

Database Caching Strategies Using Redis

May 2017

This paper has been archived.

For the latest technical content, see

<https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/welcome.html>



Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2017 Amazon Web Services, Inc. or its affiliates. All rights reserved.

Archived

Contents

Database Challenges	1
Types of Database Caching	1
Caching Patterns	3
Cache-Aside (Lazy Loading)	4
Write-Through	5
Cache Validity	6
Evictions	7
Amazon ElastiCache and Self-Managed Redis	8
Relational Database Caching Techniques	9
Cache the Database SQL ResultSet	10
Cache Select Fields and Values in a Custom Format	13
Cache Select Fields and Values into an Aggregate Redis Data Structure	14
Cache Serialized Application Object Entities	15
Conclusion	17
Contributors	17
Further Reading	17

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](#), [MongoDB](#), and [Apache Cassandra](#). The best part of caching is that it's easy to implement, and it dramatically improves the speed and scalability of your application.

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

Archived

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 vertically scaled up in a 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.¹ 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 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 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](#)² and [Memcached](#).³ They provide hundreds of thousands and up to a million requests per second per cache node. Many solutions, such as [Amazon ElastiCache for Redis](#), also provide the high availability needed for critical workloads.⁴

The average latency of a request to a remote cache is on the sub-millisecond timescale, which is orders of magnitude 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 use. 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 for Redis, for caching relational database data.

Caching Patterns

When you are caching data from your database, there are caching patterns for [Redis](#)⁵ and [Memcached](#)⁶ 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.

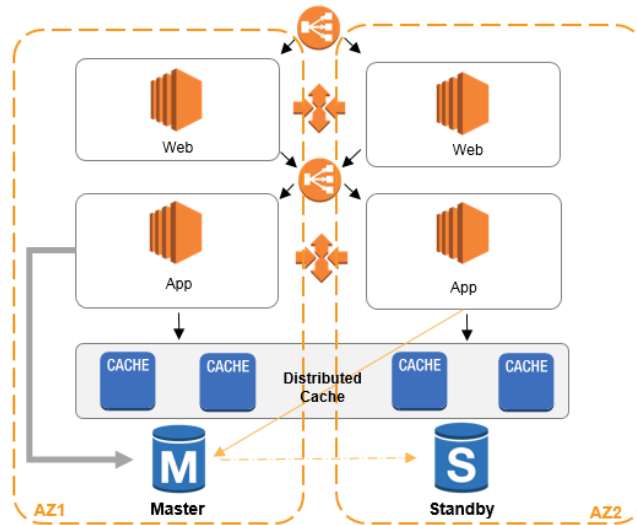


Figure 1: Architecture using remote distributed cache

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.

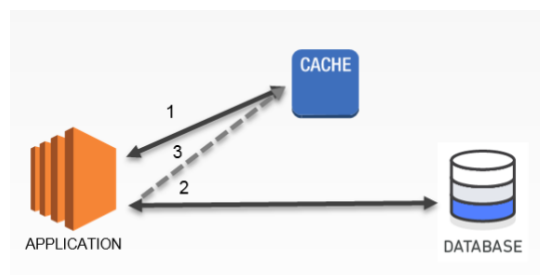


Figure 2: A cache-aside cache

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.

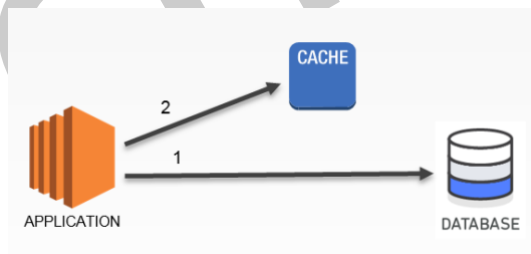


Figure 3: A write-through cache

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 type 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 (e.g., 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 selecting keys to evict in order to manage its memory. The keys that are chosen are based on the eviction policy you select.

By default, Amazon ElastiCache for Redis sets the **volatile-lru** eviction policy to your Redis cluster. When this policy is selected, the least recently used 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.

Eviction Policy	Description
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.
no- eviction	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, least recently used (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](#).⁷

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 support⁸ 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 for Redis 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 \(Amazon VPC\)](#).⁹

For more information about Redis or Amazon ElastiCache, see the [Further Reading](#) section at the end of this whitepaper.

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, we'll examine snippets of Java code to explain the logic. You can find additional information on the AWS caching site.¹⁰ The examples use the [Jedis](#) Redis client library¹¹ for connecting to Redis, although you can use any Java Redis library, including [Lettuce](#)¹² and [Redisson](#).¹³

Assume that you issued the following SQL statement against a customer database for CUSTOMER_ID 1001. We'll 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:

```
...  
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.

Let's also assume 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.

- **Pro:** When data retrieval logic is abstracted (e.g., as in a [Data Access Object](#)¹⁴ 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.
- **Con:** 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.

```
...
// 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();
    ObjectOutputStream out = new ObjectOutputStream(bos);
    out.writeObject(cachedRowSet);
    byte[] redisRSValue = bos.toByteArray();
    jedis.set(key.getBytes(), redisRSValue);
    jedis.expire(key.getBytes(), ttl);
}
...
```

The nice thing about 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.

The last statement executes an expire command to apply a TTL to the stored key. This code follows our write-through logic in that upon querying the database, the cached value is stored immediately afterward.

For lazy caching, 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:

```
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.

```
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.

- **Pro:** 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.
- **Con:** Your application is using different types of objects when querying for particular data (e.g., 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:

```
...
// 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 (e.g., customer:id:1001) and an SQL statement

string argument. It will also return whatever structure your application requires (e.g., JSON, 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.

- **Pro:** 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 provide various attribute level commands associated with setting and getting data, eliminating the overhead associated with processing the data before being able to leverage it.
- **Con:** Your application is using different types of objects when querying for particular data (e.g., 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.

```
...
// 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:

```
....  
// 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.

```

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;  
}
```

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. A managed service further simplifies using a cache in that it removes the administrative tasks associated with supporting it.

Contributors

The following individuals and organizations contributed to this document:

- Michael Labib, Specialist Solutions Architect, AWS

Further Reading

For more information, see the following resources:

- [Performance at Scale with Amazon ElastiCache \(AWS whitepaper\)](#)¹⁵
- [Full Redis command list](#)¹⁶

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/redis-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/xetorthio/jedis>

¹² <https://github.com/wg/lettuce>

¹³ <https://github.com/redisson/redisson>

¹⁴ <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

¹⁵ <https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf>

¹⁶ <https://redis.io/commands>