERD [Kalia, SIGCOMM'14]

Multi-node Memcached Clusters



Clients route requests through a request redirector/load balancer



Clients talk directly to memcached servers

Single-node KV – considerations

Storage Semantics / Key Order	**Unordered (hash table)**	**Ordered (tree)**
Cache (eviction)	Memcached [DRAM] (flash-based memcached)	
Durable Store	FAWN-DS [Flash] RAMCloud [DRAM->disk] Dynamo/S3 [disk]	LevelDB [disk] Masstree [DRAM->disk]

What changes when moving from <u>cache</u> to <u>store</u>?

What changes when moving from <u>DRAM</u> to <u>flash</u>?

Comparison of storage technologies

- DRAM, Flash, and Disk are very differ *interestingly different*

"Newer" PCIe3.0 SSD
2800 MB/s
1900 MB/s
460,000 IOPS
90,000 IOPS

	DRAM	NAND Flash/SSD	Disk
Sequential Read	10 GB/s	500 MB/s	100 MB/s
Sequential Write	10 GB/s	315 MB/s	100 MB/s
Random Read	10s millions/s	35,000 IOPS	150 IOPS
Random Write	10s millions/s	300-8,600 IOPS	150 IOPS
Durability	volatile	persistent	persistent
Lifetime	infinite	1-10K write cyc	infinite

Numbers from around
FAWN-DS era SSD

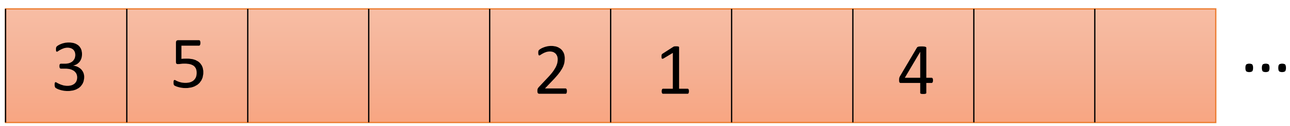
Fast Random-Reads
Slow(er) Random-Writes

Flash erase blocks

- NAND flash has limited Program/Erase (P/E) cycles
  - All SSDs use a Flash Translation Layer (FTL) to mitigate
  - Wear-leveling
- NAND flash cannot overwrite existing data
  - Must be erased first
- Erasing is inefficient
  - NAND flash is organized into *erase blocks*
  - Usually 128KB – 512KB
  - Must erase a whole block before re-writing
    (but you can write in pages; e.g., 512B, 4KB)
- What does this all mean for KV-stores…

What if we just write hash table to flash directly in-place?

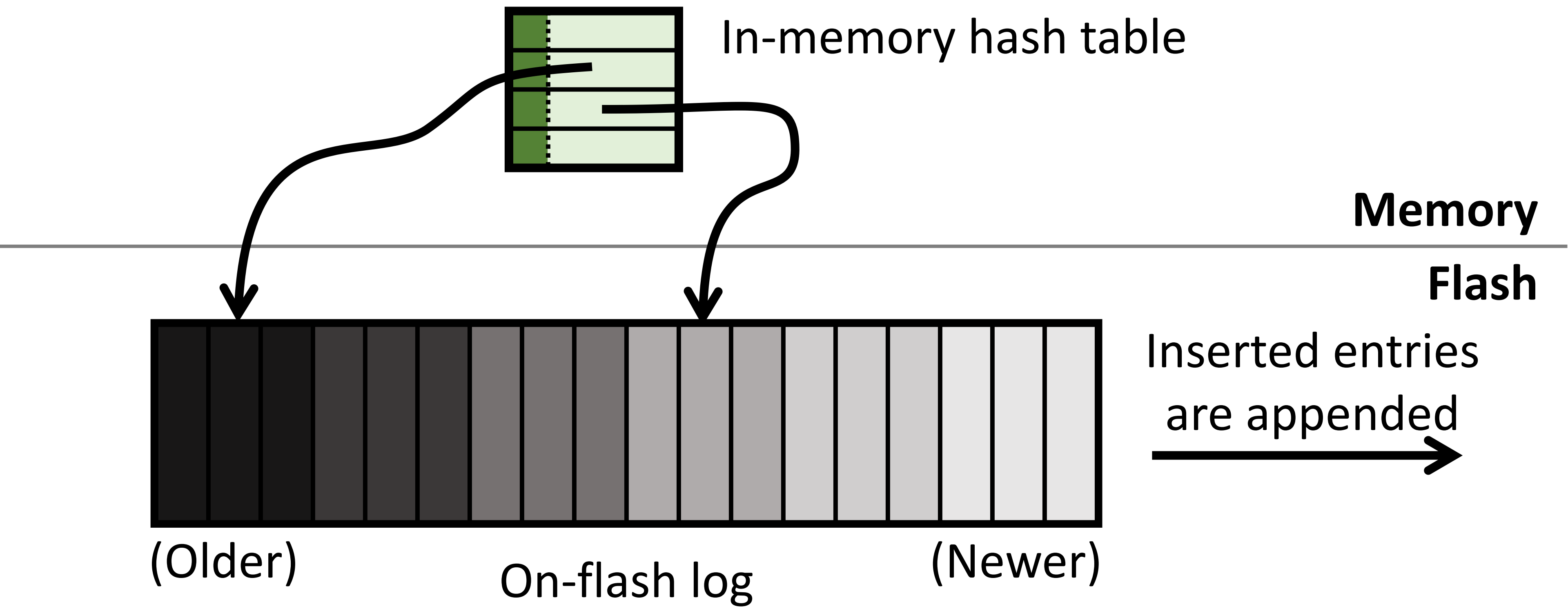
- Example: if you write 1MB as 1KB blocks, randomly to flash:



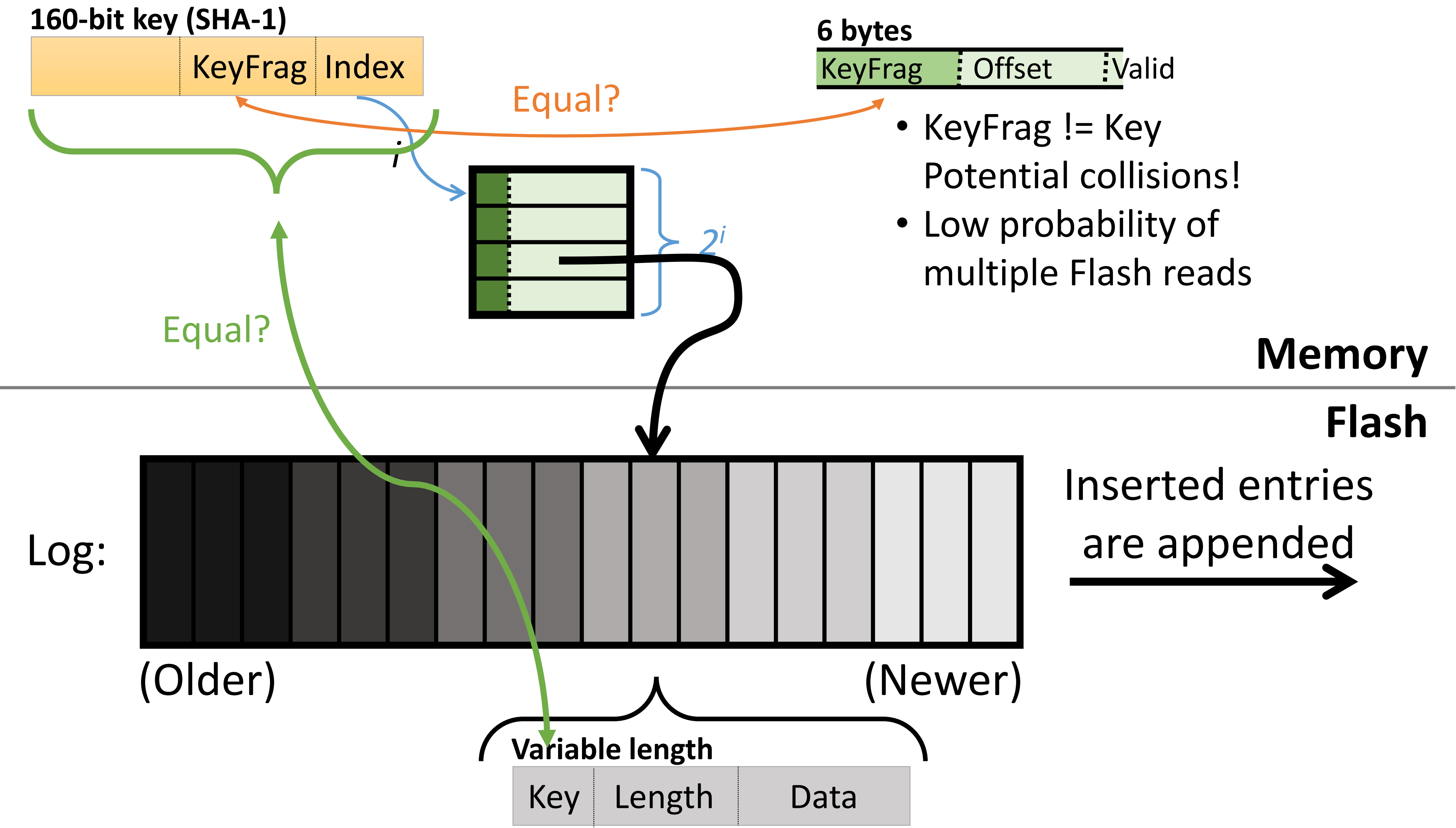
- With raw flash:
  - Each write requires reading 128KB into buffer, changing 1KB, and writing out 128KB. That's a *write amplification* of 128x
  - Thus, to write 1MB, you have to write 128MB.
  - Also, very bad for durability
- FTL helps a little
- Solution: log-structured writes

FAWN-DS: external KV store

[Andersen, SOSP'09]



FAWN-DS: GET()



- KeyFrag != Key
  Potential collisions!
- Low probability of multiple Flash reads

FAWN-DS Design Advantages

- Flash friendly:
  - GET() – Random reads
  - PUT() – Append (sequential write)
- Minimize I/O
  - Low prob. of multiple flash reads / GET()
- Memory efficient
  - “Only” 12 bytes per entry (assuming 50% load factor)
  - Modern external KV-stores use < 1 byte/index entry

FAWN-DS Design Advantages

- Reconstruction
  - On-flash Log contains all information to reconstruct index
  - FAWN-DS periodically checkpoints index and pointer to end of log to flash to speed recovery

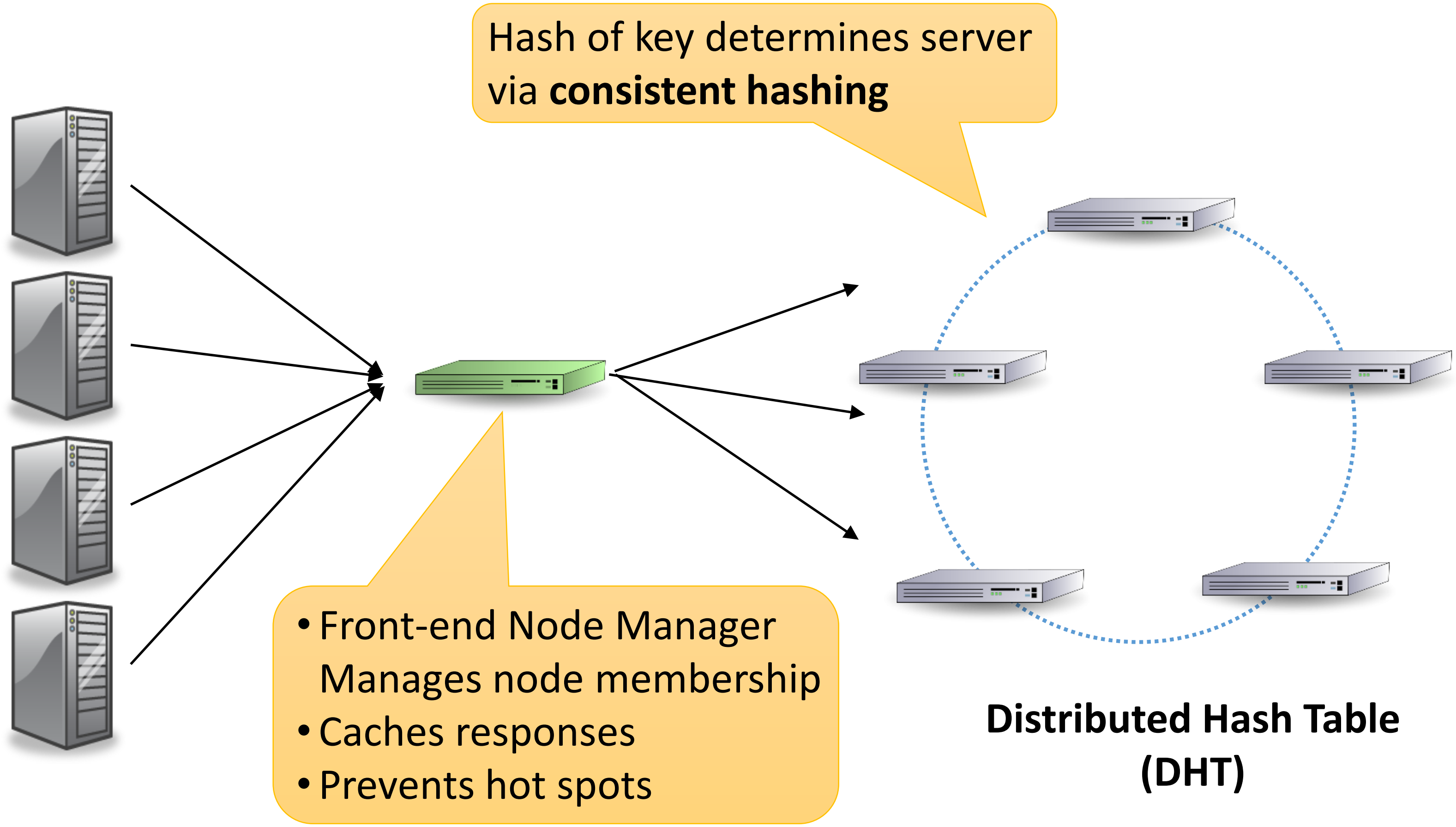
- Other operations
  - **Delete**: Write a *Delete Entry* to Log and clear the *Valid Bit*
  - **Store (PUT)**: Append to Log and update Hash Index entry
  - **Compact**: garbage collect old entries
  - **Split/Merge**: needed for FAWN-KV…coming soon

Related systems–durable store

- SILT [Lim, SOSP'11]
  - Enables very memory-efficient index: just a few bits/key with only a single flash read to retrieve value
  - Combines several KV stores into one system
  - Keep data sorted on disk (by hash of key)

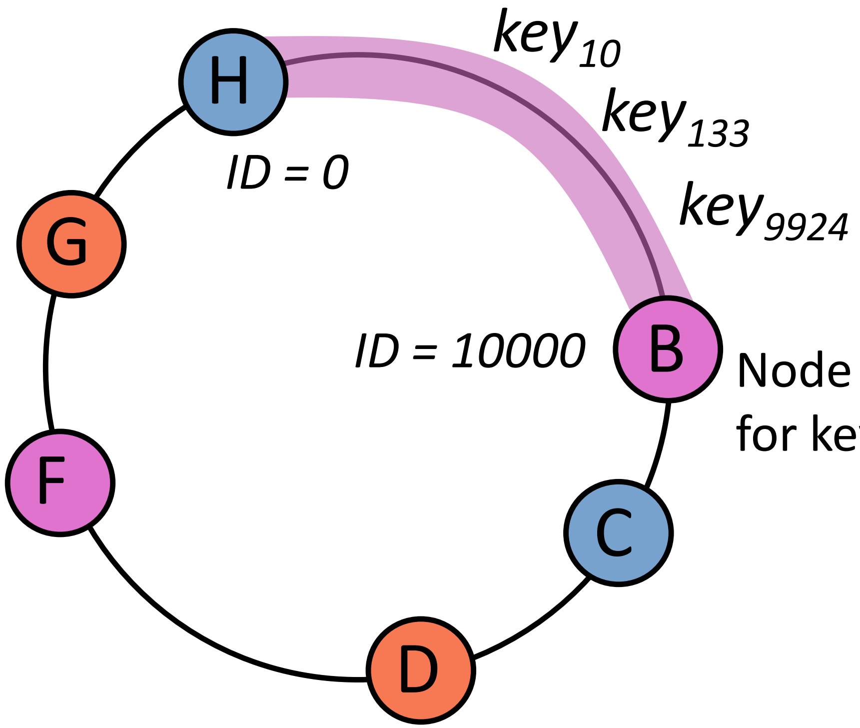
- LevelDB
  - From Google
  - Buffer and batch writes to disk (not flash)
  - Keeps on-disk data sorted by key; allows range queries
  - Lots of follow-on work (e.g., RocksDB from Facebook)

FAWN-KV: a distributed KV store



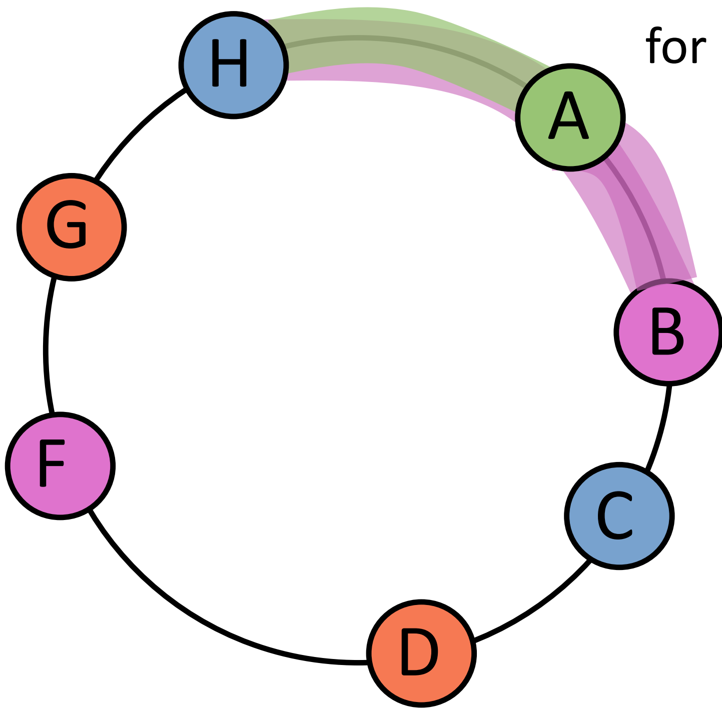
Consistent Hashing & DHTs

160-bit circular ID space for Nodes and Keys



Node B is responsible for keys in range (H,B]

FAWN-KV Join



Node A is responsible for keys in range (H,A]

Node B is responsible for keys in range ~~(H,B]~~ (A,B]

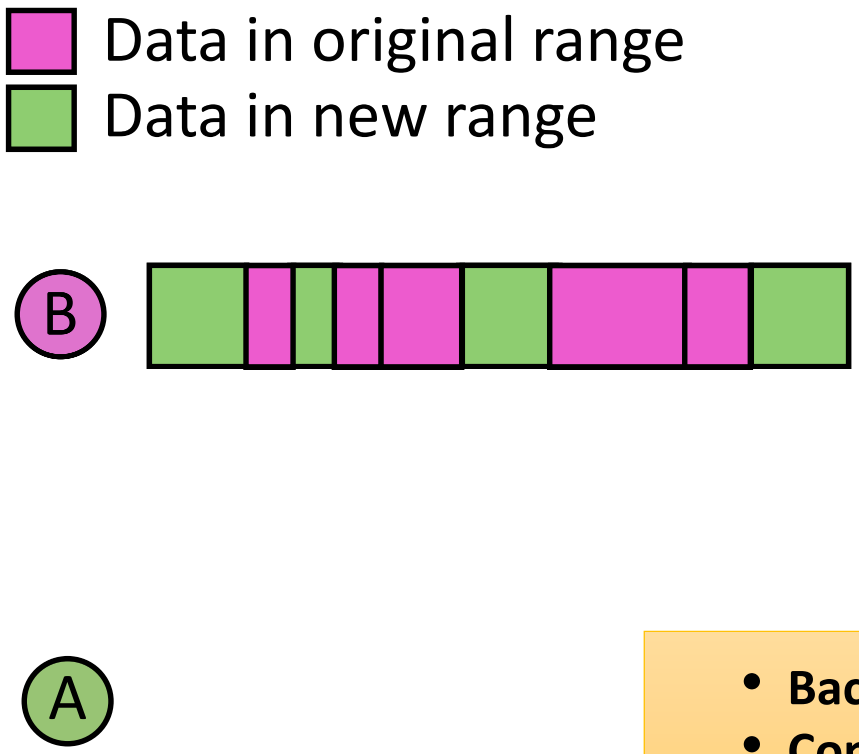
- Node additions, failures require transfer of key-ranges
- Log-structured FAWN-DS design makes this particularly efficient

FAWN-KV design choices

- DHT allows nodes to join/leave (e.g., failure) without global data movement (no “re-hashing”)
  - Need enough nodes to ensure good load balance
  - Can compensate with *virtual nodes*

- Log-structure allows for fast fail-over via sequential reads and writes; minimize time key range is locked

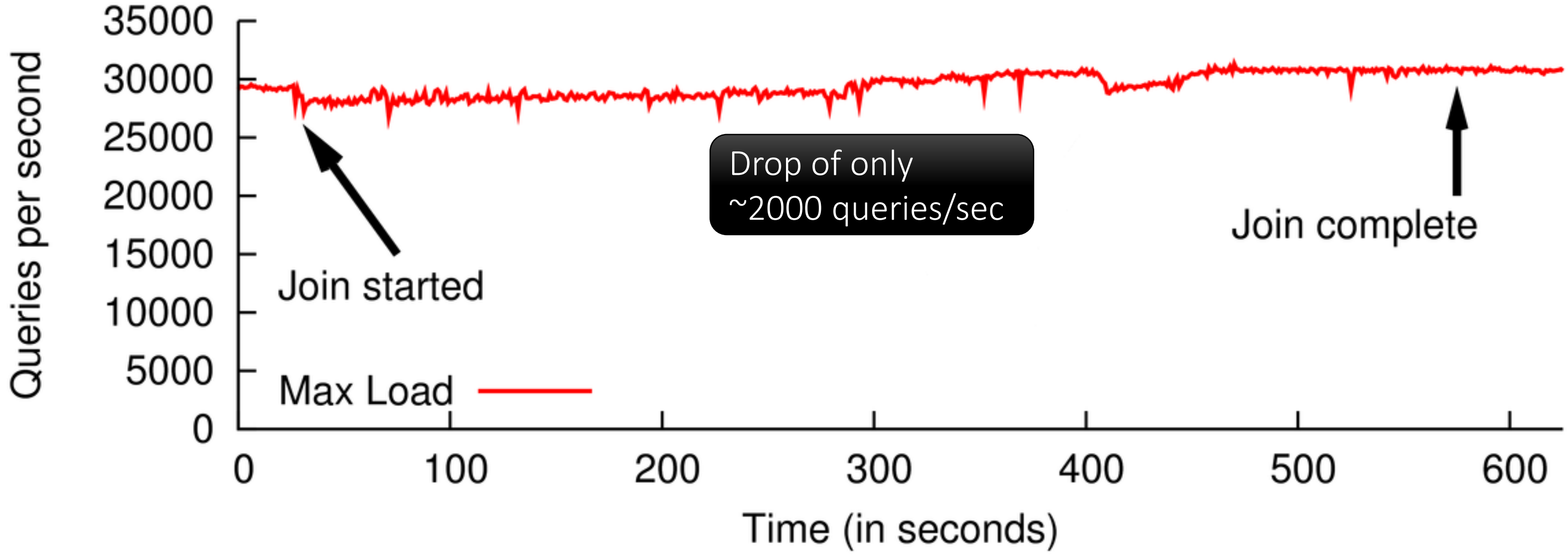
Nodes stream data range



- Stream from B to A
- Concurrent Inserts, Minimizes locking
- Compact Datastore

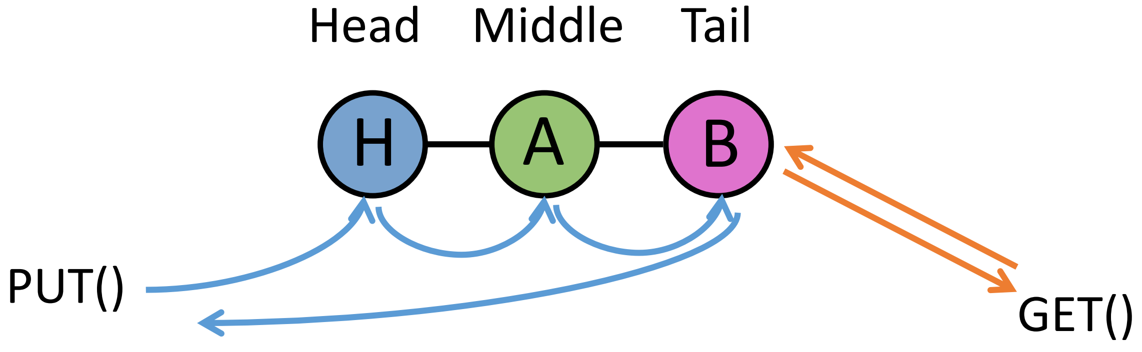
- **Background operations sequential**
- **Continue to meet SLO**

FAWN-KV performance

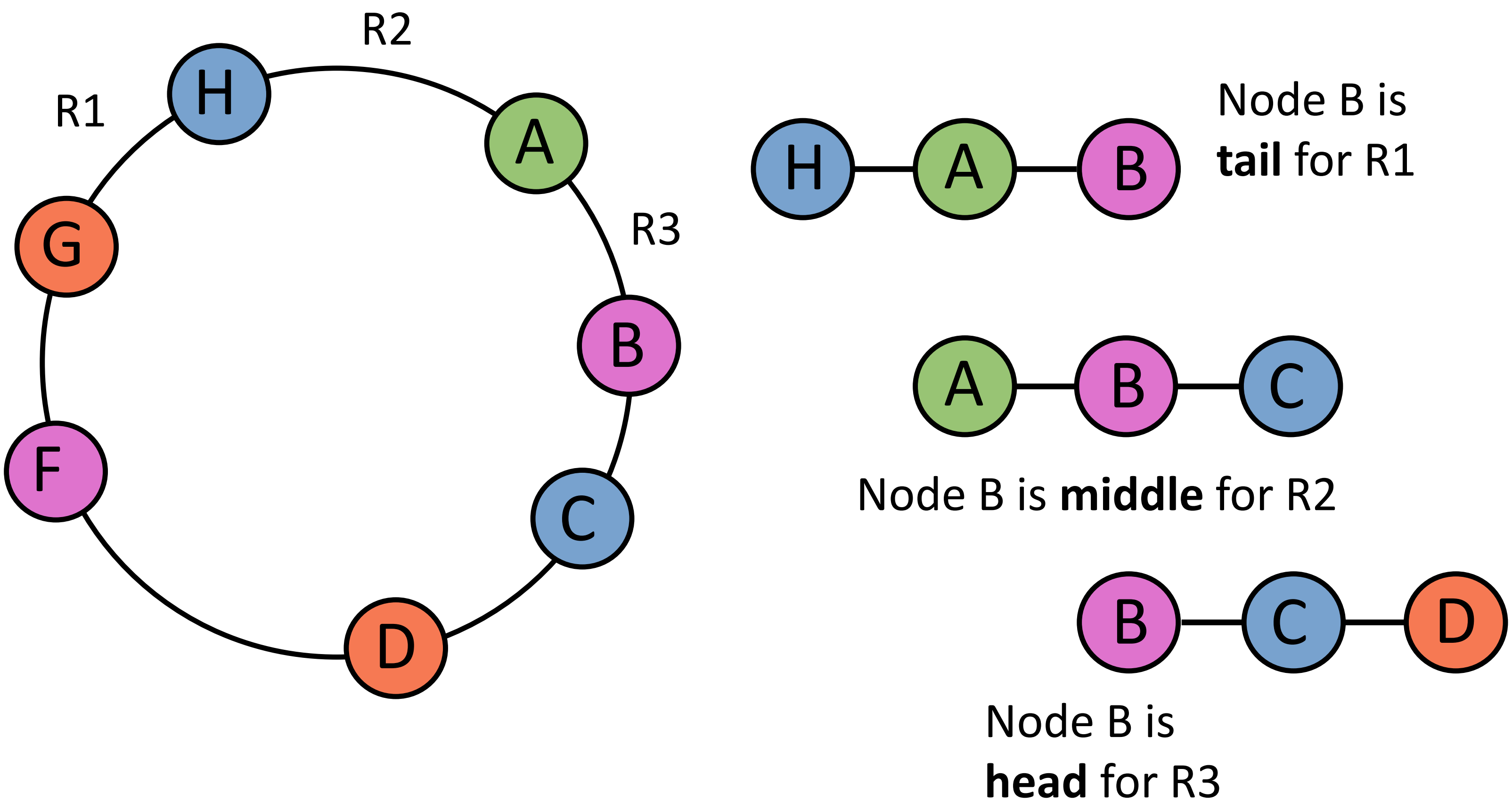


FAWN-KV Chain Replication

- Chain Replication (“primary-backup”)
  - Three copies of data on successive nodes in ring
  - Insert at head, read from tail
  - Strong Consistency: Don’t return to client until all replicas have a copy



Every node is part of three chains

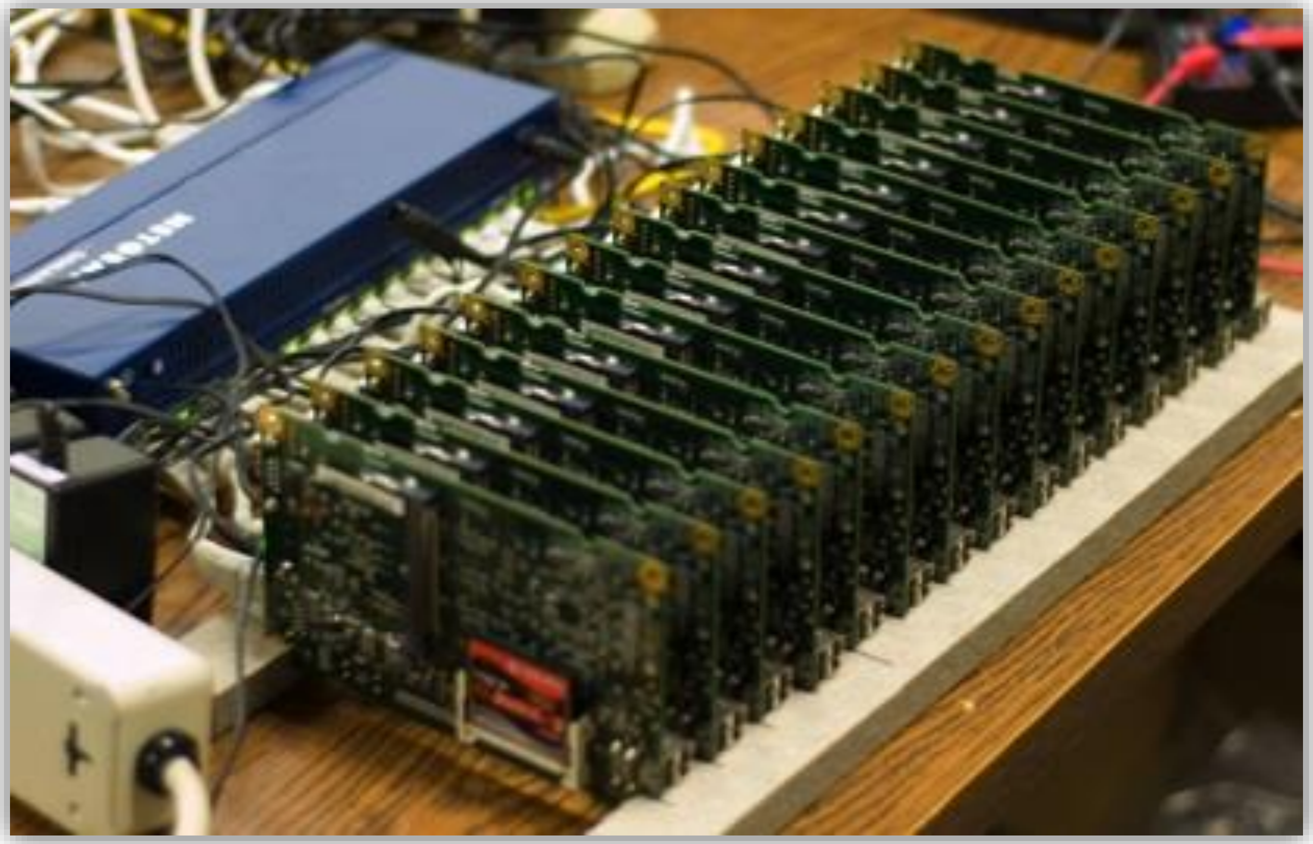


Other design choices – replication

- Quorums
  - Write and read sets must overlap (R + W > N)
  - Ex. Amazon's Dynamo
    - "Sloppy quorums"
    - Things get tricky when there are failures

- Paxos
  - Replicated State Machine
  - Popular recently
  - Relatively complex protocol; lots of corner cases

The “original” FAWN cluster



500 MHz CPU	256 MB DRAM
4 GB CompactFlash	4 W

Metrics

- Power
  - See rest of FAWN paper
- Throughput
- Latency…

Latency

- Can affect user-facing response times—this matters
  - Total round trip to user needs to be 100s of milliseconds
  - Amazon: every 100ms of latency cost them 1% in sales
  - Google: extra .5 seconds in search page generation time dropped traffic by 20%
  - A lot of that is used up by browser-to-data center delay
- Median vs. 99%
  - Effect of fan-out (from Jeff Dean):
Server with 1 ms avg. but 1 sec 99%ile latency
    - touch 1 of these: 1% of requests take ≥1 sec
    - touch 100 of these: <u>63% of requests take ≥1 sec</u>

Future topics

- Network protocols
  - Memcached, thrifty, protobufs, …
  - Batching: multiGET() and multiPUT()
  - RDMA vs. Ethernet: HERD [Kalia, SIGCOMM’14]
- Load Balancing
  - [Fan, SOCC’11], [Li, NSDI’16]
- Geo-replication—KV stores across the wide area
  - See COPS/Eiger [Lloyd, SOSP’11/NSDI’13]
- Building transactional systems on top of KV stores
  - FaRM [Dragojević, SOSP’15], Spanner [Corbett, OSDI’12], FaSST [OSDI’16]