CH6 K,V Cache

Highlights: Key Concepts

- KVCache: fast retrieval of data by key | high availability + low latency
- hash function - important to distribute data uniformly across storage buckets
- scaling concepts
  - CAP
  - partitioning
  - replication
- Consistency
  - eventual vs. strong consistency
  - how kv Cache handle availability and partition tolerance of CAP
- Usage of Cache
  - caching
  - session storage
- Optimize
  - throughput
  - caching strategies
  - Detect / handle conflict
    - Merkel Tree
    - Bloom Filter
  - failure handling
    - heatbeat
    - Gossip

Content

k,v store

- non-relational / NoSQL
- short key works better
- examples: DynamoDB, Redis, Memcached, Cassandra, BigTable
- API
  - get(key)
  - put(key, val)

Init Requirements

- The size of a key-value pair is small: less than 10 KB.
- Ability to store big data.
- High availability: The system responds quickly, even during failures.
- High scalability: The system can be scaled to support large data set.
- Automatic scaling: The addition/deletion of servers should be automatic based on traffic.
- Tuneable consistency.
- Low latency.

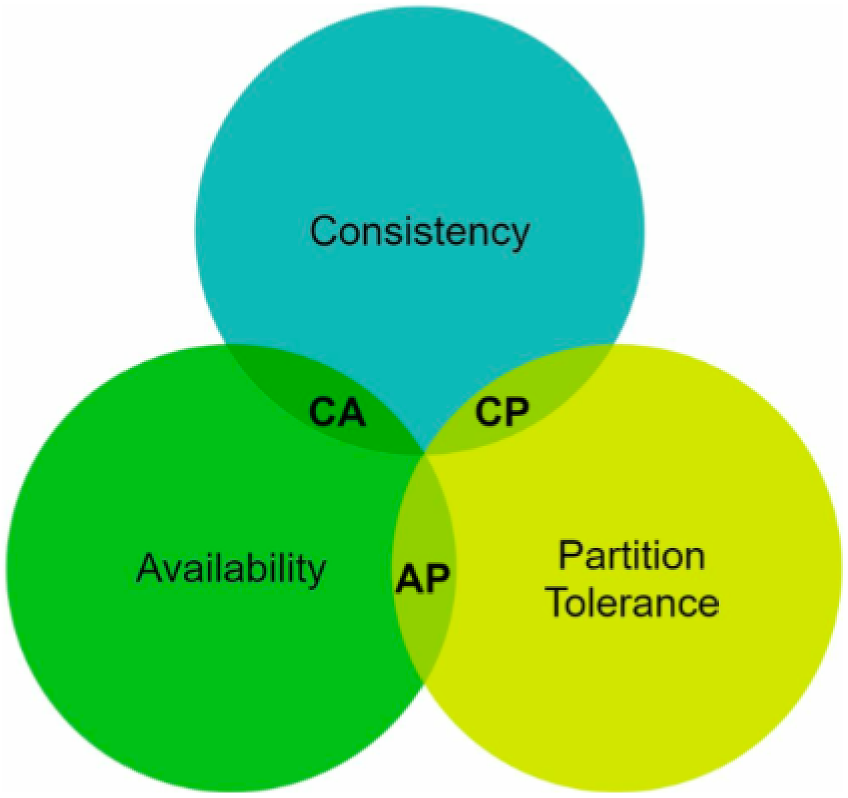
Single Server

- all in memory
  - constraint by size of data
    - Data compression
    - Store only frequently used data in memory and the rest on disk

CAP

Impossible to simultaneously achieve 3 of consistency, availability, partition tolerance.

- consistency - consistency means all clients see the same data at the same time no matter which node they connect to
- availability - availability means any client which requests data gets a response even if some of the nodes are down.
- partition tolerance - Partition tolerance means the system continues to operate despite network partitions (e.g.: communication break between two nodes.)



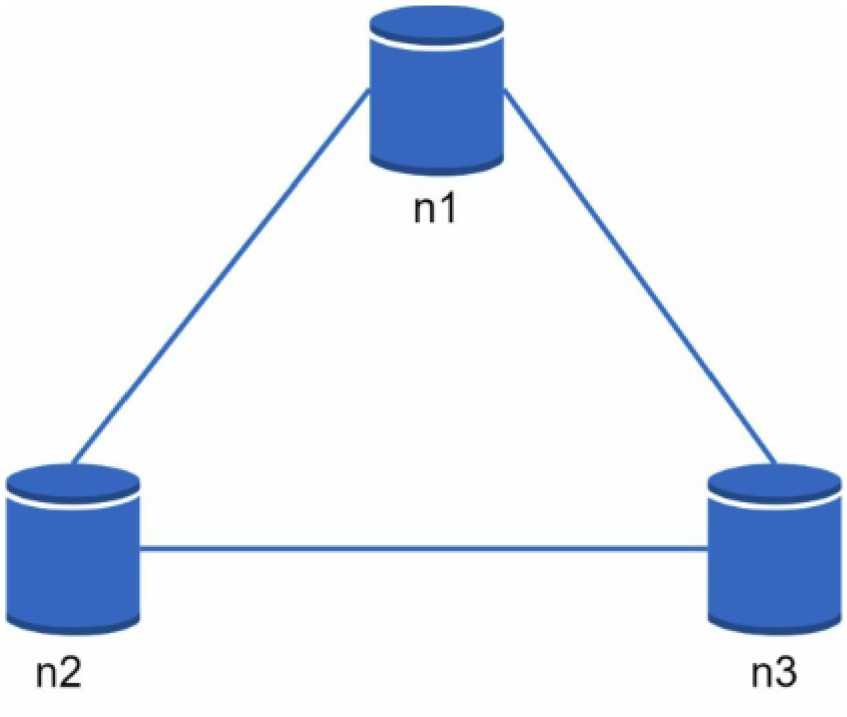
One of them need to be sacrificed to support the other 2

- CP: consistency is more important than availability - e.g.: lock a node before its data get consistent with others during an update
- AP: availability is more important than consistency - e.g.: client can still get deprecating data when updating
- CA - distributed system can not go without partition tolerance (network failure is unavoidable)

Real-world case

- when a partition occurs, we must choose between consistency and availability.
- Example
  - If clients write data to n1 or n2, data cannot be propagated to n3. If data is written to n3

but not propagated to n1 and n2 yet, n1 and n2 would have stale data.



Distributed kv store

- Data partition
  - Requirement
    - Distribute data across multiple servers evenly
    - Minimize data movement when nodes are added or removed
  - Consist hash
    - pros
      - auto-scaling - via virtual nodes
      - Heterogeneity - servers with higher capacity are assigned with more virtual nodes
- Data replication
  - Data async copy to N servers (N - number of replica) - walk **clockwise** from that position and choose the first N servers on the ring to store data copies
    - make sure N nodes have data - only choose unique servers (N=3 here)
    - use multiple data centers as replica (with high speed network) to avoid data center downtime

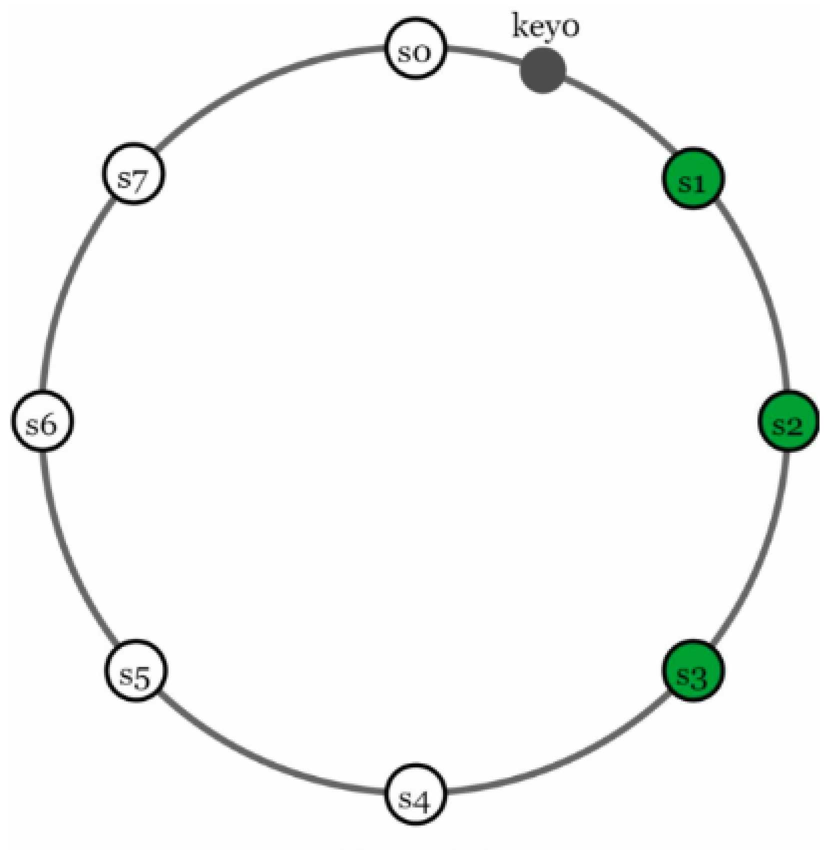
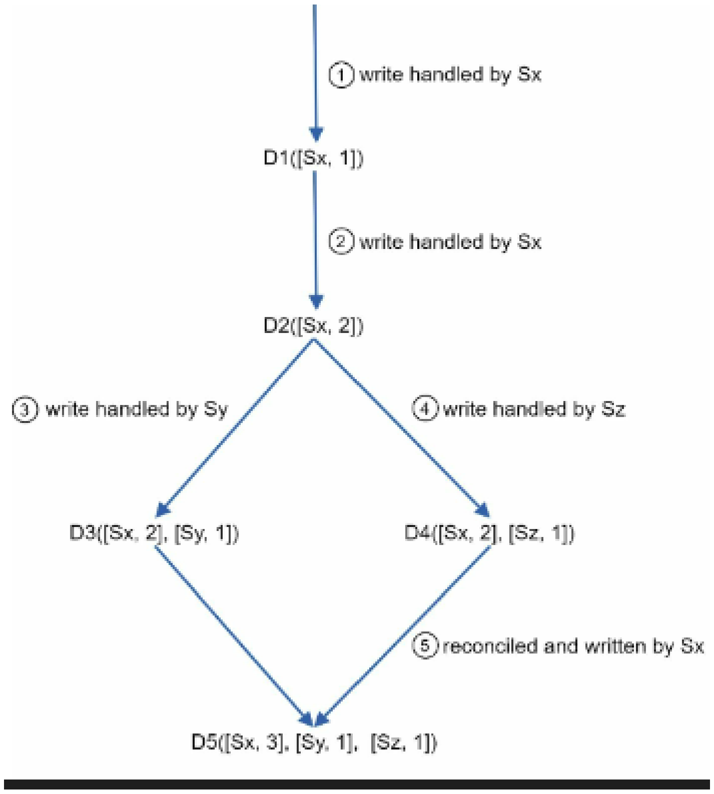


Figure 6-5

- Consistency
    - quorum consensus
        - N - number of replica
        - W - write quorum size, must be recognized by at least W ack
            - e.g.: W=1, if one write get ack, no need to wait for others
        - R - read quorum size, must receive responses of R replicas
    - consistency vs. latency tradeoff
        - If W, R = 1, fast response
        - If W, R > 1, better consistency
        - W + R > N, strong consistency - at least one overlapping node with latest data
        - e.g.:
            - If R = 1 and W = N, the system is optimized for a fast read.
            - If W = 1 and R = N, the system is optimized for fast write.
            - If W + R > N, strong consistency is guaranteed (Usually N = 3, W = R = 2).
            - If W + R <= N, strong consistency is not guaranteed.
    - consistency models
        - strong consistency - A client never sees out-of-date data
            - achieved by forcing replica not to accept read/write until all replicas agrees on current write
        - weak consistency

        - eventual consistency - Given enough time, all updates are propagated, and all replicas are consistent.
            - allow inconsistent values in systems and force client to read values to reconcile
    - Inconsistency resolution
        - versioning: each data modification is a new immutable version
        - vector clock - (server, version) pair
            - if new data coming
                - increment if `[Si, vi]` exists, else create `[Si, 1]`
            - reconcile

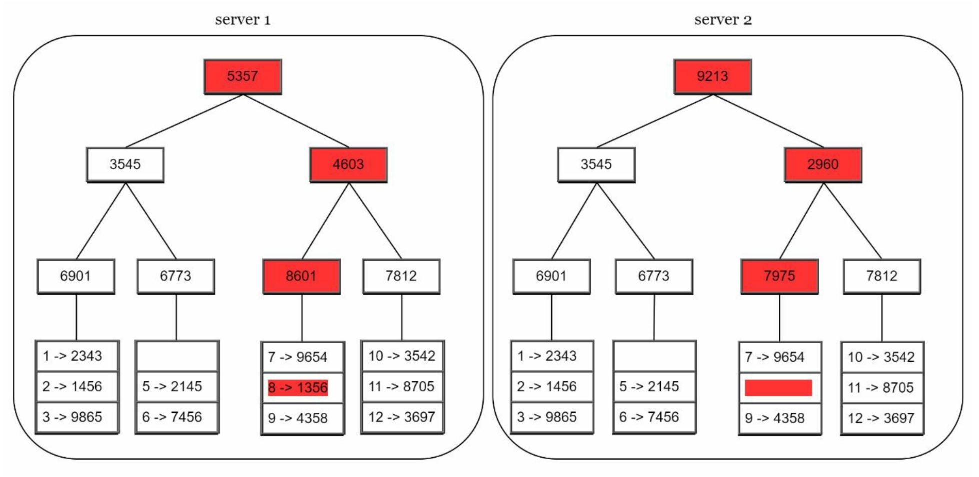


    - reconcile: "The conflict is resolved by the client and updated data is sent to the server"
        - why still keep Sy, Sz?

- Handling failures

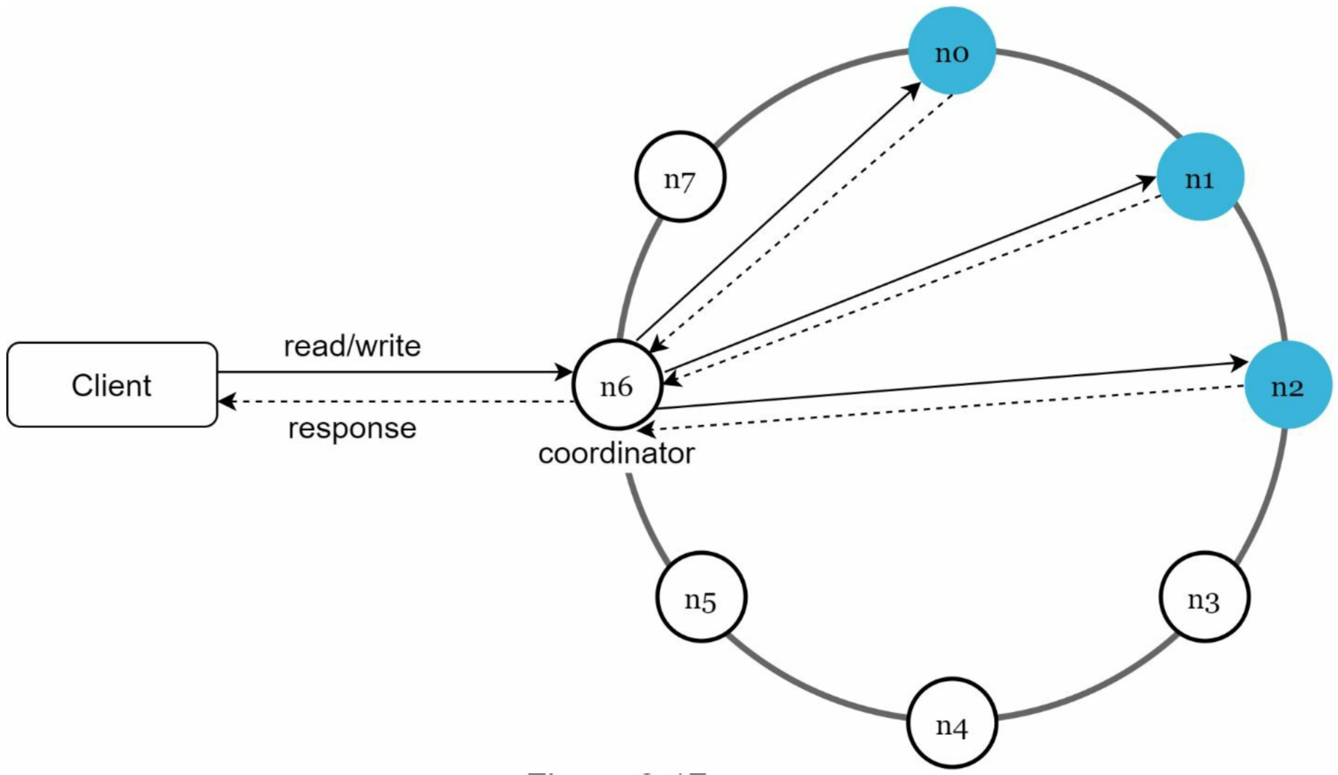
    - Failure detection - >1 source marked a server as down
        - multicasting detection - all servers vote
        - decentralized detection - Gossip Protocol
            - each node maintains a membership list <member_id, heatbeat_cnt>, send heatbeat
            - For node X, and Y in membership of X, if Y's heatbeat stopped for a long time, X ask random nodes in the X's list
            - if other nodes agree, then Y is marked as down

- Handling Failures
    - temporary failures
        - sloppy quorum - instead of enforcing quorum, select first W and first R healthy servers
        - hinted handoff: when the failed server gets back, write new data to this server
    - Permanent failure
        - Anti-entropy protocol using Merkel Tree / Hash Tree
            - each node has hash of keys / values of all its children
            - Steps
                - data (k, v) -> bucket
                - each bucket -> hash (uniform hash)
                - each node -> hash all its children



Data Center Outage - replica

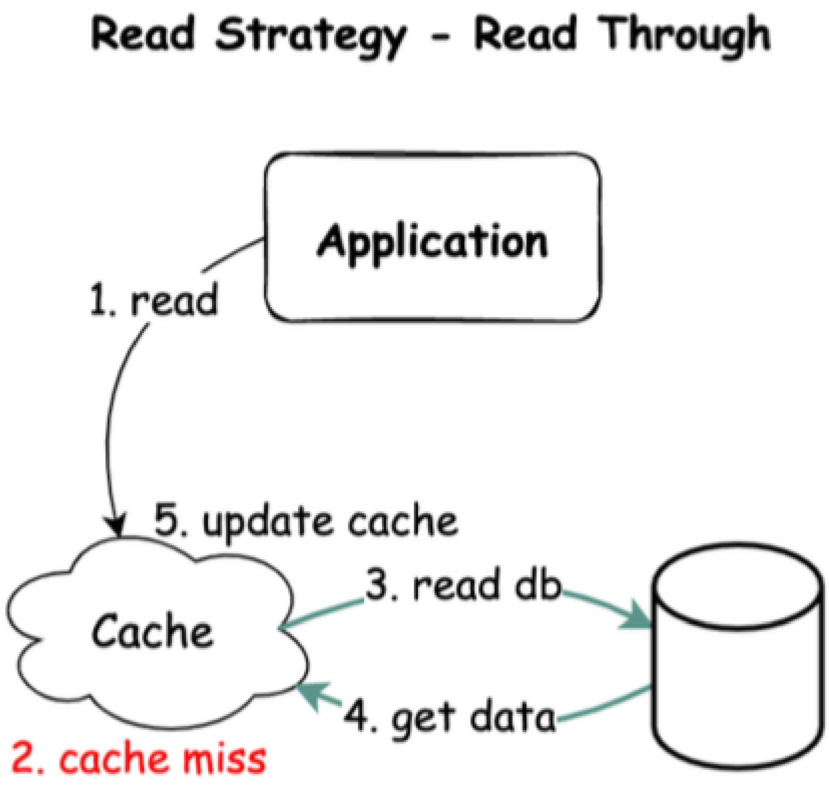
System architecture diagram



- A coordinator is a node that acts as a proxy between the client and the key-value store
- no single point of failure as every node has the same set of responsibilities

Caching Strategies

1. Read Through Cache



Brief Introduction:
In a read-through cache, the cache acts as a front to the database. When a read request is made, the cache first checks if the data is available. If it's not, the cache loads the data from the database, stores it in the cache, and then returns it to the user.

Pros:

- **Data Consistency:** Ensures data consistency between the cache and the database.
  - use write-through strategy - high consistency for writing
  - use invalidating cache - data updates are less frequent compared to reads
  - use async update - eventually consistent
- **Simplicity:** Easy to implement and manage as data is loaded into the cache only when necessary.

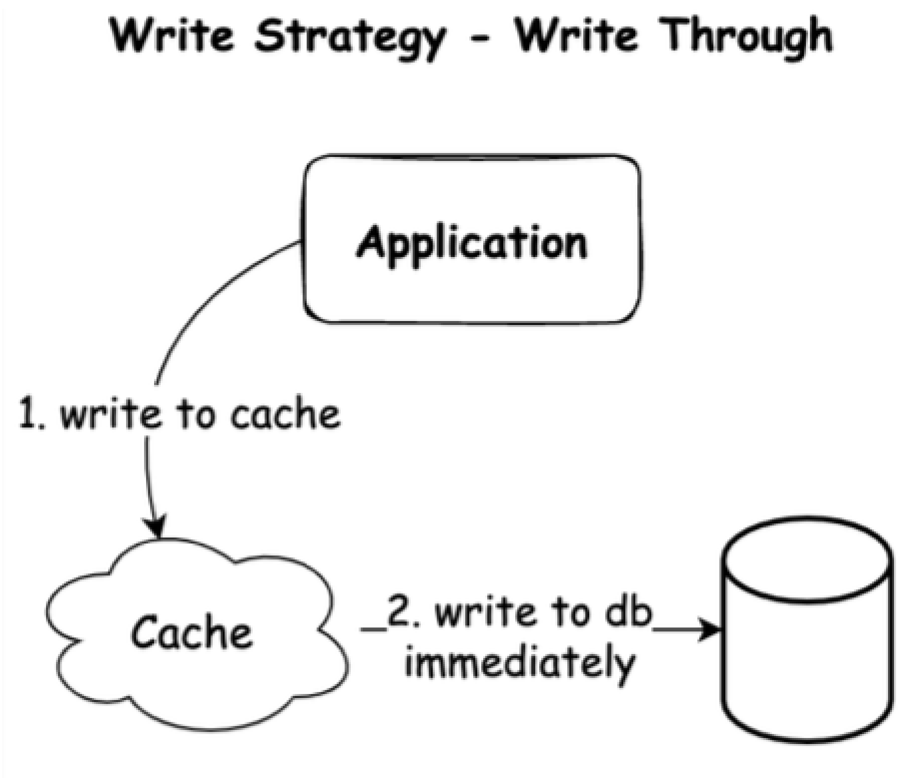
Cons:

- **Latency:** Initial read request suffers from higher latency as data needs to be loaded into the cache first.
- **Resource Utilization:** Can increase load on the database during cache misses.

Typical Use Cases:

- Applications that require high consistency and where data is frequently read but not often updated, such as user profile information in web applications.
- Read aside - application is responsible to update the database if cache miss

2. Write Through Cache



Brief Introduction:
This caching strategy ensures that writes are made simultaneously to the cache and the database. This method guarantees that the cache always contains the most up-to-date data.

Pros:

- **Data Consistency:** Maintains strong data consistency between the cache and the database.
- **Data Safety:** Reduces the risk of data loss in case of a failure after the data is written to the cache.

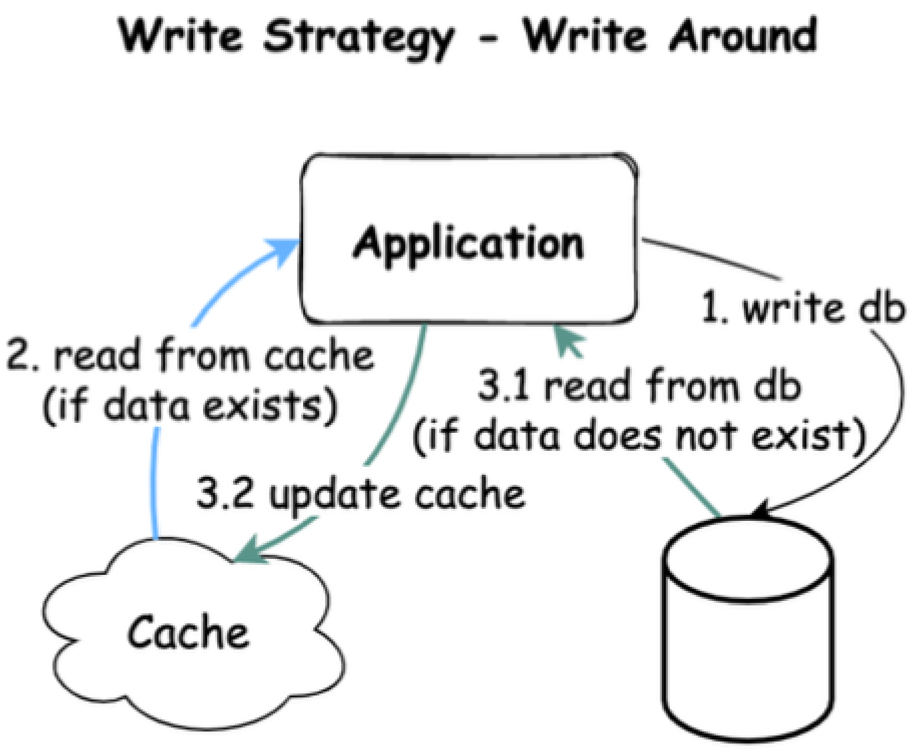
Cons:

- **Latency:** Write operations can be slower due to the need to write data to two locations.
- **Complexity:** More complex to implement, especially in distributed environments.

Typical Use Cases:

- Critical applications where data consistency and safety are paramount, such as banking and financial transaction systems.

3. Write Around Cache


Write Strategy - Write Around
Application
1. write db
2. read from cache (if data exists)
3.1 read from db (if data does not exist)
3.2 update cache
Cache

Brief Introduction:
In a write-around cache, data is written directly to the backend storage without being written to the cache. The cache only stores data that is read frequently.

Pros:

- **Minimizes Cache Pollution:** Prevents cache from being filled with data that might not be read again.
- **Improved Performance for Read Intensive Data:** Frequent read data benefits from cache without being cluttered by write data.

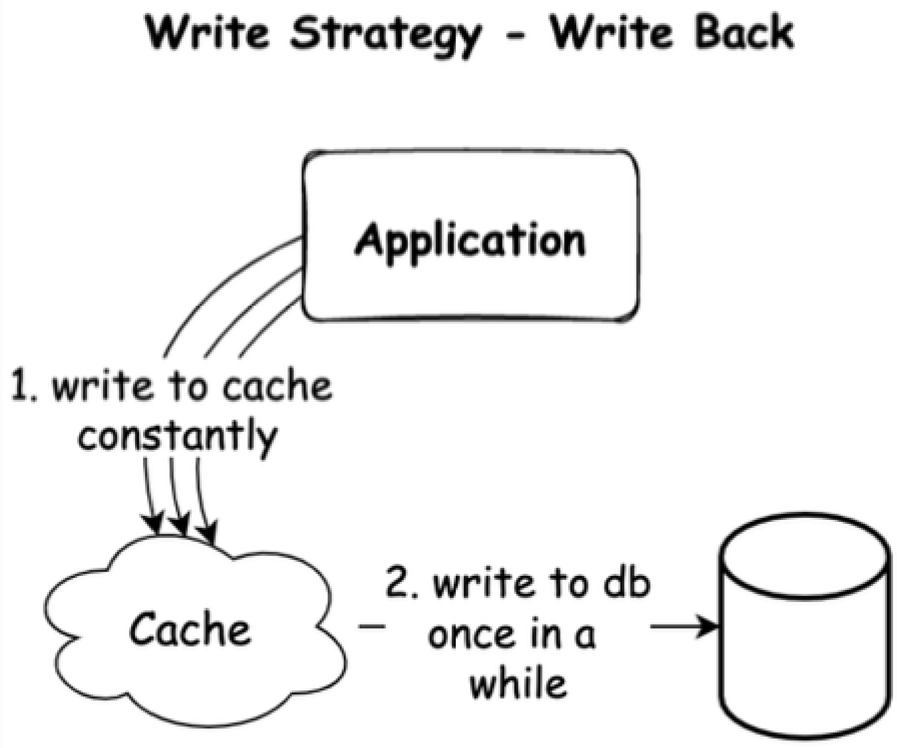
Cons:

- **Higher Read Latency for New Data:** Data that is newly written and then read soon after will cause a cache miss and thus higher latency.

Typical Use Cases:

- Applications where writes are heavy and reads for the same data are less frequent, such as logging or archival systems.

4. Write Back Cache


Write Strategy - Write Back
Application
1. write to cache constantly
Cache
2. write to db once in a while

Brief Introduction:
With write-back (or write-behind) caching, data is initially written only to the cache and not to the backend database. The write to the database is deferred as a batch later.

Pros:

- **High Write Performance:** Improves performance as writes are not immediately committed to the database.
- **Reduced Database Load:** Decreases the load on the database server.

Cons:

- **Data Safety Risks:** There's a risk of data loss if the cache fails before the data is written back to the database.
- **Complexity in Data Consistency:** Managing data consistency between cache and database can be challenging.

Typical Use Cases:

- High-performance applications where write speed is critical and temporary data loss can be tolerated, such as session storage or real-time data processing.

Bloom Filter

- a space-efficient probabilistic data structure

- used to test whether an element is a member of a set
- might have false positives - an element might not actually exist in the set

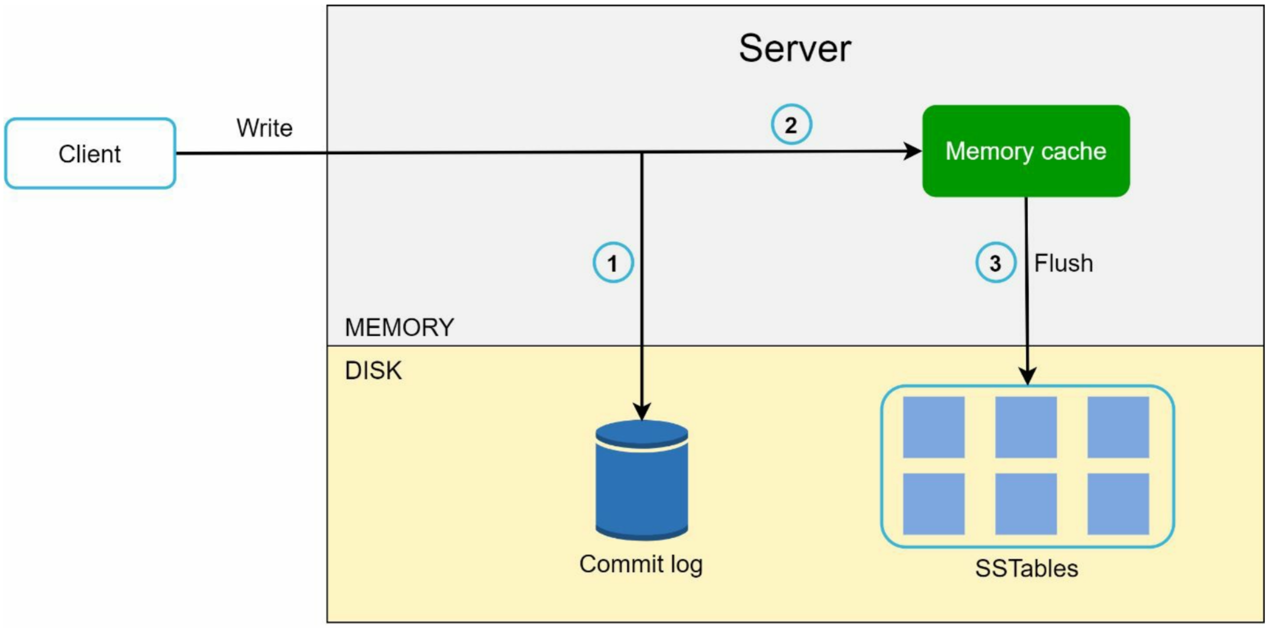
Implementation

- array of bits, initiallly all 0s
- check membership
    - use hash functions to hash a key to get different position
    - if all are set to 1, the item is in this set
    - if any 0, definitely not in this set
- Code

```python
class SimpleBloomFilter:
    def __init__(self, size, num_hashes):
        self.size = size
        self.num_hashes = num_hashes
        self.bit_array = [0] * size

    def _get_hashes(self, item):
        hashes = []
        # Convert item to string and encode to bytes
        item = str(item).encode('utf-8')

        for i in range(self.num_hashes):
            # Create a new hash for each function using a salt (i)
            hash_object = hashlib.md5(item + str(i).encode('utf-8'))
            hash_digest = hash_object.hexdigest()
            # Convert the hex digest to a number and mod it by the size of
the bit array
            hash_number = int(hash_digest, 16) % self.size
            hashes.append(hash_number)

        return hashes

    def add(self, item):
        # Get all hash values and set the corresponding indices in the bit
array to 1
        for hash_value in self._get_hashes(item):
            self.bit_array[hash_value] = 1

    def check(self, item):
        # Check if all bits are set for the given item
        for hash_value in self._get_hashes(item):
            if self.bit_array[hash_value] == 0:
                return False
        return True
```

Single Node Operations

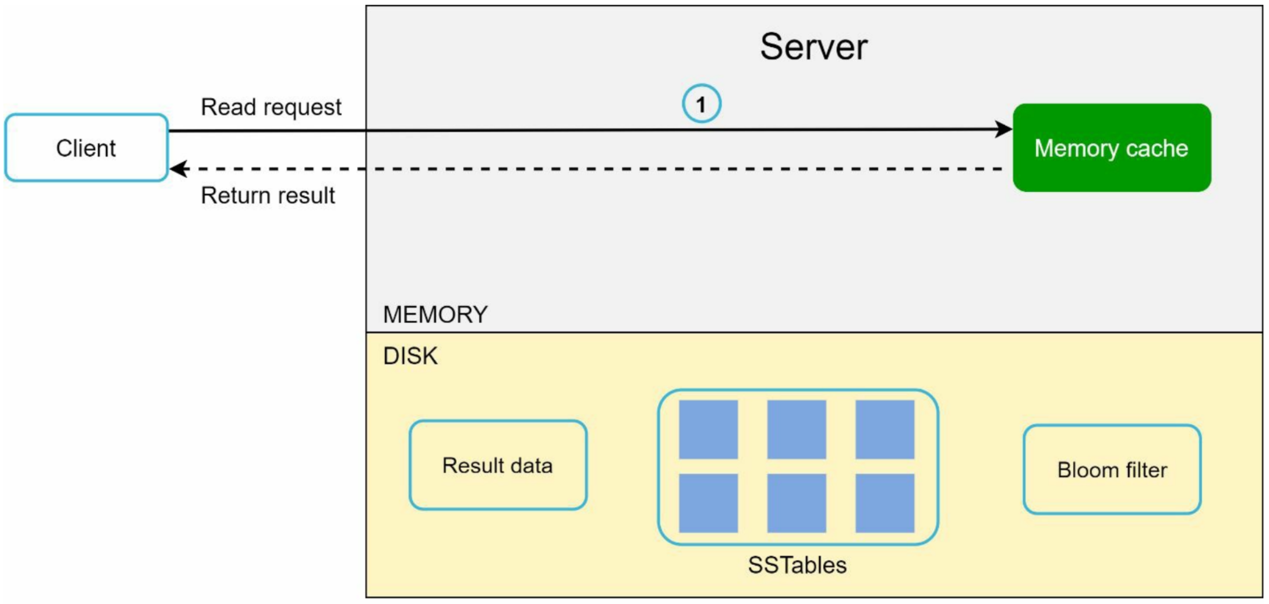
- Write path



- Write-back:
- write to cache first
- when cache is full, flush to SSTable

SSTable: sorted list

- Read path



- Read through:

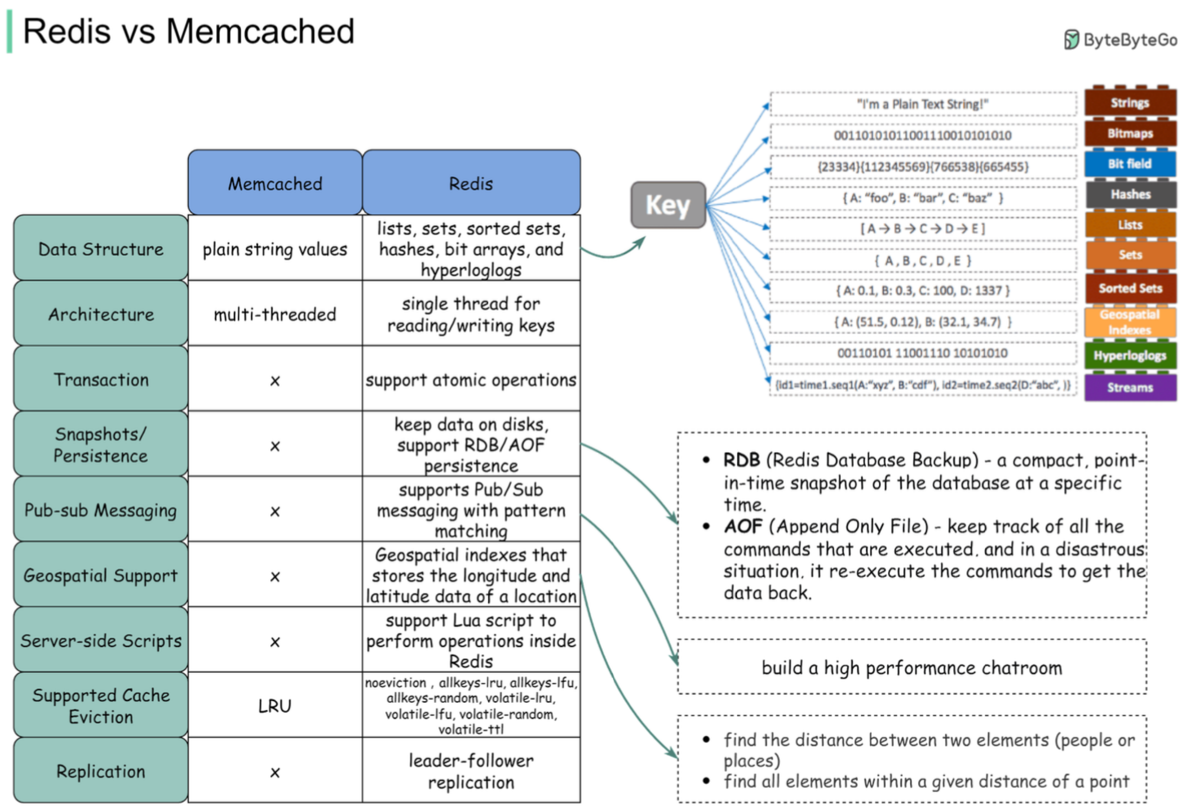
- read from cache first
- if miss, then use bloom filter to detect which SSTable has data, and read it
- update cache

Write Back + Read Though => Consistent + high write speed + read: data not often updated for single node

- level of consistency of whole system is determined by W + R, not here

Questions

- Redis vs. Memcached



Redis vs Memcached

	Memcached	Redis
Data Structure	plain string values	lists, sets, sorted sets, hashes, bit arrays, and hyperloglogs
Architecture	multi-threaded	single thread for reading/writing keys
Transaction	×	support atomic operations
Snapshots/ Persistence	×	keep data on disks, support RDB/AOF persistence
Pub-sub Messaging	×	supports Pub/Sub messaging with pattern matching
Geospatial Support	×	Geospatial indexes that stores the longitude and latitude data of a location
Server-side Scripts	×	support Lua script to perform operations inside Redis
Supported Cache Eviction	LRU	noeviction , allkeys-lru, allkeys-lfu, allkeys-random, volatile-lru, volatile-lfu, volatile-random, volatile-ttl
Replication	×	leader-follower replication

- **RDB** (Redis Database Backup) - a compact, point-in-time snapshot of the database at a specific time.
- ***AOF*** (Append Only File) - keep track of all the commands that are executed, and in a disastrous situation, it re-execute the commands to get the data back.

build a high performance chatroom

- find the distance between two elements (people or places)
- find all elements within a given distance of a point

- Memcached
- multi-threaded
- simple k,v
- high memory usage
- cons
- no persistency
- Redis
- multiple data structure
- master-slave
- persitency
- cons

- single threaded
- why single threaded can be faster?

- Redis Sentinel
  - tools to monitor status of redis master (if down, promote slave)
- How Lua Script handling consistency with high throughput