AWS
ElastiCache

Terms we should know before we learn about ElastiCache

In-Memory Data Store: A type of database management system that relies primarily on memory for data storage to achieve faster data retrieval times compared to disk-based storage.

Cache: A high-speed data storage layer that stores a subset of data, typically transient in nature, to serve future requests faster.

Key-Value Store: A type of NoSQL database where data is stored as a collection of key-value pairs. Both Redis and Memcached are examples of key-value stores.

Eviction Policy: The strategy used to decide which data to remove from the cache when it reaches its memory limit. Common policies include Least Recently Used (LRU) and First In, First Out (FIFO).

TTL (Time to Live): The duration for which a data item is stored in the cache before it is automatically invalidated.

Replication: The process of copying data from one database server (master) to another (slave) to ensure data redundancy and improve read performance.

Sharding: A method of partitioning data across multiple servers or nodes to distribute the load and enable horizontal scaling.

Cluster: A group of interconnected servers (or nodes) that work together to manage and distribute the cache data, providing scalability and fault tolerance.

Node: An individual server within a cache cluster that stores a portion of the data.

Endpoint: The connection string used by an application to connect to a specific node or cluster in ElastiCache.

Client-Side Caching: A caching strategy where the cache is maintained on the client side (e.g., within the application's memory).

Server-Side Caching: A caching strategy where the cache is maintained on the server side (e.g., using ElastiCache).

Persistence: The ability of a caching system to save its state to disk, ensuring data is not lost in case of a restart or failure (particularly relevant to Redis).

Pub/Sub (Publish/Subscribe): A messaging pattern where messages are sent by a publisher and received by subscribers, supported natively by Redis.

Lua Scripting: A lightweight, embeddable scripting language supported by Redis for executing complex operations atomically.

AWS
ElastiCache

ElastiCache in AWS is used to enhance the performance and scalability of applications that rely heavily on data retrieval.

ElastiCache: This is the managed service provided by AWS that allows you to deploy, operate, and scale an in-memory cache in the cloud. It supports two engines:

- **Memcached**
- **Redis**

- AWS managed caching service
- In-memory key-value store with (sub-millisecond latency)
- Need to provision EC2 instances (nodes for the cluster)
- Makes the application stateless because it doesn't have to cache locally
- Using ElastiCache requires heavy application code changes

Usage:

- DB Cache (lazy loading): cache read operations on a database (reduced latency)
- Session Store: store user's session data like cart info (allows the application to remain stateless)
- Global Data Store: store intermediate computation results

Two Engines → How they differ with each other

Memcached: A simple, distributed memory object caching system. It is designed for simplicity and speed, making it a great choice for caching objects and data that do not require advanced data structures or persistence. Memcached is often used for scenarios where high throughput and low latency are required.

Redis: An open-source in-memory data structure store that supports more advanced data types like strings, hashes, lists, sets, and sorted sets. Redis offers features like data persistence, replication, and high availability, making it suitable for more complex use cases beyond simple caching, such as real-time analytics, messaging, and geospatial data storage.

AWS
ElastiCache

Main Ones are:

Redis	Memcached
In-memory data store	**Distributed** memory object cache
Read Replicas (for scaling reads & HA)	No replication
Backup & restore	No backup & restore
Single-threaded	**Multi-threaded**
HIPAA compliant	**Not HIPAA compliant**

Feature	Memcached	Redis
Data Structures	Key-value store	Supports various data structures (strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs, streams)
Persistence	No persistence	Supports disk persistence (RDB snapshots, AOF logs)
Replication	No native replication	Supports master-slave replication
High Availability	No built-in high availability	Supports Redis Sentinel for high availability
Scalability	Horizontal scaling by adding/removing nodes	Horizontal scaling with Redis Cluster
Data Expiry	Supports expiration of individual keys	Supports expiration of individual keys
Transactions	No support for transactions	Supports transactions (MULTI/EXEC commands)
Pub/Sub Messaging	Not supported	Supports publish/subscribe messaging
Scripting	No support for scripting	Supports Lua scripting
Memory Management	Slab allocation	Configurable eviction policies, memory usage can be fine-tuned
Performance	Extremely fast with low latency	Very fast with additional features
Use Cases	Simple caching	Caching, real-time analytics, messaging, geospatial data, and more
Complexity	Simple and easy to use	More complex but offers richer functionality
Atomic Operations	Basic atomic operations	Rich set of atomic operations on complex data types

AWS
ElastiCache

Security & Access Management

- Network security is managed using **Security Groups**
- At rest encryption using KMS
- In-flight encryption using SSL
- Use **Redis Auth** to authenticate to ElastiCache for Redis
- Memcached supports SASL-based authentication

During Redis and Memcached differences comparison → we saw about replication, Redis supports built in Replication, we need to understand how many modes it supports that,

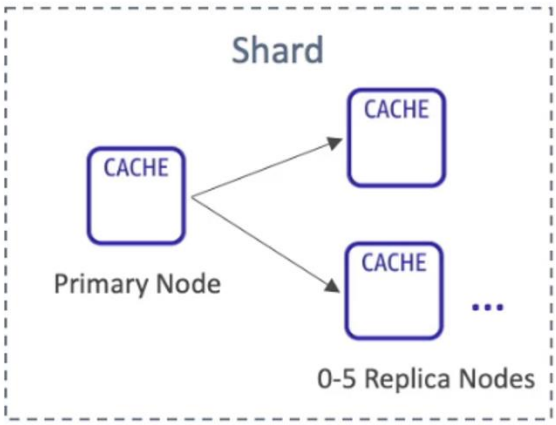
It supports in 2 modes

- One is Cluster Mode Disabled
- Second is Cluster Mode Enabled

AWS
ElastiCache

Cluster Mode Disabled

- **1 primary node and up to 5 read-replicas (asynchronous replication).** Writes can occur only at the primary node. Reads can occur at any node including the primary node.
- **No sharding** (1 shard only) - every node has complete data
- **Helpful for scaling read performance**
- **Automated failover** - if the primary node fails, one of the read-replicas will take over as the new master
- Supports multi-AZ for failover in case an entire AZ is down

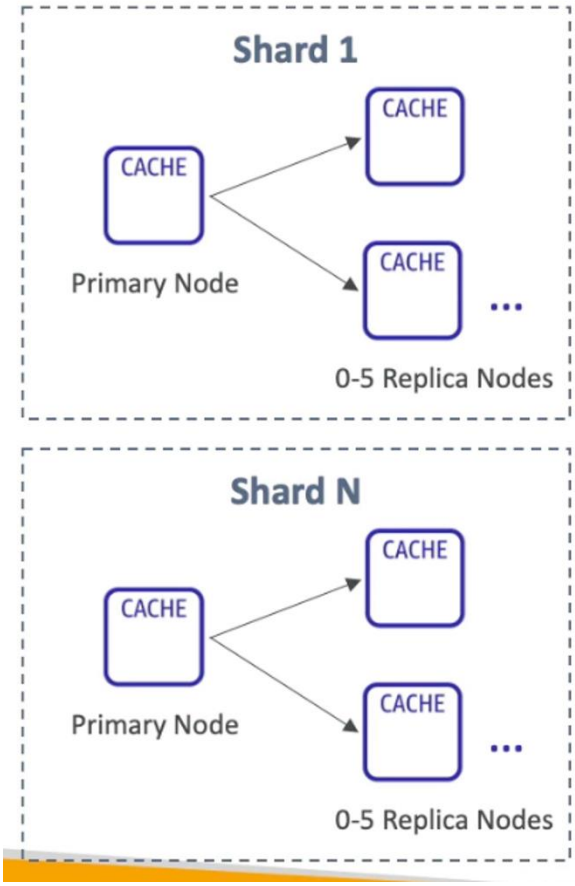


Cluster Mode Enabled

- **Sharding** - every node has partial data
- **Each shard has 1 primary and up to 5 read-replicas (asynchronous replication)**
- Sharding is helpful for scaling writes
- Read-replicas are helpful for scaling reads
- **Requires multi-AZ** for failover in case an entire AZ is down

AWS
ElastiCache

- Up to **500 nodes per cluster**
    - 500 shards without replication (no read-replica)
    - 250 shards with 1 master and 1 replica
- You cannot manually promote any of the replica nodes to primary.
- You can only change the structure of a cluster, the node type, and the number of nodes by restoring from a backup.



AWS
ElastiCache

Now – what strategies we need to follow while we designing an application in order to have a cache mechanism in place

There are 2 routes of strategies

- Lazy Loading / Cache-Aside / Lazy Population
- Write-Through

Lazy Loading / Cache-Aside / Lazy Population

This strategy loads data into the cache only when it is requested. If the data is not present in the cache (a cache miss), the application fetches it from the underlying data store, returns it to the requester, and also stores it in the cache for future requests.

Workflow:

1. Application requests data.
2. Cache is checked for the data.
3. If data is found in the cache (cache hit), it is returned to the application.
4. If data is not found in the cache (cache miss), the application fetches it from the data store.
5. The fetched data is stored in the cache for future requests.
6. The data is returned to the application.

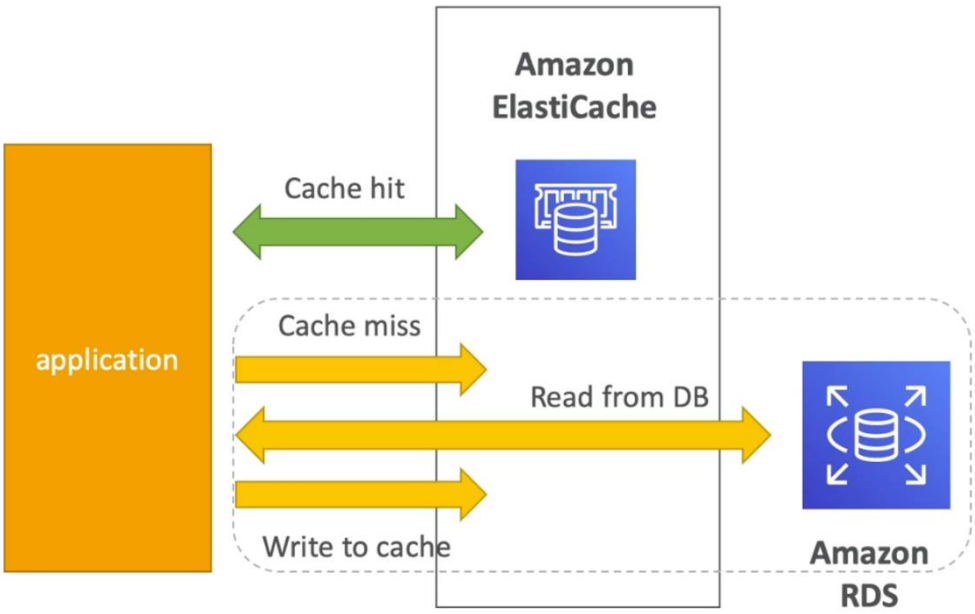
Pros

- Only the data requested from the DB is cached. The cache will not contain unused data.
- Cache failures are not fatal, they just lead to increased latency of fetching the data from the DB.

Cons

- **Cached data can become stale** (outdated)
- Cache miss results in 3 network RTT delay which affects user experience

AWS
ElastiCache



The application reads the data from the cache first. If it is present in the cache (cache hit), the application gets it immediately. If it's not (cache miss), the application reads it from the DB, and writes it to the cache for later reads. This is used to optimize reads on the cache and is the first thing you should implement as a part of caching. This is good when we have a **small cache size and we only want to cache the data that is actively fetched** from the database.

AWS
ElastiCache

Write Through

In this strategy, data is written to the cache and the underlying data store simultaneously. Every write operation updates both the cache and the database.

Workflow:

1. Application writes data.
2. Data is written to the cache.
3. Data is also written to the underlying data store.

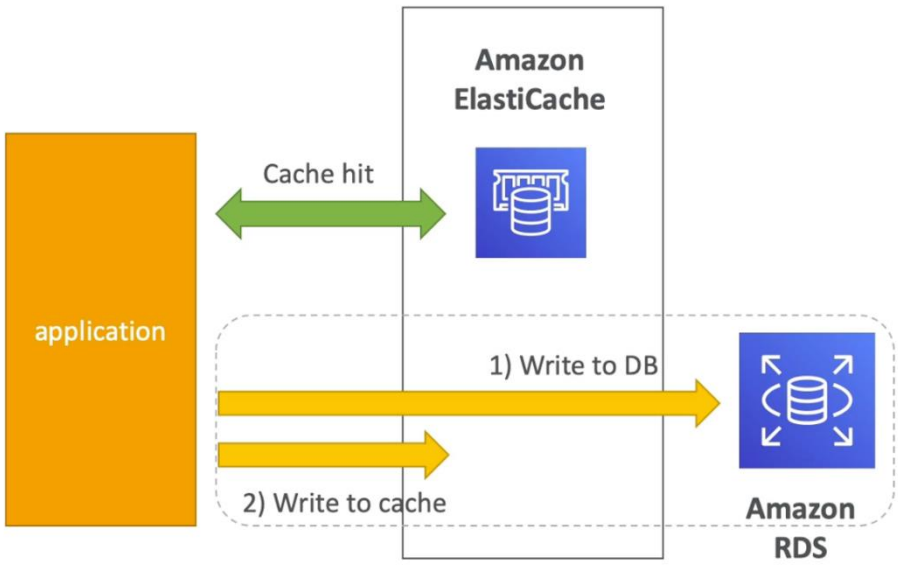
Pros

- **Cache always contains up to date data** (no stale data)
- Each write takes up 2 network RTT delay which is better than 3 RTT delay during cache miss reads in case of lazy loading.

Cons

- Data that doesn't change in the DB doesn't get cached. Solution is to implement lazy loading alongside write through.
- **Cache churn** - most of the cached data will never be read

AWS
ElastiCache



Whenever there's a change to some data in the DB (write), write the data to cache. **This is done on top of lazy loading to optimize cache writes.** This enables **faster reads at the cost of longer writes.**

The cache and the DB cannot be updated at the same time via a single atomic operation as these are two separate systems. The cache must be updated or invalidated after writing to the DB.

AWS
ElastiCache

Lazy loading can be combined with Write through to optimize cache reads and writes.

Comparison of Lazy Loading and Write-Through

Feature	Lazy Loading (Cache-Aside)	Write-Through
Read Latency	Higher on first request (cache miss)	Lower, as the cache is always populated
Write Latency	Lower, as writes do not affect the cache	Higher, as writes update both cache and DB
Cache Staleness	Data can become stale if not updated	Cache always reflects the most recent data
Implementation	Simple to implement, often used for read-heavy apps	More complex, suitable for write-heavy apps

Cache Eviction

The cache has a limited size and therefore the old or unused data must be removed to make room for new data. This is called as cache eviction. It occurs in 3 ways:

- Delete the item explicitly
- Delete the least recently used (LRU) data
- Use TTL for the data stored in cache

If too many cache evictions occur, you should consider scaling up/out your cache.

AWS
ElastiCache

ElastiCache Scenario based Questions

https://lisireddy.medium.com/aws-elasticache-scenario-based-questions-e63682e85ac3

Wish you the best …! Happy Learning ..!

Yours' Love (@lisireddy across all the platforms)