AWS Whitepaper

Database Caching Strategies Using Redis

Database Caching Strategies Using Redis: AWS Whitepaper

Table of Contents

This whitepaper is for historical reference only. Some content might be outdated and some links might not be available.

Database Caching Strategies Using Redis

Publication date: **March 8, 2021** (*Document Revisions*)

Abstract

In-memory data caching can be one of the most effective strategies for improving your overall application performance and reducing your database costs.

You can apply caching to any type of database, including relational databases (such as Amazon Relational Database Service (Amazon RDS)) or NoSQL databases (such as Amazon DynamoDB, Amazon DocumentDB (with MongoDB compatibility), and Amazon Keyspaces (for Apache Cassandra)).

One of the benefits of caching is that it's an easier option to implement, and it dramatically improves the speed and scalability of your application. Caching can also apply to objects (for instance, objects stored in Amazon Simple Storage Service), as this paper will explore.

This whitepaper describes some of the caching strategies and implementation approaches that address the limitations and challenges associated with disk-based databases.

Database challenges

When you're building distributed applications that require low latency and scalability, disk-based databases can pose a number of challenges. A few common ones include the following:

- **Slow processing queries**: There are a number of query optimization techniques and schema designs that help boost query performance. However, the data retrieval speed from disk plus the added query processing times generally put your query response times in double-digit millisecond speeds, at best. This assumes that you have a steady load and your database is performing optimally.
- **Cost to scale**: Whether the data is distributed in a disk-based NoSQL database or a vertically-scaled relational database, scaling for extremely high reads can be costly. It also can require several database read replicas to match what a single in-memory cache node can deliver in terms of requests per second.
- **The need to simplify data access**: Although relational databases provide an excellent means to data model relationships, they aren't optimal for data access. There are instances where your applications may want to access the data in a particular structure or view, to simplify data retrieval and increase application performance.

Before implementing database caching, many architects and engineers spend great effort trying to extract as much performance as they can from their databases. However, there is a limit to the performance that you can achieve with a disk-based database, and it's counterproductive to try to solve a problem with the wrong tools. For example, a large portion of the latency of your database query is dictated by the physics of retrieving data from disk.

Types of database caching

A database cache supplements your primary database by removing unnecessary pressure on it, typically in the form of frequently-accessed read data. The cache itself can live in several areas, including in your database, in the application, or as a standalone layer.

The following are the three most common types of database caches:

Database-integrated caches

Some databases, such as Amazon Aurora, offer an integrated cache that is managed within the database engine and has built-in write-through capabilities. The database updates its cache automatically when the underlying data changes. Nothing in the application tier is required to use this cache.

The downside of integrated caches is their size and capabilities. Integrated caches are typically limited to the available memory that is allocated to the cache by the database instance and can’t be used for other purposes, such as sharing data with other instances.

Local caches

A local cache stores your frequently-used data within your application. This makes data retrieval faster than with other caching architectures because it removes network traffic that is associated with retrieving data.

A major disadvantage is that among your applications, each node has its own resident cache working in a disconnected manner. The information that is stored in an individual cache node (whether it’s cached database rows, web content, or session data) can’t be shared with other local caches. This creates challenges in a distributed environment where information sharing is critical to support scalable dynamic environments.

Because most applications use multiple application servers, coordinating the values across them becomes a major challenge if each server has its own cache. In addition, when outages occur, the data in the local cache is lost and must be rehydrated, which effectively negates the cache. The majority of these disadvantages are mitigated with remote caches.

Remote caches

A remote cache (or *side cache*) is a separate instance (or separate instances) dedicated for storing the cached data in-memory. Remote caches are stored on dedicated servers and are typically built on key/value NoSQL stores, such as Redis and Memcached. They provide hundreds of thousands of requests (and up to a million) per second per cache node. Many solutions, such as Amazon ElastiCache (Redis OSS), also provide the high availability needed for critical workloads.

The average latency of a request to a remote cache is on the sub-millisecond timescale, which, in the order of magnitude, is faster than a request to a disk-based database. At these speeds, local caches are seldom necessary. Remote caches are ideal for distributed environments because they work as a connected cluster that all your disparate systems can utilize. However, when network latency is a concern, you can apply a two-tier caching strategy that uses a local and remote cache together. This paper doesn't describe this strategy in detail, but it's typically used only when needed because of the complexity it adds.

With remote caches, the orchestration between caching the data and managing the validity of the data is managed by your applications and/or processes that use it. The cache itself is not directly connected to the database but is used adjacently to it.

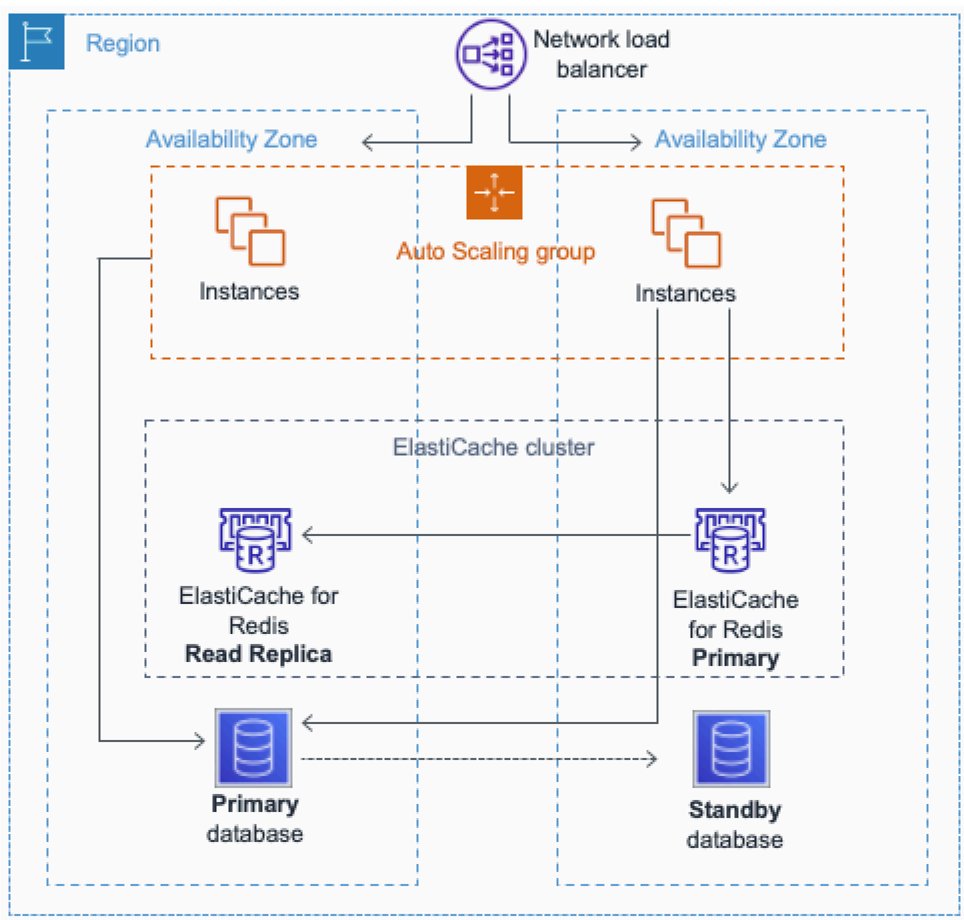
The remainder of this paper focuses on using remote caches, and specifically Amazon ElastiCache (Redis OSS), for caching relational database data.

Caching patterns

When you are caching data from your database, there are caching patterns for Redis and Memcached that you can implement, including proactive and reactive approaches. The patterns you choose to implement should be directly related to your caching and application objectives.

Two common approaches are cache-aside or lazy loading (a reactive approach) and write-through (a proactive approach). A cache-aside cache is updated after the data is requested. A write-through cache is updated immediately when the primary database is updated. With both approaches, the application is essentially managing what data is being cached and for how long.

The following diagram is a typical representation of an architecture that uses a remote distributed cache.

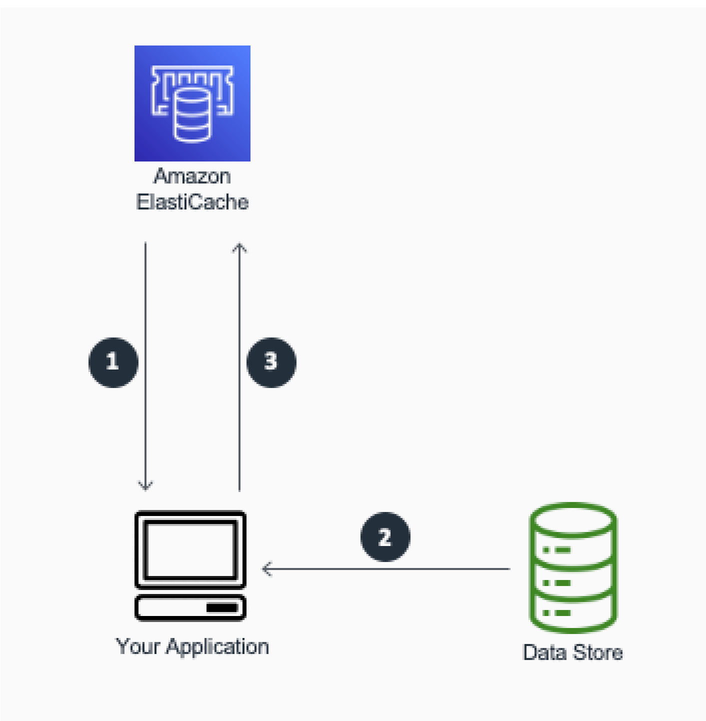


Architecture using remote distributed cache

Cache-Aside (Lazy Loading)

A cache-aside cache is the most common caching strategy available. The fundamental data retrieval logic can be summarized as follows:

1. When your application needs to read data from the database, it checks the cache first to determine whether the data is available.
2. If the data is available (*a cache hit*), the cached data is returned, and the response is issued to the caller.
3. If the data isn't available (*a cache miss*), the database is queried for the data. The cache is then populated with the data that is retrieved from the database, and the data is returned to the caller.



A cache-aside cache

This approach has a couple of advantages:

- The cache contains only data that the application actually requests, which helps keep the cache size cost-effective.

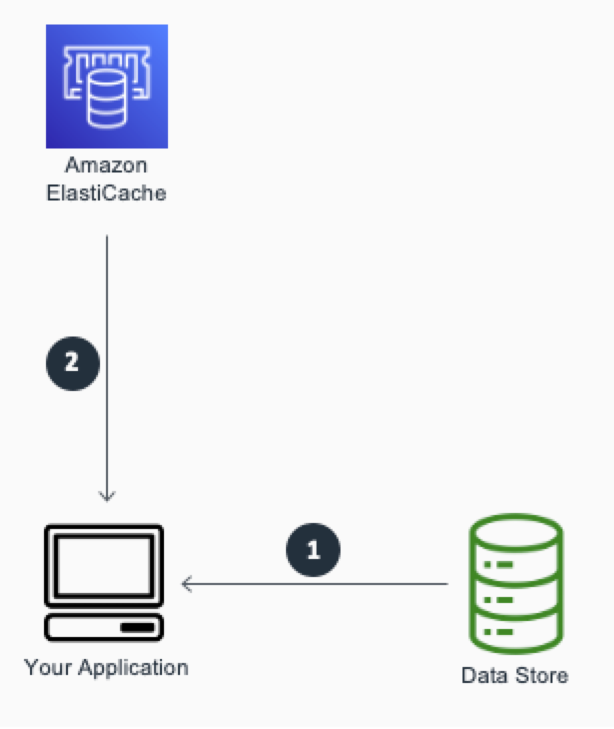
- Implementing this approach is straightforward and produces immediate performance gains, whether you use an application framework that encapsulates lazy caching or your own custom application logic.

A disadvantage when using cache-aside as the only caching pattern is that because the data is loaded into the cache only after a cache miss, some overhead is added to the initial response time because additional roundtrips to the cache and database are needed.

Write-Through

A write-through cache reverses the order of how the cache is populated. Instead of lazy-loading the data in the cache after a cache miss, the cache is proactively updated immediately following the primary database update. The fundamental data retrieval logic can be summarized as follows:

1. The application, batch, or backend process updates the primary database.
2. Immediately afterward, the data is also updated in the cache.



A write-through cache

The write-through pattern is almost always implemented along with lazy loading. If the application gets a cache miss because the data is not present or has expired, the lazy loading pattern is performed to update the cache.

The write-through approach has a couple of advantages:

- Because the cache is up-to-date with the primary database, there is a much greater likelihood that the data will be found in the cache. This, in turn, results in better overall application performance and user experience.
- The performance of your database is optimal because fewer database reads are performed.

A disadvantage of the write-through approach is that infrequently-requested data is also written to the cache, resulting in a larger and more expensive cache.

A proper caching strategy includes effective use of both write-through and lazy loading of your data and setting an appropriate expiration for the data to keep it relevant and lean.

Cache Validity

You can control the freshness of your cached data by applying a time to live (TTL) or *expiration* to your cached keys. After the set time has passed, the key is deleted from the cache, and access to the origin data store is required along with reaching the updated data.

Two principles can help you determine the appropriate TTLs to apply and the types of caching patterns to implement. First, it's important that you understand the rate of change of the underlying data. Second, it's important that you evaluate the risk of outdated data being returned back to your application instead of its updated counterpart.

For example, it might make sense to keep static or reference data (that is, data that is seldom updated) valid for longer periods of time with write-throughs to the cache when the underlying data gets updated.

With dynamic data that changes often, you might want to apply lower TTLs that expire the data at a rate of change that matches that of the primary database. This lowers the risk of returning outdated data while still providing a buffer to offload database requests.

It's also important to recognize that, even if you are only caching data for minutes or seconds versus longer durations, appropriately applying TTLs to your cached keys can result in a huge performance boost and an overall better user experience with your application.

Another best practice when applying TTLs to your cache keys is to add some time jitter to your TTLs. This reduces the possibility of heavy database load occurring when your cached data expires. Take, for example, the scenario of caching product information. If all your product data expires at the same time and your application is under heavy load, then your backend database has to fulfill all the product requests. Depending on the load, that could generate too much pressure on your database, resulting in poor performance. By adding slight jitter to your TTLs, a randomly-generated time value (for example, TTL = your initial TTL value in seconds + jitter) would reduce the pressure on your backend database and also reduce the CPU use on your cache engine as a result of deleting expired keys.

Evictions

Evictions occur when cache memory is overfilled or is greater than the **maxmemory** setting for the cache, causing the engine to select keys to evict to manage its memory. The keys that are chosen are based on the eviction policy you select.

By default, Amazon ElastiCache (Redis OSS) sets the **volatile-lru** eviction policy for your Redis cluster. When this policy is selected, the least recently used (LRU) keys that have an expiration (TTL) value set are evicted. Other eviction policies are available and can be applied in the configurable **maxmemory-policy** parameter.

The following table summarizes eviction policies:

Eviction Policy	Description
allkeys-lru	The cache evicts the least recently used (LRU) keys regardless of TTL set.
allkeys-lfu	The cache evicts the least frequently used (LFU) keys regardless of TTL set.
volatile-lru	The cache evicts the least recently used (LRU) keys from those that have a TTL set.
volatile-lfu	The cache evicts the least frequently used (LFU) keys from those that have a TTL set.
volatile-ttl	The cache evicts the keys with the shortest TTL set.
volatile-random	The cache randomly evicts keys with a TTL set.
allkeys-random	The cache randomly evicts keys regardless of TTL set.
noeviction	The cache doesn't evict keys at all. This blocks future writes until memory frees up.

A good strategy in selecting an appropriate eviction policy is to consider the data stored in your cluster and the outcome of keys being evicted.

Generally, LRU-based policies are more common for basic caching use cases. However, depending on your objectives, you might want to use a TTL or random-based eviction policy that better suits your requirements.

Also, if you are experiencing evictions with your cluster, it is usually a sign that you should scale up (that is, use a node with a larger memory footprint) or scale out (that is, add more nodes to your cluster) to accommodate the additional data. An exception to this rule is if you are purposefully relying on the cache engine to manage your keys by means of eviction, also referred to as an LRU cache.

In addition to the existing time-based LRU policy, Amazon ElastiCache (Redis OSS) also supports a least frequently used (LFU) eviction policy for evicting keys. The LFU policy, which is based on frequency of access, provides a better cache hit ratio by keeping frequently used data in-memory; it traces access counter for each object and evicts keys according to the counter. Every time the object is touched, it reduces the counter after a period called the decay period. This means data used rarely is evicted while the data used often has a higher chance of remaining in the memory.

Relational Database Caching Techniques

Many of the caching techniques that are described in this section can be applied to any type of database. However, this paper focuses on relational databases because they are the most common database caching use case.

The basic paradigm when you query data from a relational database includes executing SQL statements and iterating over the returned **ResultSet** object cursor to retrieve the database rows. There are several techniques you can apply when you want to cache the returned data. However, it's best to choose a method that simplifies your data access pattern and/or optimizes the architectural goals that you have for your application.

To visualize this, this whitepaper will examine snippets of Python code to explain the logic. You can find additional information on the [AWS caching site](). The examples use the [redis-py]() Redis client library for connecting to Redis, although you can use any other Python Redis library.

Assume that you issued the following SQL statement against a customer database for `CUSTOMER_ID 1001`. This whitepaper will examine the various caching strategies that you can use.

```
SELECT FIRST_NAME, LAST_NAME, EMAIL, CITY, STATE, ADDRESS,
COUNTRY FROM CUSTOMERS WHERE CUSTOMER_ID = “1001”;
```

The query returns this record:

Python example:

```
try:
    cursor.execute(key)
    results = cursor.fetchall()

    for row in results:
        print (row[“FirstName”])
        print (row[“LastName”])
```

Java example:

```
…
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
```

```
Customer customer = new Customer();
customer.setFirstName(rs.getString("FIRST_NAME"));
customer.setLastName(rs.getString("LAST_NAME"));
and so on …
}
…
```

Iterating over the **ResultSet** cursor lets you retrieve the fields and values from the database rows. From that point, the application can choose where and how to use that data.

Assuming that your application framework can't be used to abstract your caching implementation, how do you best cache the returned database data?

Given this scenario, you have many options. The following sections evaluate some options, with focus on the caching logic.

Cache the Database SQL ResultSet

Cache a serialized **ResultSet** object that contains the fetched database row.

- **Advantage:** When data retrieval logic is abstracted (for example, as in a [Data Access Object](https://en.wikipedia.org/wiki/Data_access_object) or DAO layer), the consuming code expects only a **ResultSet** object and does not need to be made aware of its origination. A **ResultSet** object can be iterated over, regardless of whether it originated from the database or was deserialized from the cache, which greatly reduces integration logic. This pattern can be applied to any relational database.
- **Disadvantage:** Data retrieval still requires extracting values from the **ResultSet** object cursor and does not further simplify data access; it only reduces data retrieval latency.

Note: When you cache the row, it's important that it's serializable. The following example uses a **CachedRowSet** implementation for this purpose. When you are using Redis, this is stored as a byte array value.

The following code converts the **CachedRowSet** object into a byte array and then stores that byte array as a Redis byte array value. The actual SQL statement is stored as the key and converted into bytes.

Python example:

```
if not r.exists(pickle.dumps(key)):
```

```
    try:
        cursor.execute(key)
        results = cursor.fetchall()
        r.set(pickle.dumps(key), pickle.dumps(results))
        r.expire(pickle.dumps(key), ttl)
        data = results

    except:
        print (“Error: unable to fetch data.”)
else:
    data = pickle.loads(r.get(pickle.dumps(key)))
```

Java example:

```
…
// rs contains the ResultSet, key contains the SQL statement
   if (rs != null) { //lets write-through to the cache
      CachedRowSet cachedRowSet = new CachedRowSetImpl();
      cachedRowSet.populate(rs, 1);
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      ObjectOutput out = new ObjectOutputStream(bos);
      out.writeObject(cachedRowSet);
      byte[] redisRSValue = bos.toByteArray();
      jedis.set(key.getBytes(), redisRSValue);
      jedis.expire(key.getBytes(), ttl);

   }
…
```

One advantage of storing the SQL statement as the key is that it enables a transparent caching abstraction layer that hides the implementation details. The other added benefit is that you don’t need to create any additional mappings between a custom key ID and the executed SQL statement.

At the time of setting data in the Redis, you are applying the expiry time, which is specified in milliseconds.

For lazy caching/cache aside, you would initially query the cache before executing the query against the database. To hide the implementation details, use the DAO pattern and expose a generic method for your application to retrieve the data.

For example, because your key is the actual SQL statement, your method signature could look like the following:

Python example:

```
getResultSet(sql) # sql is the sql statement
```

Java example:

```
public  ResultSet getResultSet(String key); // key is sql
statement
```

The code that calls (consumes) this method expects only a **ResultSet** object, regardless of what the underlying implementation details are for the interface. Under the hood, the **getResultSet** method executes a GET command for the SQL key, which, if present, is deserialized and converted into a **ResultSet** object.

Python example:

```
def getResultSet(key):

    redisResultSet = None
    redisResultSet = r.get(pickle.dumps(key))

    if redisResultSet:
        rs = pickle.loads(redisResultSet)

    else:
        try:
            cursor.execute(key)
            results = cursor.fetchall()
            r.set(pickle.dumps(key), pickle.dumps(results))
            r.expire(pickle.dumps(key), ttl)
            rs = results

        except:
            print (“Error: unable to fetch data”)

    return rs
```

Java example:

```
public ResultSet getResultSet(String key) {
   byte[] redisResultSet = null;
```

```
   redisResultSet = jedis.get(key.getBytes());
   ResultSet rs = null;
   if (redisResultSet != null) { // if cached value exists, deserialize it and return
 it
      try {
         cachedRowSet = new CachedRowSetImpl();
         ByteArrayInputStream bis = new
ByteArrayInputStream(redisResultSet);
         ObjectInput in = new ObjectInputStream(bis);
         cachedRowSet.populate((CachedRowSet) in.readObject()); rs =
         cachedRowSet;
         } …
} else {
// get the ResultSet from the database, store it in the rs object, then cache it.
…
}
…
return rs;
}
```

If the data is not present in the cache, query the database for it, and cache it before returning.

As mentioned earlier, a best practice would be to apply an appropriate TTL on the keys as well.

For all other caching techniques that we'll review, you should establish a naming convention for your Redis keys. A good naming convention is one that is easily predictable to applications and developers. A hierarchical structure separated by colons is a common naming convention for keys, such as **object:type:id**.

Cache Select Fields and Values in a Custom Format

Cache a subset of a fetched database row into a custom structure that can be consumed by your applications.

- **Advantage:** This approach is easy to implement. You essentially store specific retrieved fields and values into a structure such as JSON or XML and then SET that structure into a Redis string. The format you choose should be something that conforms to your application's data access pattern.
- **Disadvantage:** Your application is using different types of objects when querying for particular data (for example, Redis string and database results). In addition, you are required to parse through the entire structure to retrieve the individual attributes associated with it.

The following code stores specific customer attributes in a customer JSON object and caches that JSON object into a Redis string:

Python example:

```
try:
    cursor.execute(query)
    results = cursor.fetchall()

    for row in results:
        customer = {
            “FirstName”: row[“FirstName”],
            “LastName”: row[“LastName”]
        }
        r.set(“customer:id:” + str(row[“id”]), json.dumps(customer))

except:
    print (“Error: Unable to fetch data.”)
```

Java example:

```
…
// rs contains the ResultSet
while (rs.next()) {
    Customer customer = new Customer();
    Gson gson = new Gson();
    JsonObject customerJSON = new JsonObject();
    customer.setFirstName(rs.getString("FIRST_NAME"));
    customerJSON.add(“first_name”,
gson.toJsonTree(customer.getFirstName() );
    customer.setLastName(rs.getString("LAST_NAME"));
    customerJSON.add(“last_name”, gson.toJsonTree(customer.getLastName()
);
    and so on …
    jedis.set(customer:id:"+customer.getCustomerID(), customerJSON.toString() );
    }
…
```

For data retrieval, you can implement a generic method through an interface that accepts a customer key (for example, customer:id:1001) and an SQL statement string argument. It will also return whatever structure your application requires (for example, JSON or XML) and abstract the underlying details.

Upon initial request, the application executes a GET command on the customer key and, if the value is present, returns it and completes the call. If the value is not present, it queries the database for the record, writes-through a JSON representation of the data to the cache, and returns.

Cache Select Fields and Values into an Aggregate Redis Data Structure

Cache the fetched database row into a specific data structure that can simplify the application's data access.

- **Advantage:** When converting the **ResultSet** object into a format that simplifies access, such as a Redis Hash, your application is able to use that data more effectively. This technique simplifies your data access pattern by reducing the need to iterate over a **ResultSet** object or by parsing a structure like a JSON object stored in a string. In addition, working with aggregate data structures, such as Redis Lists, Sets, and Hashes provides various attribute level commands associated with setting and getting data, and eliminating the overhead associated with processing the data before being able to leverage it.
- **Disadvantage:** Your application is using different types of objects when querying for particular data (for example, Redis Hash and database results).

The following code creates a **HashMap** object that is used to store the customer data. The map is populated with the database data and SET into a Redis.

Python example:

```
try:
    cursor.execute(query)
    customer = cursor.fetchall()
    r.hset(“customer:id:” + str(customer[“id”]), “FirstName”, customer[0][“FirstName”])
    r.hset(“customer:id:” + str(customer[“id”]), “LastName”, customer[0][“LastName”])

except:
    print (“Error: Unable to fetch data.”)
```

Java example:

```
…
```

```
// rs contains the ResultSet
    while (rs.next()) {
        Customer customer = new Customer();
        Map<String, String> map = new HashMap<String, String>();
        customer.setFirstName(rs.getString("FIRST_NAME"));
        map.put("firstName", customer.getFirstName());
        customer.setLastName(rs.getString("LAST_NAME"));
        map.put("lastName", customer.getLastName());
        and so on …
        jedis.hmset(customer:id:"+customer.getCustomerID(), map);
        }
…
```

For data retrieval, you can implement a generic method through an interface that accepts a customer ID (the key) and an SQL statement argument. It returns a **HashMap** to the caller. Just as in the other examples, you can hide the details of where the map is originating from. First, your application can query the cache for the customer data using the customer ID key. If the data is not present, the SQL statement executes and retrieves the data from the database. Upon retrieval, you may also store a hash representation of that customer ID to lazy load.

Unlike JSON, the added benefit of storing your data as a hash in Redis is that you can query for individual attributes within it. Say that for a given request you only want to respond with specific attributes associated with the customer Hash, such as the customer name and address. This flexibility is supported in Redis, along with various other features, such as adding and deleting individual attributes in a map.

Cache Serialized Application Object Entities

Cache a subset of a fetched database row into a custom structure that can be consumed by your applications.

- **Pro:** Use application objects in their native application state with simple serializing and deserializing techniques. This can rapidly accelerate application performance by minimizing data transformation logic.
- **Con:** Advanced application development use case.

The following code converts the customer object into a byte array and then stores that value in Redis:

Python example:

```
try:
    cursor.execute(query)
    results = cursor.fetchall()
    r.set(pickle.dumps(key), pickle.dumps(results))
    r.expire(pickle.dumps(key), ttl)

except:
    print (“Error: Unable to fetch data.”)

#pickle.loads(r.get(pickle.dumps(key)))
```

Java example:

```
….
// key contains customer id
    Customer customer = (Customer) object;
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    ObjectOutput out = null;

    try {
        out = new ObjectOutputStream(bos);
        out.writeObject(customer); out.flush();
        byte[] objectValue = bos.toByteArray();
        jedis.set(key.getBytes(), objectValue);
        jedis.expire(key.getBytes(), ttl);

    }
…
```

The key identifier is also stored as a byte representation and can be represented in the customer:id:1001 format.

As the other examples show, you can create a generic method through an application interface that hides the underlying details method details. In this example, when instantiating an object or hydrating one with state, the method accepts the customer ID (the key) and either returns a customer object from the cache or constructs one after querying the backend database. First, your application queries the cache for the serialized customer object using the customer ID. If the data is not present, the SQL statement executes and the application consumes the data, hydrates the customer entity object, and then lazy loads the serialized representation of it in the cache.

Python example:

```
def getObject(key):

    customer = None
    customer = r.get(key)

    if customer:
        customer = pickle.loads(customer)

    else:
        objectData = key.split(“:”)

        try:
            query = “SELECT * FROM customers WHERE id = ‘%d’ LIMIT 1” %
 (int(objectData[2]))
            cursor.execute(query)
            results = cursor.fetchall()
            r.set(key, pickle.dumps(results))
            r.expire(key, ttl)
            customer = results

        except:
            print (“Error: Unable to fetch data.”)
    return customer

#result = getObject(“customer:id:1001”)
```

Java example:

```
public Customer getObject(String key) {

    Customer customer = null;
    byte[] redisObject = null;
    redisObject = jedis.get(key.getBytes());

    if (redisObject != null) {

        try {

            ByteArrayInputStream in = new
ByteArrayInputStream(redisObject);
            ObjectInputStream is = new ObjectInputStream(in);
```

```
            customer = (Customer) is.readObject();

        } …
            } …
    return customer;
}
```

Additional Caching with Redis

Redis is traditionally used in a database cache setting. However, it can also be used to cache output from other services, or full objects from storage services such as Amazon Simple Storage Service (Amazon S3).

The low latency and high throughput of Amazon ElastiCache (Redis OSS), coupled with the large item storage availability, make it a great choice for further optimizing throughput and scalability to new and existing applications.

Object Caching with Amazon S3

Amazon S3 is the persistent store for applications such as data lakes, media catalogs, and website-related content. These applications often have latency requirements of 10 ms or less, with frequent object requests on 1–10% of the total stored S3 data. Many customers achieve this by directly writing to S3. Some applications, such as media catalog updates, require high frequency reads and consistent throughput. For such applications, customers often complement S3 with Redis, to reduce the S3 retrieval cost and to improve performance.

By using Amazon ElastiCache (Redis OSS), applications can maintain a consistent and low-latency throughput, sustained at less than 5 ms, when serving this content outside of S3 at scale. Serving heavily-requested objects via Amazon ElastiCache (Redis OSS) in this manner can enable you to meet performance goals, while also reducing retrieval and transfer costs.

A blog post on how to Turbocharge Amazon S3 with Amazon ElastiCache (Redis OSS) covers how to set up, deploy, and organize data for this purpose.

Amazon ElastiCache and Self-Managed Redis

Redis is an open source, in-memory data store that has become the most popular key/value engine in the market. Much of its popularity is due to its support for a variety of data structures as well as other features, including Lua scripting support and Pub/Sub messaging capability. Other added benefits include high availability topologies with support for read replicas and the ability to persist data.

Amazon ElastiCache offers a fully-managed service for Redis. This means that all the administrative tasks associated with managing your Redis cluster (including monitoring, patching, backups, and automatic failover), are managed by Amazon. This lets you focus on your business and your data instead of your operations.

Other benefits of using Amazon ElastiCache (Redis OSS) over self-managing your cache environment include the following:

- An enhanced Redis engine that is fully compatible with the open source version but that also provides added stability and robustness.
- Easily modifiable parameters, such as eviction policies, buffer limits, etc.
- Ability to scale and resize your cluster to terabytes of data.
- Hardened security that lets you isolate your cluster within Amazon Virtual Private Cloud (Amazon VPC).

For more information about Redis or Amazon ElastiCache, see the Further reading section at the end of this whitepaper.

Redis Engine Support

You can use Amazon ElastiCache (Redis OSS) to build HIPAA-compliant applications. To help do this, you can enable at-rest encryption, in-transit encryption, and Redis AUTH when you create a Redis cluster using ElastiCache (Redis OSS) versions 3.2.6 or later.

Amazon ElastiCache (Redis OSS) version 6.x also support a feature called Role-Based Access Control (RBAC), which can be used instead of authenticating users with the Redis AUTH command. With RBAC, users can be created with specific permissions by using an access string. Users can also be assigned to user groups aligned with a role that is specific to your use cases. More information on RBAC can be found at: Authenticating Users with Role-Based Access Control documentation.

In versions 5.0.3 and later, Amazon ElastiCache (Redis OSS) adds dynamic network processing to enhance I/O handling. By utilizing the extra CPU power available in nodes with four or more vCPUs, Amazon ElastiCache transparently delivers an increase of up to 83% in throughput and up to 47% reduction in latency per node. That is in addition to the significant performance improvements delivered with the introduction of R5 and M5 instances on the service. You can now benefit from the enhanced I/O handling to further boost application performance and reduce costs. Refer to the Amazon Elasticache Pricing page.

Amazon ElastiCache (Redis OSS) improves throughput and reduces latency by leveraging more cores for processing I/O and dynamically adjusting to the workload. These improvements work best for applications that require a large number of concurrent connections to the Redis server. No client-side changes are needed.

At the time of publication, Amazon ElastiCache (Redis OSS) supports Redis version 6.0.5 and earlier.

Because the newer Redis versions provide a better and more stable experience, Redis versions 2.6.13, 2.8.6, and 2.8.19 are deprecated when using the Amazon ElastiCache console. We recommend against using these Redis versions. If you need to use one of them, work with the AWS CLI or Amazon ElastiCache API. The latest version information can be found on the Supported ElastiCache (Redis OSS) Versions page.

Available Instance Types

Amazon ElastiCache supports multiple instance node types. Generally speaking, the current generation types provide more memory and computational power at lower cost when compared to their equivalent previous generation counterparts.

Furthermore, you can launch both ElastiCache (Redis OSS) and Memcached on Graviton2 M6g and R6g instance families. The latest Graviton2 instance types offer you ultra-low latency and high throughput, and you can now enjoy a price/performance improvement of up to 45% over previous generation instances. Graviton2 instances are now the default choice for ElastiCache customers.

You can launch general-purpose burstable T*-Standard cache nodes in Amazon ElastiCache. These nodes provide a baseline level of CPU performance with the ability to burst CPU usage at any time until the accrued credits are exhausted. A *CPU credit* provides the performance of a full CPU core for one minute.

AWS Nitro System

The AWS Nitro System is the underlying platform for the next generation of EC2 instances that enables AWS to innovate faster, further reduce cost for customers, and deliver added benefits like increased security and new instance types.

AWS has completely re-imagined its virtualization infrastructure. Traditionally, hypervisors protect the physical hardware and bios, virtualize the CPU, storage, and networking, and provide a rich set of management capabilities. With the Nitro System, we are able to break apart those functions and offload them to dedicated hardware and software.

The Nitro System delivers practically all of the compute and memory resources of the host hardware to your instances, resulting in better overall performance.

Additionally, dedicated Nitro Cards enable high speed networking, high speed EBS, and I/O acceleration. To benefit from the AWS Nitro System in Amazon ElastiCache (Redis OSS), choose a cache type from the list of supported instances: either M5 (`cache.m5.xlarge` or higher) or R5 (`cache.r5.xlarge` or higher), or later.

Latest information on supported instances (including listings of previous generation instances), along with details around Burstable types can be found on the following page: Supported Node Types.

Redis Cluster Modes: Enabled and Disabled

Cluster Mode for Amazon ElastiCache (Redis OSS) is a feature that enables you to scale your Redis cluster without any impact on the cluster performance. While you initiate a scale-out operation by adding a specified number of new shards to a cluster, you also initiate scale-up or scale-down operation by selecting a new desired node type, Amazon ElastiCache (Redis OSS) a new cluster synchronizing the new nodes with the previous ones. Amazon ElastiCache (Redis OSS) supports three cluster configurations (Redis (single node), Redis (cluster mode disabled), and Redis (cluster mode enabled)), depending on your reliability, availability, and scaling requirements.

With Cluster Mode enabled, your Redis cluster can now scale horizontally (in or out) in addition to scaling vertically (up and down). In a Redis cluster with cluster mode disabled, you can have up to five read replicas in your replication group. Adding or removing replicas incurs no downtime to your application. In a Redis cluster with cluster mode enabled, clusters can have up to ninety shards by default (which can be increased if requested) and up to five read replicas in each node group.

Amazon ElastiCache (Redis OSS) with Cluster Mode enabled will help you to architect a cluster with unpredictable network and storage requirements, or with a write-heavy workload. This horizontal scalability is achieved by preparing a plan that results in an even distribution of the key spaces, which distributes the hash slots to the available shards within the cluster, and thus by spreading the workload over a greater number of nodes. By default, the hash slots get evenly distributed between shards, but customers can also configure a custom hash slot. It is recommended to resize your cluster during off-peak hours.

Reader Endpoint

Amazon ElastiCache (Redis OSS) cluster with multiple nodes and with cluster mode disabled provides the reader endpoint to direct all your read traffic to a single cluster level endpoint. For more information on reader endpoints, see Finding Replication Group Endpoints.

This reader endpoint splits your incoming read connection requests evenly between all read replicas. This reduces the need for the clients to direct traffic to an individual replica, and simplifies configuration to look up and access cached data. Reader endpoint also helps your Amazon ElastiCache (Redis OSS) cluster to load balance all read traffic, as well as to achieve High Availability by placing read replicas in different AWS Availability Zones (AZ).

Reader endpoints works with ElastiCache (Redis OSS) clusters with cluster-mode disabled. For more information, see Finding Replication Group Endpoints.

Amazon ElastiCache (Redis OSS) Global Datastore

Amazon ElastiCache (Redis OSS) Global Datastore is a feature that has been introduced to handle two primary use cases: the ability to have global low latency reads and to support Disaster Recovery scenarios. For more information, see Replication across AWS Regions using global datastores.

With the Amazon ElastiCache (Redis OSS) you can now architect a global application with extremely low latency local reads by reading from a geo-local Global Datastore, and also maintain cluster resiliency at the same time. In the unlikely event of the primary region degrading, it can failover to a secondary region.

Each Global Datastore is a collection of one and more clusters that replicates to one another: A Primary (Active) Cluster (that accepts writes which are replicated to all clusters) and a Secondary (Passive) Cluster (that only accepts read requests and replicates data updates from a Primary cluster). Applications with media contents can now write to an active cluster and the same content can be read in the local regions.

Amazon ElastiCache (Redis OSS) Global Datastore lets you create a reliable, secure and fully-managed cross-region replication that enables you to easily promote your secondary cluster to primary. During cross-region replication, Global Datastore can be set up on new cluster as well as existing cluster, provided the cluster is running the latest Redis engine version (5.0.6 or later). Scalability is built in to Global Datastore, with regional clusters that can be scaled both vertically and horizontally by modifying Global Datastore without any interruption.

Amazon ElastiCache (Redis OSS) Global Datastore enables encryption in transit for cross-region communication in addition to encryption at rest.

Sizing Best Practices Related to Workloads

With Redis version 5.0.5, you can scale up or scale down your Amazon ElastiCache (Redis OSS) cluster online without any downtime. This enables your cluster to stay online and respond to incoming requests while scaling.

To increase read and write capacity, you can scale up your cluster by selecting a larger node type, and alternatively read and write capacity can be reduced by selecting a smaller node. Amazon ElastiCache (Redis OSS) can resize your cluster dynamically without any downtime.

You can also scale out read capacity by adding read replicas and write capacity by adding a specified number of new shards to a cluster.

To ensure uninterrupted scaling, you need to follow these best practices:

1. To scale up, you need to ensure sufficient number of ENI (Elastic Network Interface) availability.
2. To scale down, you need to ensure smaller nodes have adequate memory to absorb the incoming traffic.
3. Perform scaling activities when traffic towards your cluster is at a minimum. Although the scaling process is designed to remain online, this will help to synchronize data to newer nodes.
4. Always test your application in a development environment, where possible.

Conclusion

Modern applications can't afford poor performance. Today's users have low tolerance for slow-running applications and poor user experiences. When low latency and scaling databases are critical to the success of your applications, it's imperative that you use database caching.

Amazon ElastiCache provides two managed in-memory key value stores that you can use for database caching. With zero-downtime scalability, Global datastore for regional low-latency endpoints, and a simplified approach to running a Clustered, distributed cache without the administrative tasks associated with it, Amazon ElastiCache (Redis OSS) is the primary choice for engineering teams and customers.

Contributors

The following individuals and organizations contributed to this document:

- Michael Labib, Sr. Manager, NoSQL & Blockchain Technologies, AWS
- Pratip Bagchi, Partner Solutions Architect, AWS
- Mike Mackay, Sr NoSQL Specialist Solutions Architect, AWS

Document Revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Minor update	Bug fixes and numerous minor changes throughout.	April 1, 2022
Whitepaper updated	Updated for latest services, resources, and technologies.	March 8, 2021
Initial Publication	First Publication	May 1, 2017

Further reading

For more information, see the following resources:

- Performance at Scale with Amazon ElastiCache (AWS whitepaper)
- Full Redis command list

Notes

1 https://aws.amazon.com/rds/aurora/

2 https://redis.io/download

3 https://memcached.org/

4 https://aws.amazon.com/elasticache/redis/

5 https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Strategies.html

6 https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html

7 https://redis.io/topics/lru-cache

8 https://www.lua.org/

9 https://aws.amazon.com/vpc/

10 https://aws.amazon.com/caching/

11 https://github.com/andymccurdy/redis-py

12 http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

13 https://docs.aws.amazon.com/whitepapers/latest/scale-performance-elasticache/scale-performance-elasticache.html

14 https://redis.io/commands

Notices