WHITE PAPER

Why Your MySQL Needs Redis

Roshan Kumar, Redis Labs

CONTENTS

Introduction

Enterprises must innovate to compete and survive in the market. If you are an architect, CIO or a CTO who is responsible for innovation in your enterprise, you face two questions:

1. **How to introduce new features, applications and solutions faster?**
2. **How to achieve scale economies and reduce costs?**

The "legacy play" is usually the biggest factor that hinders enterprises from innovating faster. Replacing legacy solutions with newer ones is often seen as expensive and risky. However, legacy solutions don't easily scale in a linear fashion. Their pricing models combined with the operational overhead make their growth cost prohibitive. This is where open source software and cloud-native application development can provide future-proof solutions.

With some creativity you could address all of the questions mentioned above in one shot: by using Redis.

Redis enables you to boost the performance of your applications without the need to replace your backend databases. It is a high speed, low latency, in-memory database giving your developers more power to develop highly engaging applications.

Redis enhances the value of your investment by delivering your current and future needs. It is open source software with an active community and talent pool behind it. With client libraries available in over sixty programming languages, you could integrate most of your applications with Redis. Redis is lightweight and flexible. You could deploy it as a software on-premises, as a container or on the cloud. To further fulfill your cloud adoption and production platform needs, you can deploy Redis Enterprise, which delivers enterprise-grade high availability, auto scaling, high throughput and 24x7 technical support. Redis Enterprise is available as downloadable software for on-premises deployment, or as a fully-managed database-as-a-service on all the popular cloud platforms for your cloud-native applications.

If you have MySQL or a similar relational database, complementing it with Redis reduces your costs, speeds up your time to market the new applications, and allows you and your team to focus on innovating new solutions.

Typical Architecture of a Legacy System

Traditional applications built with relational databases such as MySQL typically follow the architecture shown in Figure 1. The applications—consumer-facing and in-house enterprise applications—are powered by application servers that maintain sessions and manage data. The application servers in turn connect to one or more databases via a data access layer (as shown in the picture below). The data access layer provides a layer of abstraction between the applications and the actual databases. We will use this as a reference architecture to demonstrate how Redis complements your relational database.

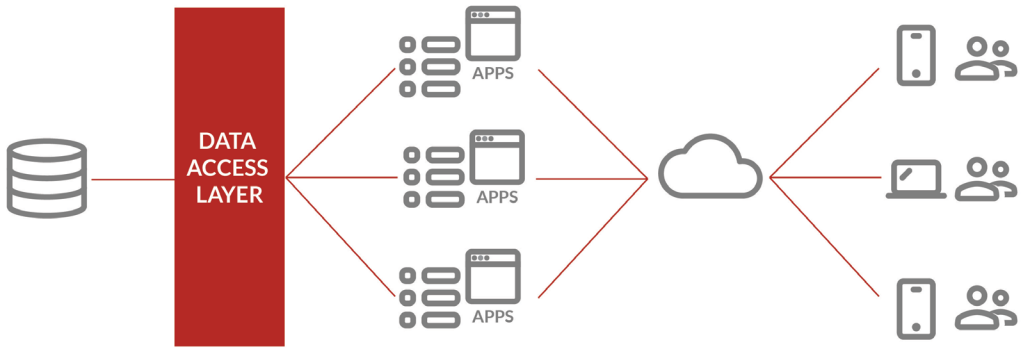

Figure 1. A typical architecture that connects applications to the databases via a data access layer

Why is it hard to innovate and scale with the traditional architecture?

Some of the reasons that make it difficult to deliver new applications and scale to accommodate growing business needs using traditional architecture are:

1. **Unsatisfactory user experience:** Quick, low latency response times are very critical for interactive applications. Given that the internet latencies take up about 100 milliseconds, the burden of delivering the real-time interactive experience falls on the applications and their databases; databases are expected to respond with sub-millisecond latency. The read/write speeds of traditional databases are not good enough for use cases such as session stores. Transferring the session data to the application layer may alleviate the problem of slow user experience, however mixing a data store with application features makes the application more complex and may lead to data loss, should the application fail.
2. **Time to develop and deploy new applications:** The architecture described above does provide an abstraction layer for data access. However, introducing new tables, or modifying an existing schema becomes extremely complex for large enterprises. Often, they are put through long approval processes that may take a few weeks, if not months. The change management after the approval is tedious too. One change in the schema may cascade over many programs that access the data. The ideal solution is a database that can work in parallel with the traditional database while offering flexible schema and data structures.
3. **Cost of scaling:** Legacy databases are limited by the number of read and write operations you can perform per second and by the number of concurrent connections you can have. Though in theory it is possible to distribute the database on more than one server, maintaining data consistency across the instances becomes an overhead. In addition to that, given the low latency and slow performance of disk-based databases, the hardware cost may get prohibitive when supporting applications at scale.
4. **Reaching the limits:** When a database receives more queries than it can handle, the queries queue up, making the database unavailable system-wide. As a quick remedy, enterprises end up shutting down applications to limit the load on the database.

How Redis Complements MySQL

Being an in-memory key-value data store, Redis is extremely fast and flexible. It's often called a "Swiss Army Knife" for data processing. It has built-in data structures such as Lists, Hashes, Sets, Sorted Sets, Bitmaps, Hyperloglog, and Geospatial Indices that help you perform some data operations more efficiently and effectively when compared to a relational database. You incorporate Redis into your architecture as a "system of engagement," e.g. the system that stores the hot data that the users engage with, while designating your MySQL as the "system of record," e.g. the database that holds the truth. With this approach you don't disrupt the compliance, regulatory and governance needs of your data.

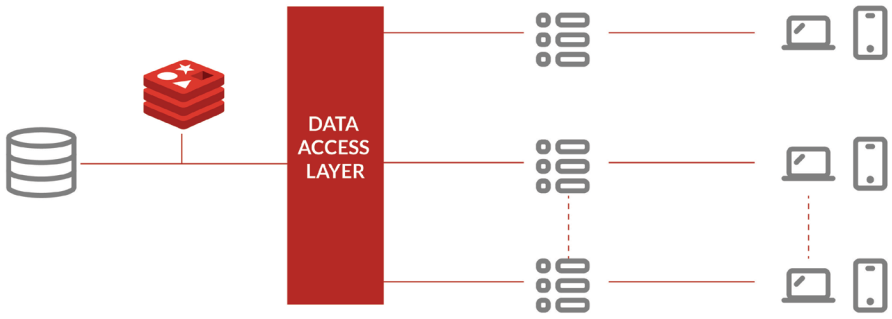

Figure 2. Introducing Redis behind the data access layer

Setting up Redis behind the data access layer (as shown in figure 2) keeps Redis behind the curtains and maintains the flexibility offered by the abstraction provided by the data access layer. The following section demonstrates a few ways that Redis can be used to complement your traditional database.

Redis as a "System of Engagement"

The purpose of a system of engagement is to deliver the best user experience, keeping the user engaged with low-latency responses. You can achieve this without your application, database or network layer creating a bottleneck. The system of engagement complements the system of record, which in this case is MySQL. By using Redis as a system of engagement, the hot data that's to be served to the active users is stored in Redis while a true copy of the data is managed in MySQL. Popular use cases for complementing Redis with MySQL are caching, session stores, rate limiting and more.

1. Caching

Caching provides a tiered model for memory access. Applications could store common, repeatedly read objects in Redis. This helps retrieve data quickly and limit the load on the database server.

When to use

- Frequent reads, infrequent writes
- Data is shared between user sessions
- Examples: images, documents, financial statements, reporting data, etc.

For more information about caching with Redis, download the white paper 15 Reasons Why Caching is Best Done With Redis. If you are wondering how to get started quickly, Appendix A shows two common techniques with which to design a caching solution.

2. Session Store

In all interactive apps, the server maintains a unique session for each active user. The session objects are generally isolated from each other. Traditional designs relied on MySQL-like relational databases to persist session data. Redis takes the session stores to the next level. It enhances the user experience with very low latency. A single cluster of Redis on decently sized servers with sufficient RAM can manage thousands, if not millions of sessions. In addition to storing basic user information, Redis allows you to compute and serve many other session metrics covered in the next section.

Find more ideas on how to get started storing sessions in Redis in Appendix B.

When to use

- Frequent reads and writes
- Data is isolated between sessions
- Examples: personalized applications, e-Commerce, gaming, social applications

3. Counting and Real-time Tracking

Gamification is a mantra for many applications today. Leaderboards, dashboards, polls, messages, counters and other real-time aggregators require constant processing and communication with end-users. Relational databases such as MySQL are designed to handle transactions well, but do a poor job collecting, processing and dissipating information to thousands of active users in real time. Gamified applications give enterprises the opportunity to engage with their available audience. Redis has powerful and highly efficient data structures that enable you to design a real-time gamification platform. Appendix C lists the ideal commands for counting and real-time tracking.

When to use

- Usage tracking, gamification, asynchronous communication
- Counting and tracking millions of simultaneous activities or objects
- Example: Social media, customer support, gaming, e-commerce, etc.

4. Redis for Rate-limiting Calls to the Legacy Servers

One of the major challenges for enterprises as they scale and add more applications is how to manage the load on the legacy

[1]System of Engagement: http://searchcio.techtarget.com/definition/systems-of-engagement

servers during peak usage times. One expensive solution is to add more computational resources to the legacy servers that act as a bottleneck. In cases where MySQL is the bottleneck, one can plan scaling out the database management system. However, this is expensive too, especially when the expansion is not needed during off-peak hours. A cost-effective way to handle the load is to rate limit the number of calls the applications make every few seconds. Redis is a popular tool for implementing rate limiting solutions. For more details visit Redis Enterprise for Metering.

When to use

- If the backend database cannot handle the peak load
- Example: Applications for retail, e-Commerce, inventory management, etc.

Redis, the Swiss Army Knife for Developers

Redis provides your developers with all the tools they need to develop highly engaging, data-centric applications. Given the data structures and highly optimized commands available in Redis with which to manage your data, you can find more than one way to leverage Redis for your applications. In addition to the examples described above, Redis can be used as a message broker, data structure store, and temporary data store for a variety of use cases. Essentially, Redis enables you to get your data closer and faster to your end user. If you flip this around, Redis also enables you to collect the data faster and quicker from your end users. With RAM's increasing affordability and the persistence features in Redis, more and more developers are using Redis as a primary database for both transactions and analytics.

Redis Enterprise takes your experience with Redis to a higher level. It offers critical enterprise capabilities such as high availability, in-memory replication, auto scaling, re-sharding, etc. It also delivers forward-looking features such as CRDT-based active-active support for distributed databases and built-in Redis modules such as RediSearch, ReJSON, Rebloom and Redis Graph. Redis Enterprise gives flexibility to your deployment models; you can deploy it on-premises, in your virtual private cloud as a fully managed service by Redis Labs, or as a hosted database-as-a-service in your public cloud environment. In conclusion, your investment in Redis Enterprise will not only meet your immediate needs, but also make you future-proof.

Appendix A. Examples of Caching Using Redis

There are many resources that can help you get started using Redis as a cache. The following sections help you get started quickly by describing two common techniques for designing the cache solution.

Technique 1: Look-aside Cache: A simple cache with no write-through

This is the quickest way to implement a cache with limited programming overhead. This method works well with aggregation and filter queries. The data follow for this technique is:

Read

1. Get the data from Redis.
2. If the data is not in Redis, then get the data from MySQL, store it in Redis.

Write

1. Always write the data to MySQL.

Figure 3. A simple look-aside cache architecture

Design Steps:

1. Identify the data or the objects that are repeatedly read by the application.
2. Determine the key format.
3. Determine the format or the data structure for the cache (Read more about Redis data structures here: https://redis.io/topics/data-types
4. Agree on a time interval after which the cached data goes stale (time-to-live, expiration).
5. Decide the eviction policy.
6. Implement the logic in the data access layer.

More on "Step 2 – Determine the key format"

Redis is an in-memory, key-value database. The data is stored as a key-value pair. A Redis key is analogous to a primary-key of a table in a relational database. The key is always a String, and the value can be a String or one of the many data structures available in Redis: List, Set, Sorted Set, Hash, Bitfield, and so on. Each String, whether a key or a value, can be as large as 512 MB. However, for optimal performance, try to keep the key as short as possible. The bottom line is, to access any data in Redis, you need to know the key or have a standard formula to construct a unique key for your data.

Maintaining a consistent formula for the keys will make the design simple. For example, suppose your SQL command is the

one shown below and your goal is to cache the results in Redis:

```
select id, name, street, city, state, country
from user_info
where city='San Francisco' and state = 'California'
```

You retrieve the results from MySQL when the query is run for the first time, and store the results in Redis. When you store the results in Redis, you also need a way to access your data. You do that through your unique key. Your key could be the whole SQL statement itself, or you could shorten it to

```
<table_name>:<parameters>
```

In our example, this will translate to:

```
 user_info:city:San Francisco:state:California
```

Depending on the size of the result set and your application's requirement, you could cache paginated results, or the whole result set as a string or a binary object. Step 3 talks more about this.

Please note that if your solution has many queries with small differences in the result set, then you may be better off storing the whole table in a Redis data structure. You may also consider having a write-through cache that is described in the next section of this appendix.

More on "Step 3 - Determine the format or the data structure for the cache"

You can cache the data in multiple ways. If you are developing on Java, for example, you could cache the whole ResultSet object as a binary string in Redis. One drawback here is that you will need to iterate through the ResultSet to retrieve the data every time you look it up. Here's the sample code snippet to store the ResultSet in Redis, and read it back from the cache. The code uses the Jedis library (https://github.com/xetorthio/jedis) to connect to Redis.

Store a ResultSet

```
ResultSet rs = stmnt.executeQuery(query);
CachedRowSet c = new CachedRowSetImpl();
c.populate(rs, 1);
ObjectOutput o = new ObjectOutputStream(new ByteArrayOutputStream());
o.writeObject(c);
jedis.set(key, o.toByteArray());
```

Retrieve a ResultSet

```
CachedRowSet c = new CachedRowSetImpl();
storedObj = jedis.get(key);
if(storedObj != null){
      ObjectInput in = new ObjectInputStream(new ByteArrayInputStream(storedObj));
      c.populate((ResultSet)in.readObject());
}
return (ResultSet) c;
```

The other technique is to extract the data from the ResultSet and store the data in Redis. You can store the result as a single string by inserting delimiters between the rows, in a Hash, a Set or a Sorted Set.

Technique 2: A Write-through Cache

This follows a true tiered-memory model. The MySQL database has the true copy of the complete dataset. Redis will have a

subset of that data. The main difference from the previous version is that the data inside Redis always matches with the data in MySQL. The data flow for this technique is as follows:

Read

1. Access the table equivalent data structure in Redis.
2. If the data structure is not in the cache, reload the missing table from MySQL, store the data in the cache, then re-run the query on the data structure in the cache.

Write

1. Write the data to MySQL.
2. Update the data in Redis. Reload the data in Redis if the table equivalent data structure doesn't exist.

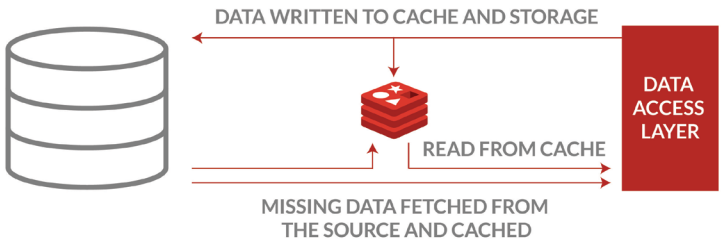

Figure 4. Architecture of the write-through cache

Design Steps:

1. Identify the tables that need to be cached.
2. Determine the data structure equivalents in Redis and how the table and indexes map to Redis data structures.
3. Determine the query and update techniques.
4. Decide the eviction policy.
5. Design the solution for periodic refresh of data from MySQL.
6. Modify the data access layer.

Example:

This simplified example shows you how to design a write-through cache for a MySQL table:

Product			
ID	Name	Description	Price
1000001	AAAA	Description of AAAA	100
1000002	BBBB	Description of BBBB	200
1000003	CCCC	Description of CCCC	200
.	.	.	.
.	.	.	.
1000100	XXYZ	Description of XXYZ	500

Equivalent Redis data structure: Hash

```
PRODUCT_1000001
        N = AAAA
        D = Description of AAAA
        P = 100
PRODUCT_1000002
        N = BBBB
        D = Description of BBBB
        P = 200
PRODUCT_1000003
        N = CCCC
        D = Description of CCCC
        P = 200
        .
        .
PRODUCT_1000100
        N = XXYZ
        D = Description of XXYZ
        P = 500
```

Look up queries based on ID:

1. SQL:

```
select * from Product where id = 1000002;
```

Redis:

```
HGETALL PRODUCT_1000002
```

2. SQL:

```
select name from Product where id = 1000003;
```

Redis:

```
HGET PRODUCT_1000003 name
```

Other lookup queries:

SQL:

```
select * from Product where price < 300;
```

Redis:

Maintain a Sorted Set data structure such that price denotes the weight of each element

Data Structure: Sorted Set

```
Name: PRODUCT_PRICE
100 – 1000001
200 – 1000002
200 – 1000003
```

```
.
.
900 – 1000100
```

Redis commands (pipeline commands for a higher throughput): The first command ZRANGEBYSCORE gets all the product identifiers between the range 0 and 300. The second part of the query, HGETALL gets all the data stored for each product ID. Since we need to repeat HGETALL for each product within the range of 0 and 300, we pipeline the commands between MULTI and EXEC. Pipelining enables you to run multiple commands as if it were a single command. It boosts the performance by reducing the round trips between the client and the server.

```
ZRANGEBYSCORE PRODUCT_PRICE 0 300
MULTI
      HGETALL PRODUCT_1000001
      HGETALL PRODUCT_1000002
      HGETALL PRODUCT_1000003
EXEC
```

Insert/Update queries:

1. SQL:

```
insert into Product (id, name, description, price) values (100200, “ZXYW”,”Descrip-
tion for ZXYW”, 300);
```

Redis:

```
MULTI
      HMSET PRODUCT_100200 N ZXYW D “Description for ZXYW” P 300
      ZADD PRODUCT_PRICE 300 PRODUCT_100200
EXEC
```

2. SQL:

```
update Product set price = 400 where id = 100200;
```

Redis:

```
MULTI
      HSET PRODUCT_100200 P 400
      ZADD PRODUCT_PRICE 400 PRODUCT_100200
EXEC
```

Appendix B: Session Store Examples

One of the primary requirements of a session store is to keep the data isolated between sessions. In other words, the design should ensure that a session cannot access data that belongs to other sessions. While designing a session store in the MySQL world, every session acquires a unique ID that is used as a primary key to access all other data.

You could construct an equivalent design in Redis. How the data is synchronized between Redis and MySQL depends on your use case. The preferred and simplified way to manage sessions is illustrated below. In this design, the data access layer loads the session into Redis at the start of the session. Redis handles all the read and write operations when the session is active. The data access layer will write the data back to MySQL when the session ends or expires. The big question here is, "What happens to the session data if the Redis node fails?" With persistence and replication in Redis, it is possible to achieve a highly available architecture and ensure zero data loss during node failures.

The typical data flow for session store is as follows:

1. Session Start: load all the session data from MySQL to Redis.
2. Save the data in Redis during the session.
3. Save the data back to MySQL upon session exit (or periodically).

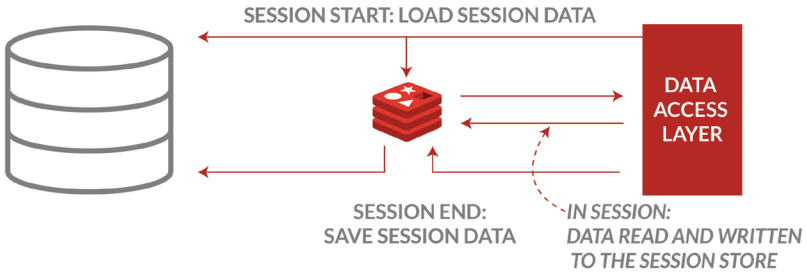

Figure 5. Session store architecture

Steps to design a session store using Redis:

1. Identify the tables that store session data.
2. Determine the Redis equivalent data structures.
3. Determine the query and update techniques.
4. Decide the procedure to load the session data and re-synchronize the data back from Redis to MySQL.
5. Modify the data access layer.

Example:

In the example shown in the table below, the SESSION table maintains the active sessions, and SHOPPING_CART maps session to the products that are in the cart:

SESSION				
ID	USER_ID	PROFILE_NAME	.	START_TIME
192730217	AAAA	Abcd	.	1513968031
349857740	BBBB	Bcde	.	1513968214
485729857	CCCC	Cdef	.	1513968300
.	.	.	.	.
.	.	.	.	.
738272890	XXYZ	Xyzz	.	1513968825

SHOPPING_CART		
SESSION_ID	PRODUCT_SKU	QUANTITY
192730217	98229	4
192730217	83490	1
192730217	78492	1
.	.	
.	.	
738272890	78492	9

Equivalent Redis data structures:

Data structure to store the session data for each active session: Hash

Example:

```
SESSION_192730217
      USER_ID AAAA
      PROFILE_NAME Abcd
      .
      .
      START_TIME 1513968031
```

Data Structure to store shopping cart information for each session: Sorted Set. The product ID is stored in the Sorted Set, and the weight associated with the product specifies the quantity being checked out.

Example:

```
SHOPPING_CART_192730217
1 83490
1 78492
4 98229
```

Sample equivalent queries between MySQL and Redis:

1. SQL:

```
select PRODUCT_SKU, QUANTITY from SHOPPING_CART where SESSION_ID = 192730217;
```

Redis:

```
zrangebyscore SHOPPING_CART_192730217 0 +inf with scores
```

2.SQL:

```
insert into SHOPPING_CART(SESSION_ID, PRODUCT_SKU, QUANTITY) values
(192730217, 78492, 1);
```

Redis:

```
zadd SHOPPING_CART_192730217 1 78492
```

Redis command to load the data from the SESSION table to the Hash data structure:

```
HMSET SESSION_192730217 USER_ID AAAA PROFILE_NAME Abcd … START_TIME 1513968031
```

Appendix C: Redis Commands for Counting and Real-time Tracking

Redis supports atomic commands to increment and decrement values. In addition to this, it offers a lock-free platform whereby multiple application threads could update the same counter and still maintain consistency. To maintain data consistency in MySQL, counting requires each client to acquire a lock to the counter and release it after increasing or decreasing the count. This technique is highly inefficient when counting at scale.

1. Atomic commands to count

Redis provides commands to increment values without requiring reading them to the application's main memory.

Command	Description
INCR key	Increment the integer value of a key by one
INCRBY key increment	Increment the integer value of a key by the given number
INCRBYFLOAT key increment	Increment the float value of a key by the given amount
DECR key	Decrement the integer value of a key by one
DECRBY key decrement	Decrement the integer value of a key by the given number
HINCRBY key field increment	Increment the integer value of a hash field by the given number
HINCRBYFLOAT key field increment	Increment the float value of a hash field by the given amount

2. Native support for data structures and commands optimized to count objects

The data structures in Redis come with counting commands that are optimized to execute efficiently. Some of the data structures help you accomplish more than just counting objects. For example, the Set data structure guarantees uniqueness to all the elements. Sorted Set goes one step ahead—it not only ensures unique elements are added to it, but also allows you to order them based on a score. Ordering your elements by time in a Sorted Set data structure, for example, will offer you a time-series database. With the help of Redis commands you could get your elements in a certain order, or delete items that you don't need anymore. Hyperloglog is another special data structure that helps counting millions of unique items without impacting memory.

Data Structure	Command	Description
List	**LLEN** key	Get the length of a list
Set	**SCARD** key	Get the number of members in a set (cardinality)
Sorted Set	**ZCARD** key	Get the number of members in a sorted set
Sorted Set	**ZLEXCOUNT** key min max	Count the number of members in a sorted set between a given lexicographical range
Hash	**HLEN** key	Get the number of fields in a hash
Hyperloglog	**PFCOUNT** key	Get the approximate cardinality of the set observed by the Hyperloglog data structure
Bitmap	**BITCOUNT** key [start end]	Count set bits in a string

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redislabs.com