ALIBABA CLOUD

Alibaba Cloud

ApsaraDB for Redis
Best Practices

Document Version: 20200901

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	Danger: Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	Warning: Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	Notice: If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	Note: You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Best Practices for Redis Enhanced Edition

1.1. Monitor user trajectories by using TairGIS

You can monitor user trajectories based on points, linestrings, and planes by using the TairGIS data structure that is provided by ApsaraDB for Redis Enhanced Edition (Tair).

Background information

Location-based services (LBS) use various technologies to locate devices in real time, and provide information and basic services for users and devices based on the mobile Internet. In recent years, a large number of industrial applications and research projects have been engaged in LBS technologies. LBS technologies play an important role in many applications.

In 2020, the COVID-19 pandemic poses huge health threats to human beings and presses the pause button around the world. To prevent the COVID-19 pandemic from spreading, China has taken proactive measures by using the power of the entire country. China has made great achievements in controlling the spread of the pandemic. The cities across China start to recover from the wounds of the COVID-19 pandemic. Gradually, employees go back to work, enterprises resume production, and students go back to schools. The spreading of the pandemic in China is basically under control, and the other countries and regions are taking efforts to flatten the curve of the COVID-19 pandemic cases. The epidemic prevention and control are still facing considerable challenges. LBS offers an efficient solution to handle these challenges. LBS allows you to monitor user trajectories to identify risks and ensure the safety of people. LBS can also facilitate epidemiological surveys.

ApsaraDB for Redis Community Edition supports native Redis GEO commands that are offered by the open source Redis service. You can use the native Redis GEO commands to describe location data. However, the native Redis GEO commands allow you to implement only limited features due to the limited granularities of geographic data. Therefore, the Redis GEO commands offer limited support for LBS applications. Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) support TairGIS commands. TairGIS commands provide more features than the native Redis GEO commands.

TairGIS uses R-tree indexes and supports Geographic Information System (GIS) API operations. Redis supports GEO commands that are suitable for Geohash strings and sorted sets. The GEO commands support geospatial indexing for one-dimensional arrays. This allows you to query points. TairGIS supports indexing for two-dimensional arrays. This allows you to query points, lines, and planes. You can use TairGIS to check intersection or inclusion relationships. TairGIS provides more powerful features.

TairGIS allows you to significantly reduce the costs of developing LBS applications. One of the typical applications of TairGIS is geofencing security systems for senior and child care.

Implementation methods

To monitor the trajectories of a specific group of users, you must obtain the location data of the users. To obtain the location data, you can use the following two methods:

- Use the Global Positioning System (GPS) service on the mobile phones of the users. In this method, users must enable the GPS service on their mobile phones.
- Cooperate with telecom carriers.

In scenarios similar to epidemic control, user trajectories are monitored to check whether the users have been to high-risk areas, such as the areas where an epidemic outbreak occurs. Therefore, you do not need to store the previous trajectory data of the users in most cases. You only need to report alerts when users enter high-risk areas. This provides maximum protection for user privacy.

You can use polygons to indicate high-risk areas based on the well-known text (WKT) language, and store the polygons in TairGIS data. You can use points, linestrings, or polygons to indicate user trajectories based on WKT, and store the points, the linestrings, or the polygons in TairGIS data. Then, you can run the TairGIS commands to query the intersections between the user trajectories and the high-risk areas to determine whether a user has passed through the high-risk areas.

> **Note** WKT is a text markup language that you can use to represent vector geometry objects on a map and the spatial reference systems of spatial objects. WKT also allows you to perform transformations between spatial reference systems.

The methods of processing location data varies based on the methods that you use to obtain the location data. The following examples provide details.

Examples

- Use the GPS service to obtain the location data

  After you obtain the current GPS data of a user, you can run the TairGIS GIS.CONTAINS command to check whether the user location is in high-risk areas. If the user is on a road, you can use the GPS data to locate the specific road. Then, run the GIS.INTERSECTS command to check whether the user is moving towards a high-risk area. If the user is moving towards a high-risk area, alerts are reported.

  You can use WKT to describe the GPS data of a user as a point, for example, `POINT(30 11)`. You can use WKT to describe the road information as a linestring, such as `LINESTRING (30 10, 40 40)`. The following sample code helps you understand how to implement the business logic.

  ```
  GIS.ADD your_province your_location 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))' // Add the GPS information of a user to TairGIS data.
  GIS.CONTAINS your_province 'POINT (30 11)'
  GIS.INTERSECTS your_province 'LINESTRING (30 10, 40 40)'
  ```

- Cooperate with telecom carriers to obtain the location data

  In scenarios where base stations are deployed by telecom carriers in a sparse manner, the location data that you obtain indicates an area. The area may be a sector that is covered by a base station or the entire coverage area of the base station. You can use WKT to describe the area as a polygon, for example, `POLYGON ((10 22, 30 45, 16 53, 10 22))`. You can run the GIS.INTERSECTS command to analyze the intersections between the polygon and the high-risk areas. The sample code is provided as follows:

  ```
  GIS.ADD your_province your_location 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))' // Add the location information that you obtain from the base stations of the telecom carrier to the TairGIS data.
  GIS.INTERSECTS your_province 'POLYGON ((10 22, 30 45, 16 53, 10 22))'
  ```

> **Note** For more information about TairGIS commands, see TairGIS commands.

Summary

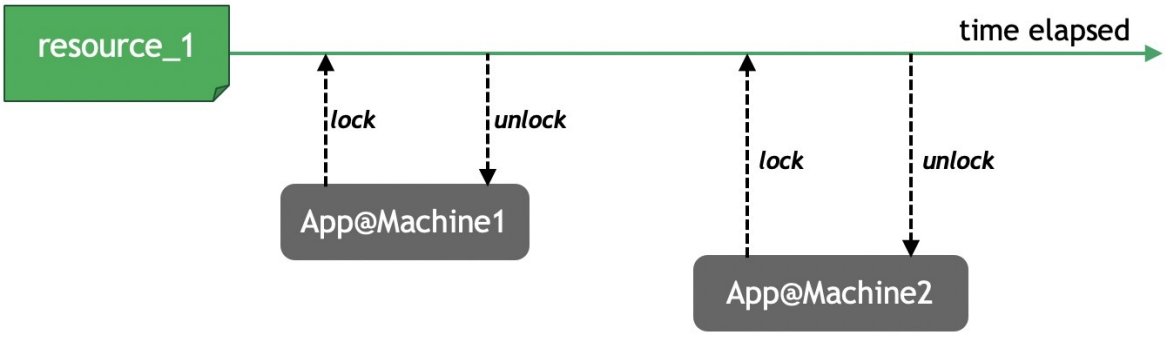
Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provide the TairGIS data structure. TairGIS provides an easy method for you to use LBS applications to store and compute geographic data. TairGIS also allows you to deliver high performance in high-concurrency scenarios.

1.2. High-performance distributed locks

Distributed locks are one of the most widely adopted features in large applications. You can implement distributed locks based on Redis by using various methods. This topic describes and analyzes common methods of implementing distributed locks. This topic also describes the best practices on how ApsaraDB for Redis Enhanced Edition (Tair) implements high-performance distributed locks. The best practices are based on the extensive experience of Alibaba Group in using ApsaraDB for Redis Enhanced Edition (Tair) and distributed locks.

Distributed locks and scenarios

If multiple threads in a process need to have concurrent access to a specified resource, you can use mutexes (also known as mutual exclusion locks) and read/write locks to create applications. If multiple processes on a host need to have concurrent access to a specified resource, you can use interprocess synchronization primitives, such as semaphores, pipelines, and shared memory. However, if multiple hosts need to have concurrent access to a specified resource, you must use distributed locks. The distributed locks are the mutual exclusion locks that have global presence. You can apply distributed locks on the resources in distributed systems to avoid logical failures by preventing race hazards.



Features of distributed locks

- Mutual exclusion

  Only one client can hold a lock at a given moment.

- Deadlock free

  Distributed locks use a lease-based locking mechanism. If a client acquires a lock and encounters an exception, the lock is automatically released after a certain period. This prevents resource deadlocks.

- Consistency

  Failovers may be triggered by external errors or internal errors. External errors include hardware failures and network exceptions, and internal errors include slow queries and system defects. After failovers are triggered, a replica serves as a master to implement high availability for ApsaraDB for Redis. In this scenario, if services have high requirements for mutual exclusion, the lock of the client must remain the same after the failovers.

Implement distributed locks based on native Redis databases

> **Note** The methods described in this section also apply to ApsaraDB for Redis Community Edition.

- Acquire a lock

  Redis provides an easy method that you can use to acquire a lock. This easy method is to run the **SET** command. A command example and the key parameters or options of the command are described as follows:

```
SET resource_1 random_value NX EX 5
```

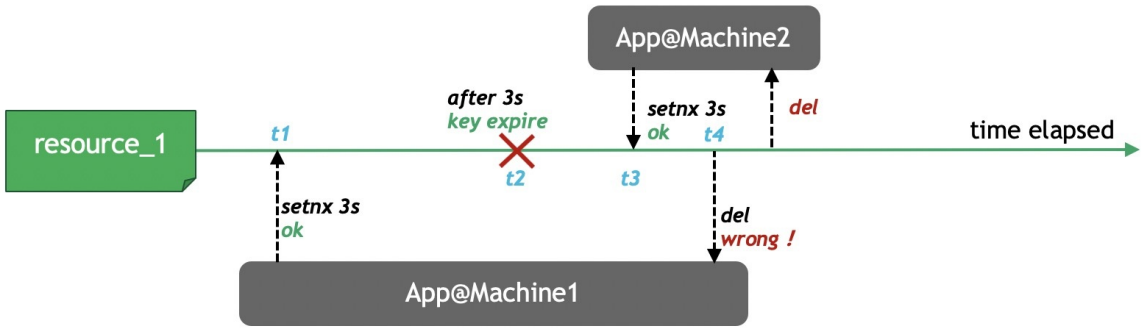
Description of key parameters or options

Parameter or option	Description
resource_1	Specifies the key of the distributed lock. If the key exists, the corresponding resource is locked and cannot be accessed by the other clients.
random_value	Specifies a random string. The value must be unique across clients.
EX	Specifies an expiration time for the key. The unit is seconds. You can also use the PX option to set an expiration time that is measured in milliseconds.
NX	Sets the key only if the key does not exist in Redis.

In the sample code, the expiration time for the resource_1 key is set to five seconds. Therefore, if the client does not release the key, the key expires in five seconds, and the system reclaims the lock. Then, the other clients can lock and access the resource.

- Release a lock

  In most cases, you can run the **DEL** command to release a lock. However, this may cause the following issue.



i. At the t1 time point, the key of the distributed lock is resource_1 for application 1, and the expiration time for the resource_1 key is set to three seconds.

ii. Application 1 remains blocked for more than three seconds due to certain reasons, such as a long response time. Therefore, the resource_1 key expires and the distributed lock is

automatically released at the t2 time point.

iii. At the t3 time point, application 2 acquires the distributed lock.

iv. Application 1 resumes from the block and runs the `DEL resource_1` command at the t4 time point to release the distributed lock that is held by application 2.

This example shows that a lock can be released only by the client that sets the lock. Therefore, before a client runs the **DEL** command to release a lock, the client must run the **GET** command to check whether the lock is set by the client. In most cases, a client uses the following Lua script in Redis to release the lock that is set by the client:

```
if redis.call("get",KEYS[1]) == ARGV[1] then
return redis.call("del",KEYS[1])
else
return 0
end
```

- Renew a lock

  If a client cannot complete the required operations within the lock validity time, the client must renew the clock. A lock can be released only by the client that sets the client. Similarly, a lock can be renewed only by the client that sets the lock. In Redis, a client can use the following Lua script to renew a lock:

```
if redis.call("get",KEYS[1]) == ARGV[1] then
return redis.call("expire",KEYS[1], ARGV[2])
else
return 0
end
```

Implement distributed locks based on ApsaraDB for Redis Enhanced Edition (Tair)

You can implement distributed locks by using the enhanced string commands of the performance-enhanced instances provided by ApsaraDB for Redis Enhanced Edition (Tair). These enhanced commands are an alternative to Lua scripts.

- Acquire a lock

  The method of acquiring a lock based on ApsaraDB for Redis Enhanced Edition (Tair) is the same as that based on native Redis databases. The method is to run the **SET** command:

  ```
  SET resource_1 random_value NX EX 5
  ```

- Release a lock

  The CAD command supported by ApsaraDB for Redis Enhanced Edition (Tair) provides an efficient way for you to release a lock:

  ```
  /* if (GET(resource_1) == my_random_value) DEL(resource_1) */
  CAD resource_1 my_random_value
  ```

- Renew a lock

  You can run the following CAS command to renew a lock:

  ```
  CAS resource_1 my_random_value my_random_value EX 10
  ```

  > **Note** The **CAS** command does not check whether the new value is the same as the original value.

Sample code based on Jedis

- Define the **CAS** and CAD commands

```
enum TairCommand implements ProtocolCommand {
CAD("CAD"), CAS("CAS");

private final byte[] raw;

TairCommand(String alt) {
raw = SafeEncoder.encode(alt);
}

@Override
public byte[] getRaw() {
return raw;
}
}
```

- Acquire a lock

```
public boolean acquireDistributedLock(Jedis jedis,String resourceKey, String randomValue, int expire
Time) {
SetParams setParams = new SetParams();
setParams.nx().ex(expireTime);
String result = jedis.set(resourceKey,randomValue,setParams);
return "OK".equals(result);
}
```

- Release a lock

```
public boolean releaseDistributedLock(Jedis jedis,String resourceKey, String randomValue) {
jedis.getClient().sendCommand(TairCommand.CAD,resourceKey,randomValue);
Long ret = jedis.getClient().getIntegerReply();
return 1 == ret;
}
```

- Renew a lock

```
public boolean renewDistributedLock(Jedis jedis,String resourceKey, String randomValue, int expire
Time) {
jedis.getClient().sendCommand(TairCommand.CAS,resourceKey,randomValue,randomValue,"EX",Stri
ng.valueOf(expireTime));
Long ret = jedis.getClient().getIntegerReply();
return 1 == ret;
}
```

Methods of ensuring lock consistency

The replications between masters and replicas are asynchronous. If a master crashes before data changes are transferred to replicas and a failover is triggered, the data changes in the buffer may not be replicated to the new master. This results in data inconsistency. Note that the failover is triggered to ensure high availability and the new master is the original replica. If the lost data is related to a distributed lock, the locking mechanism becomes faulty and service exceptions occur. This topic introduces three methods that you can use to ensure lock consistency.

- Redlock

  Redlock is proposed by the founders of the open source Redis project to ensure consistency. Redlock is based on the calculation of probabilities. Assume that a single master-replica Redis instance may lose a lock during a failover and the probability is k%. Note that the failover is triggered to implement high availability. If you use Redlock to implement distributed locks, you can calculate the probability at which N independent Redis instances lose locks at the same time based on the following formula: Probability of losing locks = (k%)^N. Due to the high stability of Redis, the probability is small, which can meet your service requirements.

  > **Note** When you implement Redlock, you do not have to ensure that all the locks in N Redis instances take effect at the same time. In most cases, Redlock can meet your business requirements if you ensure that the locks in M Redis nodes take effect at the same time. Note that M is greater than 1 and less than or equal to N.

  Redlock has the following disadvantages:

  - A client takes a long time to acquire and release a lock.
  - You must handle significant difficulties if you want to implement Redlock in the cluster or standard master-replica instances of ApsaraDB for Redis.
  - Redlock consumes a large number of resources. To implement Redlock, you must create multiple independent ApsaraDB for Redis instances or user-created Redis instances.

- WAIT command

  The **WAIT** command of Redis blocks the current client until all the previous write commands are transferred from a master to a specified number of replicas. In the WAIT command, you can specify a time-out period that is measured in milliseconds. The following **WAIT** command example is used in ApsaraDB for Redis to ensure the consistency of distributed locks.

  ```
  SET resource_1 random_value NX EX 5
  WAIT 1 5000
  ```

  If you run the WAIT command, a client continues to perform other operations only in two scenarios after the client acquires a lock. One of the scenarios is that data is transferred to the replicas. The other scenario is that the time-out period is reached. In this example, the time-out period is 5,000 milliseconds. If the output of the **WAIT** command is 1, data is replicated between the master and the replicas. In this case, data consistency is ensured. The WAIT command is more cost-effective than Redlock.

  The considerations of the WAIT command are described as follows:

- The **WAIT** command only blocks the client that sends the WAIT command, and does not affect the other clients.
- If the **WAIT** command returns a valid integer, the lock is transferred from the master to the replicas. However, if a failover is triggered to implement high availability before the command returns a successful response, data may be lost. In this case, the output of the **WAIT** command only indicates a possible replication failure, and data integrity cannot be ensured. After the WAIT command returns errors, you can acquire a lock again or verify data.
- You do not have to run the **WAIT** command to release a lock. This is because the distributed locks are mutually exclusive. Logical failures do not occur even if you release the lock after a certain period.

- ApsaraDB for Redis Enhanced Edition (Tair)

  Redlock and the WAIT command offer the following benefits:

  - Redlock improves data consistency if the number of Redis nodes increases.
  - The **WAIT** command is cost-effective.

  ApsaraDB for Redis Enhanced Edition (Tair) offers the following benefits:

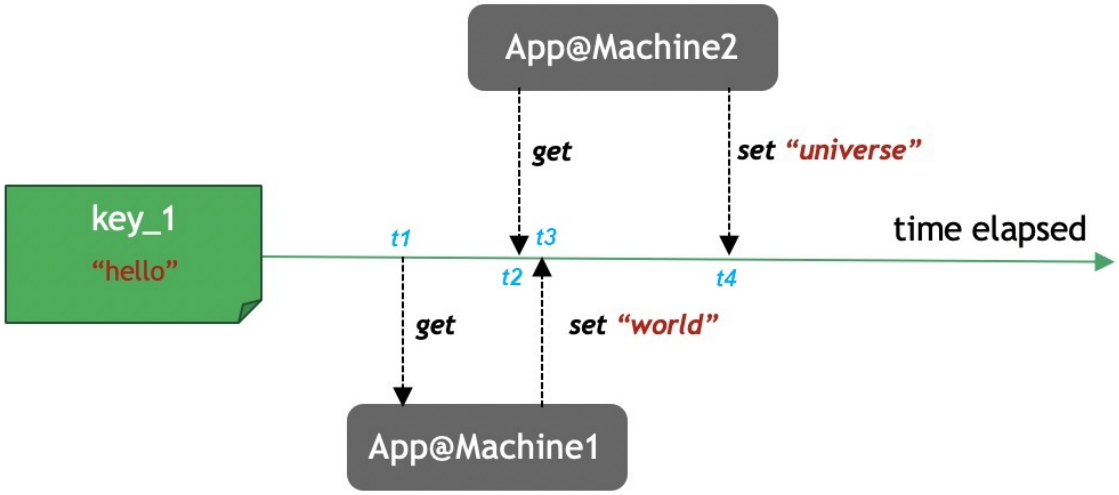
  - The unique high availability (HA) and data persistence mechanisms of ApsaraDB for Redis Enhanced Edition (Tair) help you ensure data security and service stability. ApsaraDB for Redis Enhanced Edition (Tair) allows you to ensure high data consistency even if you do not deploy multiple Redis nodes or use the **WAIT** command.
  - The **CAS** and CAD commands supported by the performance-enhanced instances help you reduce the costs of developing and managing distributed locks and improve lock performance.
  - Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) use a multi-threading model. For more information, see Enhanced multi-threading performance. The instances of this type provide the performance three times that provided by native Redis databases with the same specifications. Therefore, the ApsaraDB for Redis service remains available when highly concurrent distributed locks are used.

1.3. Concurrency control and optimistic locking

If a large number of requests are sent to concurrently access and update the shared resources stored in Redis, an accurate and efficient concurrency control mechanism is required. The mechanism must be able to help you prevent logical failures and data errors. One of the mechanism examples is optimistic locking. Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provides the TairString data structure. Therefore, compared with native Redis databases, ApsaraDB for Redis Enhanced Edition (Tair) allows you to implement optimistic locking that delivers higher performance at lower costs.

Concurrency and last-writer-wins

The following figure shows a typical scenario where concurrent requests cause race hazards.



1. At the initial stage, the value of key_1 is `hello`. The values of this key are strings.
2. At the t1 time point, application 1 reads the value of key_1: `hello`.
3. At the t2 time point, application 2 reads the value of key_1: `hello`.
4. At the t3 time point, application 1 changes the value of key_1 to `world`.
5. At the t4 time point, application 2 changes the value of key_1 to `universe`.

The value of key_1 is determined by the last write. At the t4 time point, application 1 considers the value of key_1 as world, but the actual value is universe. Therefore, the subsequent operations may become faulty. This process explains what is last-writer-wins. To resolve the issues that are caused by last-writer-wins, you must ensure the atomicity of the operations of accessing and updating string data. In other words, you must convert the string data of the shared resources into atomic variables. To do this, you can implement high-performance optimistic locking by using the TairString data structure. This data structure is offered by the performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair).

Implement optimistic locking based on TairString

TairString, also known as an extended string (exString), is a string data structure that carries a version number. Native Redis strings consist of only keys and values. TairString data consists of keys, values, and version numbers. Therefore, TairString is more suitable for optimistic locking. For more information about TairString commands, see TairString commands.

> **Note** The TairString data structure is different from the native Redis string data structure. Two sets of commands are provided for the two data structures. You can use only one set of the commands in a system.

TairString has the following features:

- A version number is provided for each key. The version number indicates the current version of a key. If you run the **EXSET** command to create a key, the default version number of the key is 1.
- If you run the **EXGET** command for a specified key, you can retrieve the values of two fields: value and version.
- Before you update a TairString value, the version is verified. If the verification fails, the following error message is returned: `ERR update version is stale`.
- After the TairString value is updated, the version number is automatically incremented by 1.
- TairString integrates all the features of native Redis strings except bit operations.

Due to these features, the lock mechanism is native to TairString data. Therefore, TairString provides an easy method for you to implement optimistic locking. An example is described as follows:

```
while(true){
{value, version} = EXGET(key); // Retrieve the value and the version of the key.
value2 = update(...) ; // Save a new value to the value2 field.
ret = EXSET(key, value2, version); // Update the key and assign the return value to the ret variable.
if(ret == OK)
break; // If the return value is OK, the update is successful and the while loop exits.
else if (ret.contanis("version is stale"))
continue; // If the return value contains the "version is stale" error message, the update fails and the
while loop is repeated.
}
```

> **Note**
>
> - If you delete TairString data and create new TairString data that has the same key as the deleted TairString data, the key version of the new TairString data is 1. The new TairString data does not inherit the key version of the deleted TairString data.
> - You can specify the ABS option to skip version verification and forcibly overwrite the current version to update TairString data. For more information, see EXSET.

Reduce resource consumption for optimistic locking

In the preceding sample code, if another client updates the shared resource after you run the **EXGET** command, you receive an update failure message and the while loop is repeated. The **EXGET** command is repeatedly run to retrieve the value and the version of the shared resource before the update is successful. As a result, two I/O operations are performed to access Redis in each while loop. However, by using the EXCAS command of TairString, you only need to send one access request in each while loop. This results in a significant decrease in the consumption of system resources and improves service performance in high concurrency scenarios.

When you run the **EXCAS** command, you can specify a version number in the command to verify the version. If the verification is successful, the TairString value is updated. If the verification fails, the following three elements are returned:

- `update version is stale`
- value
- version

If the update fails, the command returns the current version of the TairString data. You do not have to run another query to retrieve the current version, and only one access request is required for each while loop. An example is described as follows:

```
while(true){
{ret, value, version} = excas(key, new_value, old_version) // Use the CAS command to replace an origin
al value with a new value.
if(ret == OK)
break; // If the return value is OK, the update is successful and the while loop exits.
else (if ret.contanis("update version is stale")) // If the return value contains the "update version is st
ale" error message, the update fails. The values of the two variables are updated: value and old_versi
on.
update(value);
old_version = version;
}
```

1.4. Rate limiter

In flash sale scenarios where the purchase time or the quantity is limited, you have to handle traffic peaks that occur before and after the flash sales. You also have to prevent the accepted purchase orders from exceeding the upper limit. To handle these challenges, performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) offer the TairString data structure that provides a simple and efficient rate limiter. You can use the rate limiter to prevent the accepted purchase orders from exceeding the upper limit. The solutions described in this topic are also applicable to other scenarios where rate or traffic limiting and throttling are required.

Rate limiter for flash sales

Based on the integration with Alibaba Tair, performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provides the TairString data structure. TairString is more powerful than native Redis strings. TairString offers all the features of native Redis strings except bit operations.

The **EXINCRBY** and EXINCRBYFLOAT commands of TairString have similar functions to the **INCRBY** and INCRBYFLOAT commands of native Redis strings. You can use these commands to increment or decrement values. The **EXINCRBY** and EXINCRBYFLOAT commands support more options than the two commands of the native Redis strings. These options include *EX*, *NX*, *VER*, *MIN*, and *MAX*. For more information, see TairString commands. The solution described in this topic uses *MIN* and *MAX* options.

Option	Description
MIN	Specifies the minimum TairString value.
MAX	Specifies the maximum TairString value.

If you use native Redis strings to handle the challenges of flash sales, you have to implement complex code logic and handle management difficulties. This may easily cause excess purchase orders in flash sales. To be more specific, users receive prompts of successful purchases after all the items are sold out in flash sales, which causes negative effects. TairString allows you to compile and run simple code to limit the exact number of purchase orders. The pseudocode is described as follows:

```
if(EXINCRBY(key_iphone, -1, MIN:0) == "would overflow")
run_out();
```

Counters for rate limiting

Similar to Rate limiter for flash sales, you can specify the MAX option of the EXINCRBY command to implement counters for rate limiting. The pseudocode is described as follows:

```
if(EXINCRBY(rate_limitor, 1, MAX:1000) == "would overflow")
traffic_control();
```

Counters for rate limiting can be applied to various scenarios. For example, you can use the counters to limit the number of concurrent requests, the access frequency, and the number of password changes. For example, in concurrency limiting scenarios, the number of concurrent requests unexpectedly exceeds the system performance threshold. To prevent service crashes and serious issues, you can use a rate limiter to control the number of concurrent requests. This offers an appropriate temporary solution. This solution ensures that the system responds to the concurrent requests in a timely manner if the number of the concurrent requests does not exceed the performance threshold. The **EXINCRBY** command of TairString allows you to compile simple code to set a rate limiter for concurrent requests. The sample code is described as follows:

```
public boolean tryAcquire(Jedis jedis,String rateLimitor,int limiter){
try {
jedis.getClient().sendCommand(TairCommand.EXINCRBY,rateLimitor,"1","EX","1","MAX",String.valueOf(li
miter)); // Set a rate limiter.
jedis.getClient().getIntegerReply();
return true;
}catch (Exception e){
if(e.getMessage().contains("increment or decrement would overflow")){ // Check whether the result c
ontains error messages.
return false;
}
throw e;
}
}
```

1.5. TairHash memory consumption and expiration policies

Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provide the TairHash data structure. TairHash supports efficient and dynamic expiration policies, and these policies provide a quick method for you to release memory. However, the adoption of these policies increases the memory consumption of TairHash data. This topic describes the memory consumption and the two expiration policies of TairHash data. This helps you handle the tradeoff between data expiration efficiency and memory consumption, and reduce service costs.

Memory consumption comparison between native Redis hashes and TairHash data

If no expiration time is specified, the amount of memory consumed by native Redis hashes is similar to that consumed by TairHash data. The following tables provide testing details.

- Test 1

  Test environment

Item	Test command	Test condition
TairHash	EXHSET tairhashkey field value	◦ The size of the field is 1 KB. ◦ The size of the value is 1 KB. ◦ The append only file (AOF) and Redis database backup (RDB) persistence policies are disabled. ◦ The instance contains only one node. No replicas are available.
Native Redis hashes	HSET hashkey field value	

  Test results

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes
10000	29.79 MB	29.79 MB
100000	297.02 MB	297.02 MB
1000000	2.9 GB	2.9 GB

- Test 2

  Test environment

Item	Test command	Test condition
TairHash	EXHSET tairhashkey field value	◦ The size of the field is 64 bytes. ◦ The size of the value is 10 KB. ◦ The AOF and RDB persistence policies are disabled. ◦ The instance contains only one node. No replicas are available.
Native Redis hashes	HSET hashkey field value	

  Test results

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes
10000	104.17 MB	103.79 MB
100000	1.02 GB	1.02 GB
1000000	10.19 GB	10.19 GB

- Test 3

  Test environment

Item	Test command	Test condition
TairHash	`EXHSET tairhashkey field value`	◦ The size of the field is 64 bytes. ◦ The size of the value is 64 bytes. ◦ The AOF and RDB persistence policies are disabled. ◦ The instance contains only one node. No replicas are available.
Native Redis hashes	`HSET hashkey field value`	

  Test results

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes
10000	2.39 MB	2.02 MB
100000	25.33 MB	19.31 MB
1000000	253.29 MB	191.1 MB

The test results show that if you do not specify expiration policies, the amount of memory consumed by TairHash data is similar to or the same as that consumed by native Redis hashes. If you specify the expiration time for the fields of TairHash data, expiration policies affect the memory consumption of TairHash data.

Expiration policies supported by TairHash

Similar to native Redis hashes, TairHash supports two expiration policies: active expiration and passive expiration.

- Active expiration

  The active expiration policy of TairHash differs from that of native Redis hashes.

  - The native Redis service runs periodic tasks to select a key at random from the keys where the expiration time is specified and to check whether the key expires. If the key expires, the key is deleted. If the key does not expire, the key is retained. This results in an inefficient process.
  - Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) also run periodic tasks to check the TairHash fields where the expiration time is specified. If a field expires, the field is deleted to release the consumed memory. In addition, performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) use min heaps to sort the fields based on the expiration time of the fields. The system always selects the field that expires before the other fields expire. The active expiration policy of TairHash is more efficient than that of native Redis hashes, but requires additional memory. For more information, see Memory consumption caused by active expiration.

- Passive expiration

  The passive expiration policy of TairHash is similar to that of native Redis hashes. If a client accesses an expired field of TairHash data, the passive expiration policy is triggered and the expired field is deleted to release the consumed memory. The following example describes the passive expiration process of TairHash data.

  i. A field of TairHash data expired a short of period ago, and is not deleted in an active way. At this time, a client runs the **EXHGET** command to retrieve the field.
  ii. An ApsaraDB for Redis Enhanced Edition (Tair) instance determines whether the field expires.
  iii. The field has expired. Therefore, the instance deletes the field to release the consumed memory, and returns a null value to the client.

  The passive expiration policy requires no additional memory. However, if an expired field is not accessed by clients, the memory consumed by the expired field cannot be released.

For more information about the rules that determine when the two expiration policies take effect, see Rules that determine when the expiration policies take effect. In actual business scenarios, we recommend that you combine active expiration and passive expiration policies. For more information, see Best practices.

Rules that determine when the expiration policies take effect

The expiration policies of TairHash data take effect based on the following rules:

- By default, active expiration and passive expiration are enabled at the same time.
- Passive expiration takes effect for all the fields where the expiration time is specified in TairHash data.
- Active expiration takes effect for individual fields. If you run a command to specify the expiration time for a field of TairHash data, you can add the *NOACTIVE* option to disable active expiration for the field.

> **Note** For more information about TairHash commands, see TairHash commands.

If you do not add the *NOACTIVE* option when you specify the expiration time, active expiration takes effect and requires additional memory. For more information about the additional memory consumption that is caused by active expiration, see Memory consumption caused by active expiration.

Memory consumption caused by active expiration

Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) use min heaps to sort all the fields of TairHash data based on the expiration time of the fields. The instances also create indexes for the fields. After you specify the expiration time for a field of TairHash data, an ApsaraDB for Redis Enhanced Edition (Tair) instance creates a heap node and adds the heap node to a min heap. The heap node stores the field indexes that contain the key information. This allows you to find and delete the corresponding TairHash data and fields after the heap node expires. As a result, the memory that is consumed by the expired TairHash data and fields is released. However, the system cannot immediately release the memory consumed by the expired keys and fields after the command is run. This is because the heap node references the field indexes that contain the key information. Therefore, if you use active expiration, the following additional memory is required:

- The memory consumed by the heap node
- The memory consumed by the keys
- The memory consumed by the fields where the expiration time is specified

The following tables about comparison testing offer an intuitive display of the memory consumption of TairHash data. You can view the memory consumption of TairHash data in different test scenarios: active expiration and passive expiration.

Compare the memory consumption of TairHash in different test scenarios: active expiration and passive expiration

- Test 1

  Test environment

Item	Test command	Test condition

Item	Test command	Test condition
TairHash data where no expiration time is specified	`EXHSET tairhashkey field value`	◦ The size of the field is 1 KB. ◦ The size of the value is 1 KB. ◦ The AOF and RDB persistence policies are disabled. ◦ The instance contains only one node. No replicas are available.
TairHash data where the expiration time is specified	`EXHSET tairhashkey field value EX 1000`	

Test results

Number of fields	Memory consumed by the TairHash data where no expiration time is specified	Memory consumed by the TairHash data where the expiration time is specified
10000	29.79 MB	46.03 MB
100000	297.02 MB	460.36 MB
1000000	2.9 GB	4.6 GB

- **Test 2**

  Test environment

Item	Test command	Test condition
TairHash data where no expiration time is specified	`EXHSET tairhashkey field value`	◦ The size of the field is 64 bytes. ◦ The size of the value is 64 bytes. ◦ The AOF and RDB persistence policies are disabled. ◦ The instance contains only one node. No replicas are available.
TairHash data where the expiration time is specified	`EXHSET tairhashkey field value EX 1000`	

  Test results

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes
10000	2.39 MB	4.38 MB
100000	25.33 MB	45.16 MB
1000000	253.29 MB	451.66 MB

- **Test 3**

  Test environment

Item	Test command	Test condition
TairHash data where no expiration time is specified	`EXHSET tairhashkey field value`	◦ The size of the field is 64 bytes. ◦ The size of the value is 64 bytes. ◦ The AOF and RDB persistence policies are disabled. ◦ The instance contains only one node. No replicas are available. ◦ When a command is run to specify the expiration time, the *NOACTIVE* option is added to disable active expiration.
TairHash data where the expiration time is specified	`EXHSET tairhashkey field value EX 1000 NOACTIVE`	

Test results

Number of fields	Memory consumed by TairHash data	Memory consumed by native Redis hashes
10000	2.39 MB	2.39 MB
100000	25.33 MB	25.33 MB
1000000	253.29 MB	253.29 MB

Best practices

TairHash supports the *NOACTIVE* option for the following commands:

- EXHSET
- EXHEXPIRE
- EXHEXPIREAT
- EXHPEXPIRE
- EXHPEXPIREAT
- EXHINCRBY
- EXHINCRBYFLOAT

The Rules that determine when the expiration policies take effect section describes the rules that determine when the expiration policies take effect. Based on these rules, if you run the preceding commands to specify the expiration time for a field of TairHash data, you can add the *NOACTIVE* option to disable active expiration. If you do not add this option, active expiration is enabled. If you add the *NOACTIVE* option, active expiration does not apply to the target field and no additional memory is required. For more information, see Additional memory caused by active expiration. However, if active expiration is disabled and an expired field is not accessed by clients, the system cannot release the memory consumed by the expired field.

> **Notice** If you run the **EXHGETALL**, **EXHKEYS**, **EXHVALS**, or **EXHSCAN** command to access TairHash data, passive expiration is not triggered. As a result, the system does not delete the expired fields to release the memory that is consumed by these fields. However, the system filters out the expired fields from the outputs of these commands to provide a quick response for clients and reduce the risks of slow queries.

We recommend that you determine whether to use active expiration or on which data active expiration takes effect based on your business characteristics. The following examples help you determine whether to use and how to use active expiration.

- If the data validity time is short and the data volume increases at a fast speed, we recommend that you use active expiration. By using active expiration, the system can delete the expired data at the earliest opportunity to release the memory that is consumed by the expired data. The released memory can be used to store new data.
- If the data validity time is long and the data volume increases at a slow speed, we recommend that you do not use active expiration if possible. This reduces the total memory consumption.
- If cold data is clearly distinguished from hot data, you can apply active expiration on cold data to improve the memory release efficiency. However, if you write hot data, we recommend that you add the *NOACTIVE* option to disable active expiration. This helps you to improve the expiration efficiency and reduce the memory consumption.

2.Best Practices for All Editions

2.1. Migrate MySQL data to ApsaraDB for Redis

You can efficiently migrate data from RDS MySQL or on-premises MySQL databases to ApsaraDB for Redis by using the pipeline feature of ApsaraDB for Redis. You can also migrate data from RDS databases that use other engines to ApsaraDB for Redis based on the steps described in this topic.

Scenarios

You can use ApsaraDB for Redis as a cache between your applications and databases to expand the service capabilities of traditional relational databases. In this way, you can optimize the business ecosystem. This is one of the classic application scenarios of ApsaraDB for Redis. This service stores hot data in the business. You can easily obtain common data in ApsaraDB for Redis from your applications, or use ApsaraDB for Redis to save sessions of active users in interactive applications. This service can greatly reduce the load on the backend relational database and improve the user experience.

To use ApsaraDB for Redis as a cache, you must first transmit data from a relational database to ApsaraDB for Redis. You cannot directly transmit tables in a relational database to the ApsaraDB for Redis database that stores data in a key-value structure. Before the migration, you must convert the source data to a special structure. This topic describes how to use the native tool to easily and efficiently migrate tables from MySQL databases to ApsaraDB for Redis. You can use the pipeline feature of ApsaraDB for Redis to transmit data in MySQL tables to hash tables of ApsaraDB for Redis.

> **Note** This topic describes Alibaba Cloud RDS MySQL instance as the migration source and ApsaraDB for Redis instance as the migration destination. In this example, you install the Linux environment that runs the migration command on the ECS instance. These instances run in the same VPC, so they can interconnect with each other.

Similarly, you can migrate data from other relational databases to ApsaraDB for Redis. In this migration, you need to extract data from the source database, convert the data format, and then transmit the data to the heterogeneous database. This migration method is also suitable for data migration between other heterogeneous databases.

Prerequisites

- You have created an RDS MySQL instance as the source where table data is available for migration.
- You have created an ApsaraDB for Redis instance as the destination.
- You have created an ECS instance that runs the Linux system.
- These instances run in the same VPC of the same region.
- You have added the internal IP address of the ECS instance to the whitelists of RDS MySQL and ApsaraDB for Redis instances.
- You have installed MySQL and Redis databases on the ECS instance to extract, convert, and transmit data.

> **Note** These prerequisites apply only when you migrate data in Alibaba Cloud. If you want to migrate data in your on-premises environment, make sure that the Linux server that performs migration can connect to the source relational database and the destination ApsaraDB for Redis database.

Data before migration

This topic describes how to migrate the test data stored in the **company** table of the **custm_info** database. The **company** table contains test data as shown in the following table.

```
MySQL [custm_info]> SELECT * FROM company;
+----------------------------------+----------------------------------+------------+----------------------------------+----------------------------------+----------------------+
| id                               | name                             | sdate      | email                            | domain                           | city                 |
+----------------------------------+----------------------------------+------------+----------------------------------+----------------------------------+----------------------+
| [redacted]d96b5                  | [redacted]unter                  | 1986-05-23 | [redacted].com                   | [redacted]@example.net           | Michaelborough       |
| [redacted]662a7                  | [redacted]                       | 1970-12-11 | [redacted]                       | [redacted]@example.com           | Pamelaborough        |
| [redacted]db6c1                  | [redacted]                       | 2001-09-06 | [redacted]                       | [redacted]example.net            | Port Melodybury      |
| [redacted]38c2a                  | [redacted]                       | 1979-06-08 | [redacted]th.org                 | [redacted]ple.org                | New Tracymouth       |
| [redacted]65613                  | [redacted]Hernandez              | 1975-11-01 | [redacted]                       | [redacted]@example.net           | North Matthewhaven   |
| [redacted]bb993                  | [redacted]d Wagner               | 2004-06-29 | [redacted]                       | [redacted]le.net                 | East Angelamouth     |
| [redacted]132b1                  | [redacted]                       | 2018-01-03 | [redacted]mith.com               | [redacted]ple.org                | Bradleychester       |
| [redacted]0898c                  | [redacted]                       | 1971-12-07 | [redacted]                       | [redacted]e.org                  | East Christianhaven  |
| [redacted]a5882                  | [redacted]                       | 1976-07-14 | [redacted]                       | [redacted]e.org                  | Port Christopherberg |
```

This table includes six columns. After the migration, the value of the id column in the MySQL table changes to the key of the hash table in ApsaraDB for Redis. The column names of other columns change to the fields of the hash table, and the values of these columns change to the values of the corresponding fields. You can modify the scripts and commands for the migration according to actual scenarios.

Procedure

1. Analyze the source data structure, create the following migration script on the ECS instance, and then save the script to the mysql_to_redis.sql file.

```
SELECT CONCAT(
"*12\r\n", #The number 12 specifies the number of the following fields, and depends on the data s
tructure of the MySQL table.
'$', LENGTH('HMSET'), '\r\n', #HMSET specifies the command that you use when writing data to Aps
araDB for Redis.
'HMSET', '\r\n',
'$', LENGTH(id), '\r\n', #id specifies the first field after you run the HMSET command for fields. This
field changes to the key of the hash table in ApsaraDB for Redis.
id, '\r\n',
'$', LENGTH('name'), '\r\n', #'name' is passed to the hash table as a string field. Other fields such a
s 'sdate' are processed in the same way.
'name', '\r\n',
'$', LENGTH(name), '\r\n', #The name variable specifies the company name in the MySQL table. Thi
s variable changes to the value of the field generated by the 'name' parameter. Other fields such
as 'sdate' are processed in the same way.
name, '\r\n',
'$', LENGTH('sdate'), '\r\n',
'sdate', '\r\n',
'$', LENGTH(sdate), '\r\n',
sdate, '\r\n',
'$', LENGTH('email'), '\r\n',
'email', '\r\n',
'$', LENGTH(email), '\r\n',
email, '\r\n',
'$', LENGTH('domain'), '\r\n',
'domain', '\r\n',
'$', LENGTH(domain), '\r\n',
domain, '\r\n',
'$', LENGTH('city'), '\r\n',
'city', '\r\n',
'$', LENGTH(city), '\r\n',
city, '\r\n'
)
FROM company AS c
```

2. Run the following command on the ECS instance to migrate data.

```
mysql -h <MySQL host> -P <MySQL port> -u <MySQL username> -D <MySQL database name> -p --s
kip-column-names --raw < mysql_to_redis.sql | redis-cli -h <Redis host> --pipe -a <Redis password
>
```

Options

Option	Description	Example
-h	The endpoint of the RDS MySQL database. Note This is the *-h* option following `mysql` .	rm-bp1xxxxxxxxxxx.mysql.rds.aliyuncs.com Note Use the endpoint for connecting the Linux server to the RDS MySQL database.
-P	The service port of the RDS MySQL database.	3306
-u	The username of the RDS MySQL database.	testuser
-D	The database where the MySQL table that you want to migrate is located.	mydatabase
-p	The password for connecting to the RDS MySQL database. Note ◦ If you do not have any password, you do not need to set this option. ◦ To improve security, you can enter *-p* and do not have the password following this option. You can run the command and then enter the password according to the command-line interface (CLI) prompt.	Mysqlpwd233
--skip-column-names	The column name is not written into the query result.	No value is required.
--raw	The output column value is not escaped.	No value is required.
-h	Specifies the endpoint of ApsaraDB for Redis. Note This is the *-h* option following `redis-cli` .	r-bp1xxxxxxxxxxx.redis.rds.aliyuncs.com Note Use the endpoint for connecting the Linux server to the ApsaraDB for Redis database.

Option	Description	Example
--pipe	Use the pipeline feature of ApsaraDB for Redis to transmit data.	No value is required.
-a	The password for connecting to ApsaraDB for Redis. **Note** If you do not have any password or do not need a password, you do not need to set this option.	Redispwd233

Sample code

```
[root@ ~]# mysql -h rm-bp1 .mysql.rds.aliyuncs.com -P 3306 -u  -D custm_info -
p --skip-column-names --raw < mysql_to_redis.sql | redis-cli -h r-bp1 .redis.rds.aliyuncs.com --pipe -a
Enter password:
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 100
```

Note In the result, `errors` indicates the number of errors that occur when the system runs the command, and `replies` indicates the number of responses the system returns. If the value of `errors` is 0 and the value of `replies` equals the number of items in the MySQL table, the migration is completed.

Data after migration

After the migration, one item in the MySQL table corresponds to one item in the hash table of ApsaraDB for Redis. You can run the **HGETALL** command to query an item and view the following result.

```
r-bp1 .redis.rds.aliyuncs.com:6379> HGETALL 6b132b1
 1) "name"
 2) " ons"
 3) "sdate"
 4) "2018-01-03"
 5) "email"
 6) " @ smith.com"
 7) "domain"
 8) " @example.org"
 9) "city"
10) "Bradleychester"
```

You can adjust the migration solution based on the query method required in actual scenarios. For example, you can convert other columns in the MySQL table to the keys in the hash table and convert the id column to a field, or ignore the id column.

2.2. Online player score ranking

Compared with Redis, the features of ApsaraDB for Redis are similar. You can use ApsaraDB for Redis to create a ranking list for an online game.

Sample code

```
import java.util.ArrayList;
```

```
import java.util.List;
import java.util.Set;
import java.util.UUID;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Tuple;
public class GameRankSample {
static int TOTAL_SIZE = 20;
public static void main(String[] args)
{
//Connection information. This information can be obtained from the console
String host = "xxxxxxxxxx.m.cnhz1.kvstore.aliyuncs.com";
int port = 6379;
Jedis jedis = new Jedis(host, port);
try {
//Instance password
String authString = jedis.auth("password");//password
if (! authString.equals("OK"))
{
System.err.println("AUTH Failed: " + authString);
return;
}
//Key
String key = "Game name: Keep Running, Ali!" ; ;
//Clears any existing data
jedis.del(key);
//Generates several simulated players
List<String> playerList = new ArrayList<String>();
for (int i = 0; i < TOTAL_SIZE; ++i) 
{
//Randomly generates an ID for each player
playerList.add(UUID.randomUUID().toString());
}
System.out.println("Inputs all players ");
//Records the score for each player
for (int i = 0; i < playerList.size(); i++) 
{
//Randomly generates numbers as the scores of each simulated player
int score = (int)(Math.random()*5000); 
String member = playerList.get(i);
System.out.println("Player ID:" + member + ", Player Score: " + score);
//Adds the player ID and score to SortedSet of the corresponding key
```

```
jedis.zadd(key, score, member);
}
//Prints out the ranking list of all players
System. out. println ();
System.out.println("       "+key);
System.out.println(" Ranking list of all players");
//Obtains the sorted list of players from SortedSet of the corresponding key
Set<Tuple> scoreList = jedis.zrevrangeWithScores(key, 0, -1);
for (Tuple item : scoreList) {
System.out.println("Player ID:"+item.getElement()+", Player Score:"+Double.valueOf(item.getScore()).i
ntValue());
}
//Prints out the top five players
System. out. println ();
System.out.println("       "+key);
System.out.println("       Top players");
scoreList = jedis.zrevrangeWithScores(key, 0, 4);
for (Tuple item : scoreList) {
System.out.println("Player ID:"+item.getElement()+", Player Score:"+Double.valueOf(item.getScore()).i
ntValue());
}
//Prints out a list of specific players
System. out. println ();
System.out.println("         "+key);
System.out.println(" Players with scores from 1,000 to 2,000");
//Obtains a list of players with scores from 1,000 to 2,000 from SortedSet of the corresponding key
scoreList = jedis.zrangeByScoreWithScores(key, 1000, 2000);
for (Tuple item : scoreList) {
System.out.println("Player ID:"+item.getElement()+", Player Score:"+Double.valueOf(item.getScore()).i
ntValue()); 
}
} catch (Exception e) {
e.printStackTrace();
}finally{
jedis.quit();
jedis.close();
}
}
}
```

Results

After you access the ApsaraDB for Redis instance with the correct address and password and run the Java code, the following output is displayed:

```
Input all players
Player ID: 9193e26f-6a71-4c76-8666-eaf8ee97ac86. Score: 3,860
Player ID: db03520b-75a3-48e5-850a-071722ff7afb. Score: 4,853
Player ID: d302d24d-d380-4e15-a4d6-84f71313f27a. Score: 2,931
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6. Score: 1,796
Player ID: ec24fb9e-366e-4b89-a0d5-0be151a8cad0. Score: 2,263
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa. Score: 1,848
Player ID: 4c396f67-da7c-4b99-a783-25919d52d756. Score: 958
Player ID: a6299dd2-4f38-4528-bb5a-aa2d48a9f94a. Score: 2,428
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65. Score: 4,478
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7. Score: 1,655
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1. Score: 4,064
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e. Score: 4,852
Player ID: 19e2aa6b-a2d8-4e56-bdf7-8b59f64bd8e0. Score: 3,394
Player ID: cb62bb24-1318-4af2-9d9b-fbff7280dbec. Score: 3,405
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968. Score: 4,391
Player ID: 2c814a6f-3706-4280-9085-5fe5fd56b71c. Score: 2,510
Player ID: 9ee2ed6d-08b8-4e7f-b52c-9adfe1e32dda. Score: 63
Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd. Score: 1,008
Player ID: 674bbdd1-2023-46ae-bbe6-dfcd8e372430. Score: 2,265
Player ID: 34574e3e-9cc5-43ed-ba15-9f5405312692. Score: 3,734
Game name: Keep Running, Ali!
Ranking list of all players
Player ID: db03520b-75a3-48e5-850a-071722ff7afb. Score: 4,853
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e. Score: 4,852
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65. Score: 4,478
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968. Score: 4,391
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1. Score: 4,064
Player ID: 9193e26f-6a71-4c76-8666-eaf8ee97ac86. Score: 3,860
Player ID: 34574e3e-9cc5-43ed-ba15-9f5405312692. Score: 3,734
Player ID: cb62bb24-1318-4af2-9d9b-fbff7280dbec. Score: 3,405
Player ID: 19e2aa6b-a2d8-4e56-bdf7-8b59f64bd8e0. Score: 3,394
Player ID: d302d24d-d380-4e15-a4d6-84f71313f27a. Score: 2,931
Player ID: 2c814a6f-3706-4280-9085-5fe5fd56b71c. Score: 2,510
Player ID: a6299dd2-4f38-4528-bb5a-aa2d48a9f94a. Score: 2,428
Player ID: 674bbdd1-2023-46ae-bbe6-dfcd8e372430. Score: 2,265
Player ID: ec24fb9e-366e-4b89-a0d5-0be151a8cad0. Score: 2,263
```

```
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa. Score: 1,848
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6. Score: 1,796
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7. Score: 1,655
Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd. Score: 1,008
Player ID: 4c396f67-da7c-4b99-a783-25919d52d756. Score: 958
Player ID:9ee2ed6d-08b8-4e7f-b52c-9adfe1e32dda. Score:63
Game name: Keep Running, Ali!
Top players
Player ID: db03520b-75a3-48e5-850a-071722ff7afb. Score: 4,853
Player ID: 99bc5b4f-e32a-4295-bc3a-0324887bb77e. Score: 4,852
Player ID: 2e4ec631-1e4e-4ef0-914f-7bf1745f7d65. Score: 4,478
Player ID: ec0f06da-91ee-447b-b935-7ca935dc7968. Score: 4,391
Player ID: e3e8e1fa-6aac-4a0c-af80-4c4a1e126cd1. Score: 4,064
Game name: Keep Running, Ali!
Players scored between 1,000 and 2,000
Player ID: 0293b43a-1554-4157-a95b-b78de9edf6dd. Score: 1,008
Player ID: 24235a85-85b9-476e-8b96-39f294f57aa7. Score:1,655
Player ID: bee46f9d-4b05-425e-8451-8aa6d48858e6. Score: 1,796
Player ID: e11ecc2c-cd51-4339-8412-c711142ca7aa. Score: 1,848
```

2.3. Correlation analysis on E-commerce store items

You can use ApsaraDB for Redis to perform a correlation analysis on E-commerce store items.

Scenario introduction

The correlation between items is the case where multiple items are added to the same shopping cart. The analysis results are crucial for the E-commerce industry and can be used to analyze shopping behaviors. For example:

- On the details page of a specific item, recommend related items to the user who is browsing this page.
- Recommend related items to a user who just added an item to the shopping cart.
- Place highly correlated items together on the shelf.

You can use ApsaraDB for Redis to create a sorted set for each item. For a specific item, the set consists of items that are added with this item to the shopping cart. Members of the set are scored based on how often they appear in the same cart with that specific item. Each time item A and item B appear in the same shopping cart, the respective sorted sets for item A and item B in ApsaraDB for Redis are updated.

Sample code

```
package shop.kvstore.aliyun.com;
import java.util.Set;
```

```
import java.util.Set;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Tuple;
public class AliyunShoppingMall {
public static void main(String[] args)
{
//ApsaraDB for Redis connection. This information can be obtained from the console
String host = "xxxxxxxx.m.cnhza.kvstore.aliyuncs.com";
int port = 6379;
Jedis jedis = new Jedis(host, port);
try {
//ApsaraDB for Redis instance password
String authString = jedis.auth("password");//password
if (! authString.equals("OK"))
{
System.err.println("AUTH Failed: " + authString);
return;
}
//Products
String key0="Alibaba Cloud: Product: Beer";
String key1 = "Alibaba Cloud: Product: Chocolate";
String key2 = "Alibaba Cloud: Product: Cola";
String key3 = "Alibaba Cloud: Product: Gum";
String key4 = "Alibaba Cloud: Product: Beef Jerky";
String key5="Alibaba Cloud: Product: Chicken Wings";
final String[] aliyunProducts=new String[]{key0,key1,key2,key3,key4,key5};
//Initialize to clear the possible existing data
for (int i = 0; i < aliyunProducts.length; i++) {
jedis.del(aliyunProducts[i]);
}
//Simulated shopping behaviors
for (int i = 0; i < 5; i++) { //Simulates the shopping behaviors of multiple customers
customersShopping(aliyunProducts,i,jedis);
}
System.out.println();
//Uses ApsaraDB for Redis to generate the correlated relationship between items
for (int i = 0; i < aliyunProducts.length; i++) {
System.out.println(">>>>>>>>>>and"+aliyunProducts[i]+"was purchased with <<<<<<<<<<<<<");
Set<Tuple> relatedList = jedis.zrevrangeWithScores(aliyunProducts[i], 0, -1);
for (Tuple item : relatedList) {
System.out.println("Item name:"+item.getElement()+", Purchased together times:"+Double.valueOf(ite
```

```
m.getScore()).intValue());
}
System.out.println();
}
} catch (Exception e) {
e.printStackTrace();
}finally{
jedis.quit();
jedis.close();
}
}
private static void customersShopping(String[] products, int i, Jedis jedis) {
//Simulates three simple shopping behaviors and randomly selects one as the behavior of the user
int bought=(int)(Math.random()*3);
if(bought==1){
//Simulated business logic: the user has purchased the following products:
System.out.println("User"+i+"purchased"+products[0]+","+products[2]+","+products[1]);
//Records the correlations between the items to SortSet in ApsaraDB for Redis
jedis.zincrby(products[0], 1, products[1]);
jedis.zincrby(products[0], 1, products[2]);
jedis.zincrby(products[1], 1, products[0]);
jedis.zincrby(products[1], 1, products[2]);
jedis.zincrby(products[2], 1, products[0]);
jedis.zincrby(products[2], 1, products[1]);
}else if(bought==2){
//Simulated business logic: the user has purchased the following products
System. out. println ("user" + i + "purchased" + products [4] + ", "+ products [2] +", "+ products [3]);
//Records the correlations between the items to SortSet in ApsaraDB for Redis
jedis.zincrby(products[4], 1, products[2]);
jedis.zincrby(products[4], 1, products[3]);
jedis.zincrby(products[3], 1, products[4]);
jedis.zincrby(products[3], 1, products[2]);
jedis.zincrby(products[2], 1, products[4]);
jedis.zincrby(products[2], 1, products[3]);
}else if(bought==0){
//Simulated business logic: the user has purchased the following products:
System.out.println("user"+i+"purchased"+products[1]+","+products[5]);
//Records the correlations between the items to SortSet in ApsaraDB for Redis
jedis.zincrby(products[5], 1, products[1]);
jedis.zincrby(products[1], 1, products[5]);
}
```

```
}
}
```

Results

After you access the ApsaraDB for Redis instance with the correct address and password and run the Java code, the following output is displayed:

```
User 0 purchased Alibaba Cloud: Product: Chocolate, Alibaba Cloud: Product: Chicken Wings
User 1 purchased Alibaba Cloud: Product: Beef Jerky, Alibaba Cloud: Product: Cola, Alibaba Cloud: Pro
duct: Gum
User 2 purchased Alibaba Cloud: Product: Beer, Alibaba Cloud: Product: Cola, Alibaba Cloud: product:
Chocolate
User 3 purchased Alibaba Cloud: Product: Beef Jerky, Alibaba Cloud: Product: Cola, Alibaba Cloud: Pro
duct: Gum
User 4 purchased Alibaba Cloud: Product: Chocolate, Alibaba Cloud: Product: Chicken Wings
>>>>>>>>>>Alibaba Cloud: Product: Beer was purchased with<<<<<<<<<<<<<<<
Item Name: Alibaba Cloud: Product: Chocolate. Purchased together times: 1
Item name: Alibaba Cloud: Product:Cola. Purchased together times: 1
>>>>>>>>>>Alibaba Cloud: Product: Chocolate was purchased with<<<<<<<<<<<<<<<<<<<<
Item name: Alibaba Cloud: Product: Chicken Wings. Purchased together times: 2
Item name: Alibaba Cloud: Product: Beer. Purchased together times: 1
Item name: Alibaba Cloud: Product: Cola. Purchased together times: 1
>>>>>>>>>>Alibaba Cloud: Product: Cola was purchased with<<<<<<<<<<<<<<<<<
Item name: Alibaba Cloud:Product: Beef Jerky. Purchased together times: 2
Item name: Alibaba Cloud: Product: Gum. Purchased together times: 2
Item name: Alibaba Cloud: Product: Chocolate. Purchased together times: 1
Item name: Alibaba Cloud: Product: Beer. Purchased together times: 1
>>>>>>>>>>Alibaba Cloud: Product: Gum was purchased with<<<<<<<<<<<<<<<<<<<<
Item name: Alibaba Cloud: Product: Beef Jerky. Purchased together times: 2
Item name: Alibaba Cloud: Product: Cola. Purchased together times: 2
>>>>>>>>>>Alibaba Cloud: Product: Beef Jerky was purchased with<<<<<<<<<<<<<<<
Item name: Alibaba Cloud: Product: Cola. Purchased together times: 2
Item name: Alibaba Cloud: Product: Gum. Purchased together times: 2
>>>>>>>>>>Alibaba Cloud: Product: Chicken Wings was purchased with<<<<<<<<<<<<<<
Item name: Alibaba Cloud: Product: Chocolate. Purchased together times: 2
```

2.4. Publish and subscribe to messages

Similar to Redis, ApsaraDB for Redis provides publishing (pub) and subscription (sub) features. ApsaraDB for Redis allows multiple clients to subscribe to messages published by a client.

Scenario introduction

It must be noted that messages published using ApsaraDB for Redis are "non-persistent". This means the message publisher is only responsible for publishing a message and does not save previously sent messages, whether or not these messages were received. Thus, messages are "lost once published". Message subscribers can only receive messages that are subscribed. They will not receive the earlier messages in the channel.

In addition, the message publisher (publish client) does not necessarily connect to a server exclusively. While publishing messages, you can perform other operations (for example, the List operations) from the same client at the same time. However, the message subscriber (subscribe client) needs to connect to a server exclusively. That is, during the subscription period, the client may not perform any other operations. Rather, the operations are blocked while the client is waiting for messages in the channel. Therefore, message subscribers must use a dedicated server connection or thread (see the following example).

Sample code

For the message publisher (publish client)

```
package message.kvstore.aliyun.com;
import redis.clients.jedis.Jedis;
public class KVStorePubClient {
private Jedis jedis;
public KVStorePubClient(String host,int port, String password){
jedis = new Jedis(host,port);
//KVStore instance password
String authString = jedis.auth(password);
if (! authString.equals("OK"))
{
System.err.println("AUTH Failed: " + authString);
return;
}
}
public void pub(String channel,String message){
System. out. println ("> Publish> channel: "+ channel +"> message sent: "+ message );
jedis.publish(channel, message);
}
public void close(String channel){
System.out.println(" >>> PUBLISH End > Channel:"+channel+" > Message:quit");
//The message publisher stops sending by sending a “quit” message
jedis.publish(channel, "quit");
}
}
```

For the message subscriber (subscribe client)

```
package message.kvstore.aliyun.com;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPubSub;
public class KVStoreSubClient extends Thread{
private Jedis jedis;
private String channel;
private JedisPubSub listener;
public KVStoreSubClient(String host,int port, String password){
jedis = new Jedis(host,port);
//ApsaraDB for Redis instance password
String authString = jedis.auth(password); //password
if (! authString.equals("OK"))
{
System.err.println("AUTH Failed: " + authString);
return;
}
}
public void setChannelAndListener(JedisPubSub listener,String channel){
this.listener=listener;
this.channel=channel;
}
private void subscribe(){
if(listener==null || channel==null){
System.err.println("Error:SubClient> listener or channel is null");
}
System.out.println(" >>> SUBSCRIBE > Channel:"+channel);
System.out.println();
//When the recipient is listening for subscribed messages, the process is blocked until the quit messa
ge is received (passively) or the subscription is canceled actively
jedis.subscribe(listener, channel);
}
public void unsubscribe(String channel){
System.out.println(" >>> UNSUBSCRIBE > Channel:"+channel);
System.out.println();
listener.unsubscribe(channel);
}
@Override
public void run() {
try{
```

```
System.out.println();
System.out.println("---------Subscription begins-------");
subscribe();
System.out.println("----------Subscription ends-------");
System.out.println();
}catch(Exception e){
e.printStackTrace();
}
}
}
```

For the message listener

```
package message.kvstore.aliyun.com;
import redis.clients.jedis.JedisPubSub;
public class KVStoreMessageListener extends JedisPubSub{
@Override
public void onMessage(String channel, String message) {
System.out.println(" <<< SUBSCRIBE< Channel:" + channel + " >Message received:" + message );
System.out.println();
//When a quit message is received, the subscription is canceled (passively)
if(message.equalsIgnoreCase("quit")){
this.unsubscribe(channel);
}
}
@Override
public void onPMessage(String pattern, String channel, String message) {
// TODO Auto-generated method stub
}
@Override
public void onSubscribe(String channel, int subscribedChannels) {
// TODO Auto-generated method stub
}
@Override
public void onUnsubscribe(String channel, int subscribedChannels) {
// TODO Auto-generated method stub
}
@Override
public void onPUnsubscribe(String pattern, int subscribedChannels) {
// TODO Auto-generated method stub
}
@Override
public void onPSubscribe(String pattern, int subscribedChannels) {
// TODO Auto-generated method stub
}
}
```

Sample main process

```
package message.kvstore.aliyun.com;
import java.util.UUID;
import redis.clients.jedis.JedisPubSub;
public class KVStorePubSubTest {
//The connection information of ApsaraDB for Redis. This information can be obtained from the consol
e
static final String host = "xxxxxxxxxx.m.cnhza.kvstore.aliyuncs.com";
static final int port = 6379;
static final String password="password";//password
public static void main(String[] args) throws Exception{
KVStorePubClient pubClient = new KVStorePubClient(host, port,password);
final String channel = "KVStore Channel-A";
//The message sender starts sending messages, but there are no subscribers, so the messages will n
ot be received
pubClient.pub(channel, "Alibaba Cloud message 1: (No subscribers. This message will not be received)
");
// Message recipient
KVStoreSubClient subClient = new KVStoreSubClient(host, port,password);
JedisPubSub listener = new KVStoreMessageListener();
subClient.setChannelAndListener(listener, channel);
// Message recipient starts subscribing
subClient.start();
//The message sender continues sending messages
for (int i = 0; i < 5; i++) {
String message=UUID.randomUUID().toString();
pubClient.pub(channel, message);
Thread.sleep(1000);
}
// The message recipient cancels the subscription
subClient.unsubscribe(channel);
Thread.sleep(1000);
pubClient.pub(channel, "Alibaba Cloud message 2:(Subscription canceled. This message will not be rec
eived)");
//The message publisher stops sending by sending a “quit” message
//When other message recipients, if any, receive "quit" in listener.onMessage(), the "unsubscribe" ope
ration is performed.
pubClient.close(channel);
}
}
```

Output

After you access the ApsaraDB for Redis instance with the correct address and password and run the preceding Java code, the following output is displayed.

```
>>> PUBLISH > Channel:KVStore Channel-A > Sends the message Aliyun Message 1: (No subscribers. T
his message will not be received)
----------Subscription starts-------
>>> SUBSCRIBE> Channel: KVStore Channel-A
>>> PUBLISH> Channel: KVStore Channel-A> sends message: 0f9c2cee-77c7-4498-89a0-1dc5a2f65889
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: 0f9c2cee-77c7-4498-89a0-1dc5a2f658
89
>>> PUBLISH> Channel: KVStore Channel-A> sends message: ed5924a9-016b-469b-8203-7db63d06f812
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: ed5924a9-016b-469b-8203-7db63d06f
812
>>> PUBLISH> Channel: KVStore Channel-A> sends message: f1f84e0f-8f35-4362-9567-25716b1531cd
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: f1f84e0f-8f35-4362-9567-25716b1531
cd
>>> PUBLISH> Channel: KVStore Channel-A> sends message: 746bde54-af8f-44d7-8a49-37d1a245d21b
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: 746bde54-af8f-44d7-8a49-37d1a245d
21b
>>> PUBLISH> Channel: KVStore Channel-A> sends message: 8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: 8ac3b2b8-9906-4f61-8cad-84fc1f15a3
ef
>>> UNSUBSCRIBE> Channel: KVStore Channel-A
----------Subscription ends-------
>>> PUBLISH > Channel:KVStore Channel-A > sends the message Aliyun Message 2: (The subscription
has been canceled, so the message will not be received)
>>> PUBLISH ends> Channel:KVStore Channel-A > Message:quit
```

The preceding example demonstrates a situation with one publisher and one subscriber. There can be multiple publishers, subscribers, and even multiple message channels. In such scenarios, you are required to slightly change the code to fit the scenario.

2.5. Pipeline

Similar to Redis, ApsaraDB for Redis provides the pipeline feature.

Scenario introduction

A client interacts with a server through one-way pipelines, one for sending requests and the other for receiving responses. You can send operation requests consecutively from the client to the server. However, during this period, the server does not send the responses to each operation request. The client receives the response to each request from the server until it sends a quit message to the server.

Pipelines are useful, for example, when several operation commands need to be quickly submitted to the server but the responses and operation results are not required immediately. In this case, pipelines are used as a batch processing tool to optimize the performance. The performance is enhanced because the overhead of the TCP connection is reduced.

However, the client using pipelines in the app connects to the server exclusively, and non-pipeline operations are blocked until the pipelines are closed. If you need to perform other operations at the same time, you can establish a dedicated connection for pipeline operations to separate them from conventional operations.

Sample code 1

Performance comparison

```
package pipeline.kvstore.aliyun.com;
import java.util.Date;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Pipeline;
public class RedisPipelinePerformanceTest {
static final String host = "xxxxxx.m.cnhza.kvstore.aliyuncs.com";
static final int port = 6379;
static final String password = "password";
public static void main(String[] args) {
Jedis jedis = new Jedis(host, port);
//ApsaraDB for Redis instance password
String authString = jedis.auth(password); // password
if (! authString.equals("OK")) {
System.err.println("AUTH Failed: " + authString);
jedis.close();
return;
}
//Executes several commands consecutively
final int COUNT=5000;
String key = "KVStore-Tanghan";
// 1 ---Without using pipeline operations---
jedis.del(key);//Initializes the key
Date ts1 = new Date();
for (int i = 0; i < COUNT; i++) {
//Sends a request and receives the response
```

```
jedis.incr(key);
}
Date ts2 = new Date();
System.out.println("Without Pipeline > value is:"+jedis.get(key)+" > Time elapsed:" + (ts2.getTime() - t
s1.getTime())+ "ms");
//2 ---Using pipeline operations---
jedis.del(key);//Initializes the key
Pipeline p1 = jedis.pipelined();
Date ts3 = new Date();
for (int i = 0; i < COUNT; i++) {
//Sends the request
p1.incr(key);
}
// Receives the response
p1.sync();
Date ts4 = new Date();
System.out.println("Using Pipeline > value is:"+jedis.get(key)+" > Time elapsed:" + (ts4.getTime() - ts3
.getTime())+ "ms");
jedis.close();
}
}
```

Output 1

After you access the ApsaraDB for Redis instance with the correct address and password and run the preceding Java code, the following output is displayed. The output shows that the performance is enhanced with pipelines.

```
Without pipelines > value: 5000 > Time elapsed: 5844 ms
With pipelines > value: 5000 > Time elapsed: 78 ms
```

Sample code 2

With pipelines defined in Jedis, responses are processed in two methods, as shown in the following sample code:

```
package pipeline.kvstore.aliyun.com;
import java.util.List;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Pipeline;
import redis.clients.jedis.Response;
public class PipelineClientTest {
static final String host = "xxxxxxxx.m.cnhza.kvstore.aliyuncs.com";
```

```
static final int port = 6379;
static final String password = "password";
public static void main(String[] args) {
Jedis jedis = new Jedis(host, port);
//ApsaraDB for Redis instance password
String authString = jedis.auth(password);// password
if (! authString.equals("OK")) {
System.err.println("AUTH Failed: " + authString);
jedis.close();
return;
}
String key = "KVStore-Test1";
jedis.del(key);//Initialization
// -------- Method 1
Pipeline p1 = jedis.pipelined();
System.out.println("-----Method 1-----");
for (int i = 0; i < 5; i++) {
p1.incr(key);
System.out.println("Pipeline sends requests");
}
// After sending all requests, the client starts receiving responses
System.out.println("Sending requests completed. Start to receive response");
List<Object> responses = p1.syncAndReturnAll();
if (responses == null || responses.isEmpty()) {
jedis.close();
throw new RuntimeException("Pipeline error: no responds received");
}
for (Object resp : responses) {
System.out.println("Pipeline receives response: " + resp.toString());
}
System.out.println();
//-------- Method 2
System.out.println("-----Method 2-----");
jedis.del(key);//Initialization
Pipeline p2 = jedis.pipelined();
//Declare the responses first
Response<Long> r1 = p2.incr(key);
System.out.println("Pipeline sends requests");
Response<Long> r2 = p2.incr(key);
System.out.println("Pipeline sends requests");
Response<Long> r3 = p2.incr(key);
```

```
System.out.println("Pipeline sends requests");
Response<Long> r4 = p2.incr(key);
System.out.println("Pipeline sends requests");
Response<Long> r5 = p2.incr(key);
System.out.println("Pipeline sends requests");
try{
r1.get(); //Errors occur because the client has not started receiving responses
}catch(Exception e){
System.out.println(" <<< Pipeline error: the client has not started receiving responses >>> ");
}
// After sending all requests, the client starts receiving responses
System.out.println("Sending requests completed. Start to receive response");
p2.sync();
System.out.println("Pipeline receives response: " + r1.get());
System. Out. println ("Pipeline receives response:" + r2.get ());
System. Out. println ("Pipeline receives response:" + r3.get ());
System. Out. println ("Pipeline receives response:" + r4.get ());
System. Out. println ("Pipeline receives response:" + r5.get ());
jedis.close();
}
}
```

Output 2

After you access the ApsaraDB for Redis instance with the correct address and password and run the preceding Java code, the following output is displayed.

```
----- Method 1 -----
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
After sending all requests, the client starts receiving responses.
Pipeline receives response 1
Pipeline receives response 2
Pipeline receives response 3
Pipeline receives response 4
Pipeline receives response 5
----- Method 2 -----
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
Pipeline sends a request
<Pipeline error: The client has not started receiving responses>
After sending all requests, the client starts receiving responses.
Pipeline receives response 1
Pipeline receives response 2
Pipeline receives response 3
Pipeline receives response 4
Pipeline receives response 5
```

2.6. Transactions

ApsaraDB for Redis supports a mechanism to define transactions, as in Redis.

Scenario introduction

The transaction feature allows you to use the **MULTI**, **EXEC**, **DISCARD**, **WATCH**, and **UNWATCH** commands to execute atomic transactions.

Note that the definition of transaction in Redis is different from that in relational databases. If an operation fails or the transaction is canceled by the **DISCARD** command, Redis does not perform the "transaction rollback".

Sample code 1: Two clients operate on different keys

```
package transcation.kvstore.aliyun.com;
import java.util.List;
import redis.clients.jedis.Jedis;
```

```
import redis.clients.jedis.Transaction;
public class KVStoreTranscationTest {
static final String host = "xxxxxx.m.cnhza.kvstore.aliyuncs.com";
static final int port = 6379;
static final String password = "password";
//**Note that these two keys have different content
static String client1_key = "KVStore-Transcation-1";
static String client2_key = "KVStore-Transcation-2";
public static void main(String[] args) {
Jedis jedis = new Jedis(host, port);
//ApsaraDB for Redis instance password
String authString = jedis.auth(password);//password
if (! authString.equals("OK")) {
System.err.println("authentication failed: " + authString);
jedis.close();
return;
}
jedis.set(client1_key, "0");
//Starts another thread to simulate another client
new KVStoreTranscationTest().new OtherKVStoreClient().start();
Thread.sleep(500);
Transaction tx = jedis.multi();//Starts the transaction
//The following operations are collectively submitted to the server as "atomic operations"
tx.incr(client1_key);
tx.incr(client1_key);
Thread.sleep(400);//The suspension of the thread does not affect the consecutively executed operati
ons in a transaction. Other thread operations cannot be executed.
tx.incr(client1_key);
Thread.sleep(300);//The suspension of the thread does not affect the consecutively executed operati
ons in a transaction. Other thread operations cannot be executed.
tx.incr(client1_key);
Thread.sleep(200);//The suspension of the thread does not affect the consecutively executed operati
ons in a transaction. Other thread operations cannot be executed.
tx.incr(client1_key);
List<Object> result = tx.exec();//Submits the operation for execution
//Parses and prints out the results
for(Object rt : result){
System.out.println("Client 1 > in transaction> "+rt.toString());
}
jedis.close();
}
```

```
class OtherKVStoreClient extends Thread{
@Override
public void run() {
Jedis jedis = new Jedis(host, port);
//ApsaraDB for Redis instance password
String authString = jedis.auth(password); //password
if (! authString.equals("OK")) {
System.err.println("AUTH Failed: " + authString);
jedis.close();
return;
}
jedis.set(client2_key, "100");
for (int i = 0; i < 10; i++) {
try {
Thread.sleep(300);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Client 2 > "+jedis.incr(client2_key));
}
jedis.close();
}
}
}
```

Output 1

After you access the ApsaraDB for Redis instance with the correct address and password and run the preceding Java code, the following output is displayed. Here, we can see that Client 1 and Client 2 are in different threads. The operations in the transaction submitted by Client 1 are executed sequentially. Client 2 requests for operating on another key during this period, but the operation is blocked and Client 2 has to wait until all the operations in the Client 1 transaction have been completed.

```
Client 2 > 101
Client 2 > 102
Client 2 > 103
Client 2 > 104
Client 1> in transaction> 1
Client 1> in transaction> 2
Client 1> in transaction> 3
Client 1> in transaction> 4
Client 1> in transaction> 5
Client 2> 105
Client 2> 106
Client 2> 107
Client 2> 108
Client 2> 109
Client 2> 110
```

Sample code 2: Two clients operate on the same key

By slightly modifying the preceding code, we can have the two clients operate on the same key. The other parts of the code remain unchanged.

```
......
// *** Note that the content of these two keys is now the same
static String client1_key = "KVStore-Transcation-1";
static String client2_key = "KVStore-Transcation-1";
......
```

Output 2

After the modified Java code is executed, the output is displayed as follows. We can see that the two clients are in different threads but operate on the same key. However, while Client 1 uses the transaction mechanism to operate on this key, Client 2 is blocked and has to wait until all the operations in the Client 1 transaction are completed.

```
Client 2> 101
Client 2> 102
Client 2> 103
Client 2> 104
Client 1> in transaction> 105
Client 1> in transaction> 106
Client 1> in transaction> 107
Client 1> in transaction> 108
Client 1> in transaction> 109
Client 2> 110
Client 2> 111
Client 2> 112
Client 2> 113
Client 2> 114
Client 2> 115
```

2.7. Discovery and solutions of hotkey problems

Frequently queried keys in Redis are hotkeys. If not taken care of, hotkeys may cause serious services problems. You can find the solutions for hotkey problems in this topic.

Overview

Causes

There are two causes of hotkey problems:

- The size of user consumption data is much greater than that of production data, as in the cases of hot sale items, hot news, hot issue comments, and celebrity broadcasts.

  Hotkey problems tend to occur unexpectedly, for example, the sales price promotion of popular commodities during Double 11 Shopping Festival. When one of these commodities is browsed or purchased tens of thousands of times, a large number of requests occur, which causes a hotkey problem. Similarly, hotkey problems tend to occur in scenarios where there are more writes than reads. For example, hot news, hot issue comments, and celebrity broadcasts.

- In these cases, the hotkey access is much higher than the access of other Redis keys. Therefore, most of the access traffic is centralized to a specific Redis instance, and the Redis instance may reach a performance bottleneck.

  When a piece of data is accessed on the server, the data is usually split or sliced. During this process, the corresponding key is accessed on the server. When the access traffic exceeds the performance threshold of the server, the hotkey key problem occurs.

Impact of hotkey problems

- The traffic is concentrated and reaches the upper limit of the physical network adapter.
- Too many requests queue up, crashing the sharding service of the cache.
- The database is overloaded. A service avalanche occurs.

As mentioned above, when the number of hotkey requests on a server exceeds the upper limit of the network adapter on the server, the server stops providing other services due to the excessive concentration of traffic. If the distribution of hotspots is too dense, a large number of hotkeys are cached. When the cache capacity is exhausted, the sharding service of the cache crashes. After the cache service crashes, the newly generated requests are cached on the background database. Due to its poor performance, this database is prone to exhaustion when handling a large number of requests. The exhaustion of the database leads to a service avalanche and a dramatic downgrading of the performance.

Common solutions

Reconstruct the server or client to improve the performance.

Server cache solution

The client sends requests to the server. The server is a multi-thread service, and a local cache space based on the cache LRU policy is available. When the server is congested, it directly repatriates the requests rather than forwarding them to the database. Only after the congestion is cleared can the server send the requests from the client to the database and re-write the data to the cache. By using this solution, the cache is accessed and rebuilt.

However, this solution has the following problems:

- Cache building problem of the multi-thread service when the cache fails
- Cache building problem when the cache is missing
- Dirty reading problem

"MemCache + Redis" solution

In this solution, a separate cache is deployed on the client to resolve the hotkey problem. The client first accesses the service layer and then the cache layer of the same server. This solution has the following advantages: nearby access, high speeds, and no bandwidth limit. However, it has the following disadvantages:

- Wasted memory resources
- Dirty reading problem

Local cache solution

Using the local cache incurs the following problems:

- Hotspots must be detected in advance.
- The cache capacity is limited.
- The inconsistency duration is long.
- The omission of hotkeys.

If traditional hotkey solutions are all defective, how can the hotkey problems be resolved?

ApsaraDB for Redis solution for hotkey problems

Read/write splitting solution

The following describes the functions of different nodes in the architecture:

- Load balancing is implemented at the SLB layer.
- Read/write splitting and automatic routing are implemented at the proxy layer.
- Write requests are processed by the master node.
- Read requests are processed by the read-only node.
- High availability (HA) is implemented on the replica node and the master node.

In practice, the client sends requests to SLB, and SLB distributes these requests to multiple proxies. The proxies identify and classify the requests and distribute them. For example, a proxy sends all write requests to the master node and all read requests to the read-only node. But the read-only node in the module can be expanded to solve the problem of hotkey reading. Read/write splitting supports flexible scaling for hotkey reading and can store a large number of hotkeys. It is client-friendly.

Hotspot data solution

In this solution, hotkeys are discovered and stored to resolve the hotkey problem. The client accesses SLB and distributes requests to a proxy through SLB. Then, the proxy forwards the requests to the background Redis by the means of routing.

A cache is added on the server. Specifically, a local cache is added to the proxy. This cache uses the LRU algorithm to cache hotspot data. A hotspot data calculation module is added to the background database node to return the hotspot data.

The proxy architecture has the following benefits:

- The proxy caches the hotspot data locally, and its reading capability is horizontally scalable.
- The database node regularly calculates the hotspot data set.
- The database feeds the hotspot data back to the proxy.
- The proxy architecture is completely transparent to the client, and no compatibility is required.

Process hotkeys

Read hotspot data

The processing of hotkeys is divided into two jobs: writing and reading. During the data writing process, SLB receives data K1 and writes it to a Redis database through a proxy. If K1 becomes a hotkey after the calculation conducted by the background hotspot module, the proxy caches the hotspot. In this way, the client can directly access K1 the next time, without using Redis. The proxy can be horizontally expanded, so the accessibility of the hotspot data can be enhanced infinitely.

Discover hotspot data

The database first counts the requests that occur in a cycle. When the number of requests reaches the threshold, the database locates the hotkeys and stores them in an LRU list. When a client attempts to access data by sending a request to the proxy, Redis enters the feedback phase and marks the data if it finds that the target access point is a hotspot.

The database uses the following methods to calculate the hotspots:

- Hotspot statistics based on statistical thresholds.
- Hotspot statistics based on the statistical cycle.
- Statistics collection method based on the version number without resetting the initial value.
- Calculating hotspots on the database has little impact on the performance and only occupies a small amount of memory.

Comparison of two solutions

The preceding analysis shows that compared with the traditional solutions, Alibaba Cloud has made significant improvements in resolving the hotkey problem. The read/write splitting solution and the hotspot data solution can be expanded horizontally. These two solutions are transparent to the client, though they cannot ensure complete data consistency. The read/write splitting solution supports storing a larger amount of hotspot data, while the proxy-based hotspot data solution is more cost-effective.

2.8. ApsaraDB for Redis supports Double 11 Shopping Festival

ApsaraDB for Redis works as an important support for processing surging e-commerce promotions and orders during Double 11 Shopping Festival.

Background

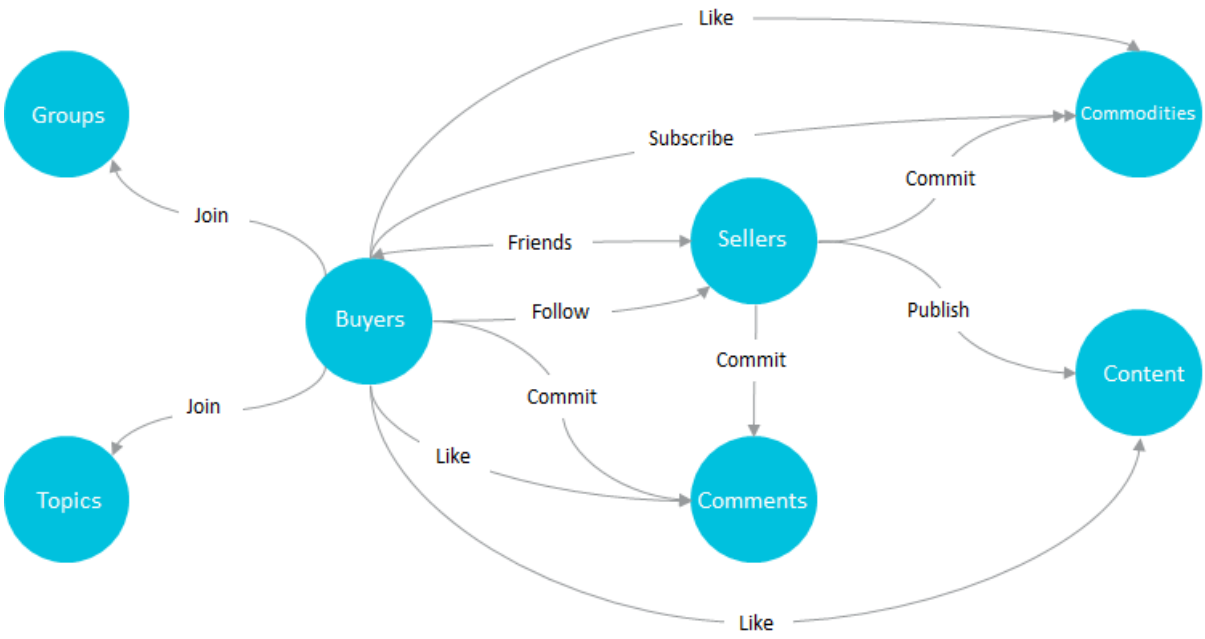
ApsaraDB for Redis provides multiple editions as follows: standard single-replica edition, standard dual-replica edition, and cluster edition.

The standard single-replica edition and standard dual-replica edition feature high compatibility and support Lua scripting and geographical location-based computing. The cluster edition provides large capacities and high performance, and solves the issues caused by single-server performance limits due to Redis single-thread model.

ApsaraDB for Redis works in a two-node hot standby structure by default and supports backup and recovery. Also, the Redis source code team of Alibaba Cloud constantly optimizes and upgrades the ApsaraDB for Redis service, and provides powerful security protections. This topic simplifies some scenarios of Double 11 Shopping Festival and describes the features of ApsaraDB for Redis. Actual scenarios are more complex.

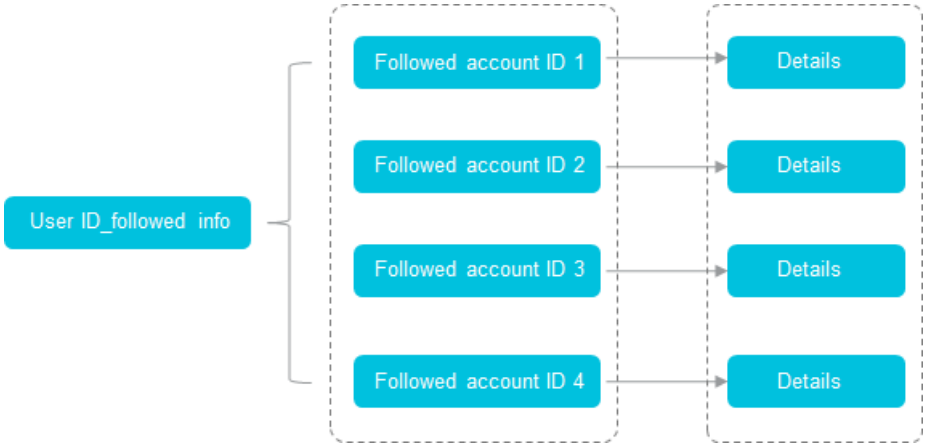
Store social relations for hundreds of millions of users in Weitao community

Weitao community carries social relations for hundreds of millions of Taobao users. Taobao users can specify a list of followers and merchants can maintain the data of regular customers or followers. The following figure shows the overall social relations.



To express these social relations, a traditional relational database model requires complex business design and results in poor user experience. A cluster instance of ApsaraDB for Redis caches followers chains of Weitao community. This simplifies the storage of followers data, and ensures excellent user experience during Double 11 Shopping Festival. Hash tables store followers data of Weitao community. The following figure shows the storage structure. You can call required API operations to query the following data:

- Whether Users A and B are followers of each other
- List of items User A is following

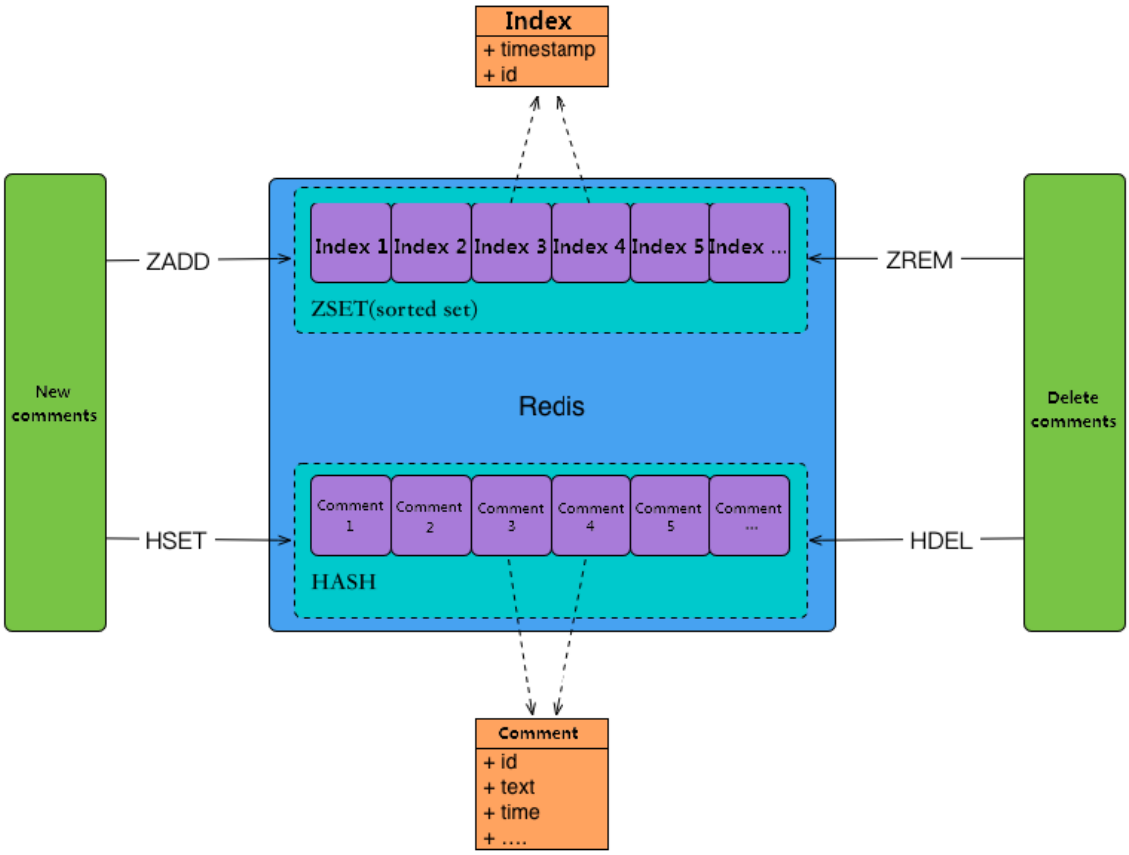


Paginate comments to live videos in Tmall based on a cursor

When mobile users view live videos during Double 11 Shopping Festival, they can obtain more comments to the live videos in three ways:

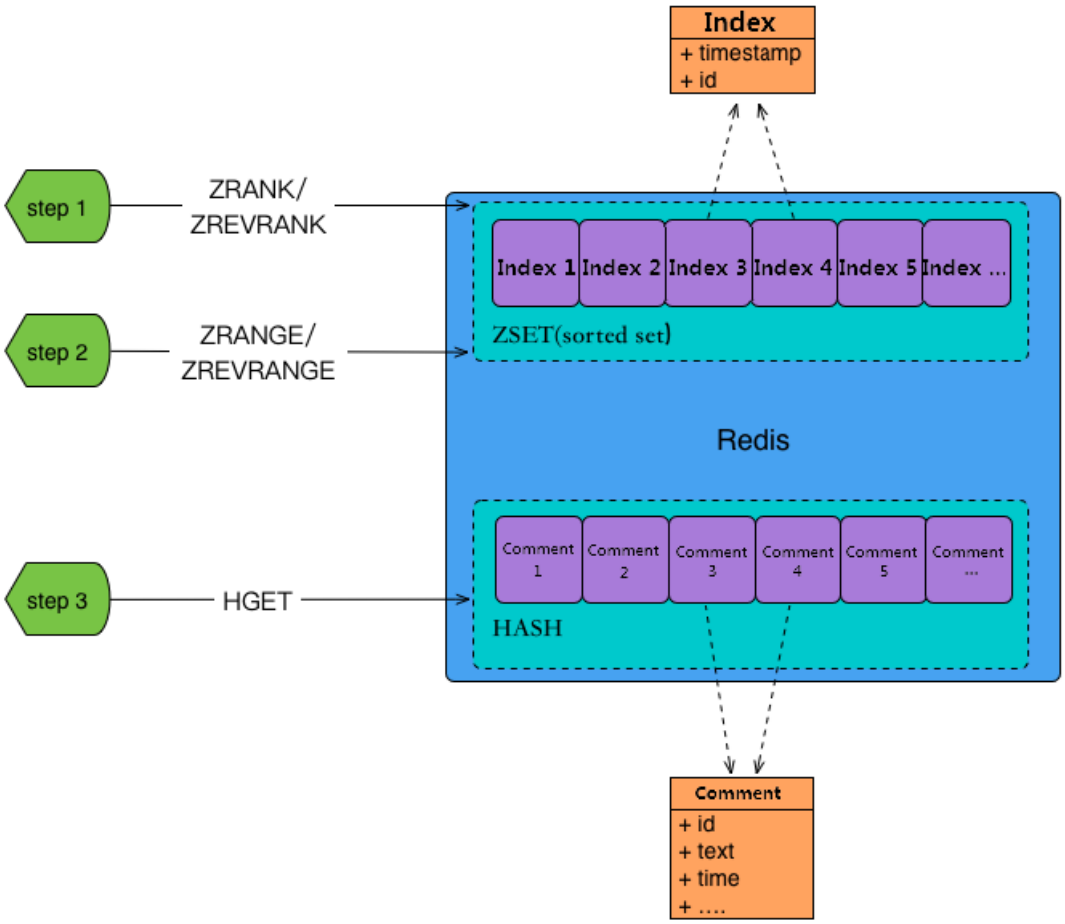
- Pull down for incremental comments: obtain a specified number of incremental comments from the specified position up.
- Pull-down refresh: obtain a specified number of the latest comments.
- Pull up for incremental comments: obtain a specified number of incremental comments from the specified position down.

The mobile live video streaming system uses ApsaraDB for Redis to optimize the business scenario. This ensures the success rate of comments to live videos and supports more than 50,000 transactions per second (TPS) and response time in milliseconds. The live video streaming system writes two types of data for each live video, including indexes and comments. The system writes indexes in sorted sets to sort comments, and stores the comments in hash tables. You can obtain an index ID from the indexes and retrieve a list of comments by reading the hash tables. The following figure shows the process of writing comments.



After a user refreshes the list, the background retrieves the corresponding comments. This process is as follows:

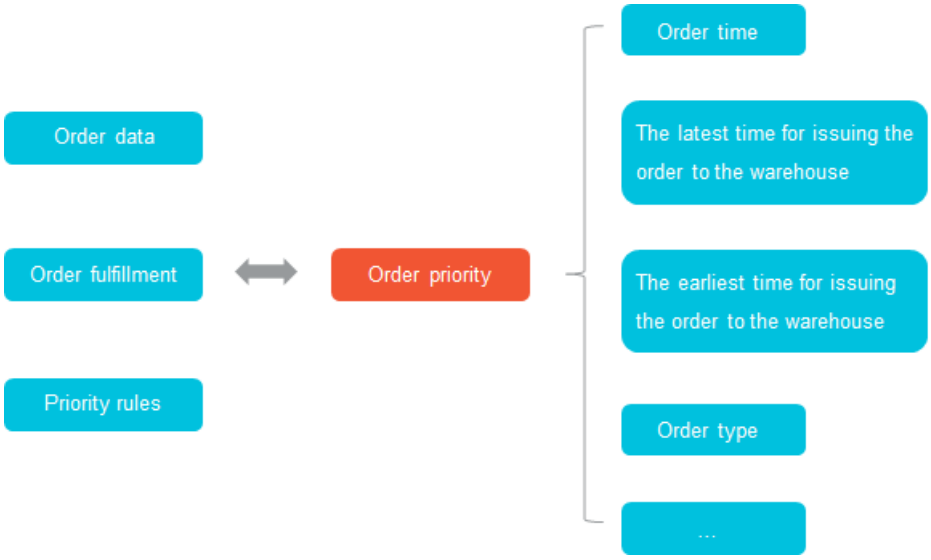
1. Obtain the current index ID.
2. Retrieve the index list.
3. Obtain the comments.


Index
+ timestamp
+ id
step 1
ZRANK/
ZREVRANK
step 2
ZRANGE/
ZREVRANGE
Index 1
Index 2
Index 3
Index 4
Index 5
Index ...
ZSET(sorted set)
Redis
step 3
HGET
Comment 1
Comment 2
Comment 3
Comment 4
Comment 5
Comment ...
HASH
Comment
+ id
+ text
+ time
+ ....

Sort orders in Cainiao order fulfillment center

After a user buys a commodity during Double 11 Shopping Festival, Cainiao warehouse and distribution system generates and processes a corresponding logistics order. The decision-making system generates an order fulfillment plan based on the order data. Therefore, the warehouse and distribution system can provide intelligent and collaborative services across each stage. The plan specifies the time for issuing the order to the warehouse, the time for outbound delivery, the time for item collection, and the time for delivering the item. The order fulfillment center provides the logistics service according to the order fulfillment plan. Due to the limited capacities of warehouses and distribution, the system processes the earliest orders in priority. Therefore, ApsaraDB for Redis sorts the orders by priority before the order fulfillment center issues them to the warehouse or for delivery.

The order fulfillment center uses ApsaraDB for Redis to sort logistics orders and determine the priorities of these orders.



2.9. Use ApsaraDB for Redis to build a business system for handling flash sales

The flash sales strategy is commonly used for promotion activities and brand marketing in the e-commerce industry. This strategy can help you increase the number of unique visitors and customer loyalty of your platform. An excellent business system can improve the stability of your platform and ensure the fairness of flash sales. This improves user experience and the reputation of your platform, and maximizes the benefits of flash sales. This topic describes how to use the caching feature of ApsaraDB for Redis to build a highly concurrent business system for handling flash sales.

Characteristics of flash sales

A flash sales activity is intended to sell scarce or special commodities for specified quantities in a timed manner, and attract a large number of buyers. However, only a few buyers place orders during the promotion. A flash sales activity brings visits and order requests dozens or hundreds of times that in regular sales activities on your platform within a short time period.

A flash sales activity is divided into three phases:

- Before the promotion: Buyers keep refreshing the commodity details page. As a result, the number of requests for this page instantaneously spikes.
- During the promotion: Buyers place orders. The number of order requests reaches the instantaneous peak.
- After the promotion: Some buyers that have successfully placed orders keep querying order status or cancel orders. Most buyers keep refreshing the commodity details page and wait for opportunities to place orders after other buyers cancel their orders.

In most cases, a database uses row-level locking to handle requests submitted by buyers. The database allows only requests that hold the lock to query inventory data and place orders. However, the database is incapable of handling high concurrency in this case. The service may be blocked by a large number of requests, which is considered a server crash to the buyers.

Business system for handling flash sales

During a flash sales activity, the business system may receive a large amount of user traffic. However, only a few of the requests are valid. Based on the hierarchy of the system architecture, you can identify and block invalid requests in advance in each phase.

Use the browser cache and Content Delivery Network (CDN) to process user traffic that request static content

Before a flash sales activity, buyers keep refreshing the commodity details page. As a result, the number of requests for this page instantaneously spikes. To resolve this issue, you must present details about commodities for flash sales and details about regular commodities on different web pages. Use static elements to present details about commodities for flash sales. Static data can be cached in the browser and on CDN, except for the place order function that requires interaction between the browser and server. In this way, only a small fraction of the traffic incurred by refreshing the page before the promotion flows to the server.

Use a read/write splitting instance of ApsaraDB for Redis to cache content and block invalid requests

CDN is used to filter and block user traffic in phase 1. In phase 2, you can use a read/write splitting instance of ApsaraDB for Redis to block invalid requests. In phase 2, the business system is responsible for retrieving data. The read/write splitting instance is capable of handling more than 600,000 queries per second, which completely meets the business demands.

Use the data control module to cache the data of commodities for flash sales to the read/write splitting instance, and specify the flag that indicates the promotion activity begins:

```
"goodsId_count": 100 //The total number of commodities.
"goodsId_start": 0 //The flag that indicates the promotion activity begins.
"goodsId_access": 0 //The number of order requests that have been accepted.
```

1. Before the promotion activity begins, the value of the goodsId_Start flag retrieved by the server cluster is 0, which indicates that the promotion activity has not started.
2. After the data control module changes the value of the goodsId_start flag to 1, the promotion activity begins.
3. The server cluster then caches the goodsId_start flag and accepts order requests. The cluster updates the number of accepted order requests in goodsId_access. The number of remaining commodities is calculated as follows: goodsId_count - goodsId_access.
4. After the number of placed orders reaches the value of goodsId_count, the business system blocks subsequent order requests. The number of remaining commodities is set to 0.

In this way, the business system accepts only a small fraction of the order requests. In case of high concurrency, it is acceptable that more traffic may flow to the system. Therefore, you can control the percentage of order requests that the system accepts.

Use a master-replica instance of ApsaraDB for Redis to cache inventory data and accelerate inventory deduction

After the business system receives an order request, the system checks the order information and deducts the inventory. To avoid retrieving data directly from the backend database, you can use a master-replica instance of ApsaraDB for Redis to deduct the inventory. The master-replica instance supports more than 100,000 QPS. ApsaraDB for Redis can help you optimize inventory query, block invalid order requests, and increase the overall throughput of the business system for handling flash sales.

You can use the data control module to cache the inventory data to the ApsaraDB for Redis instance before the promotion activity begins. The instance stores the commodity data for promotion in a hash table.

```
"goodsId" : {
"Total": 100
"Booked": 100
}
```

To deduct the inventory, the server runs the following Lua script and connects to the ApsaraDB for Redis instance to obtain the permission on placing orders. Lua ensures the atomicity across multiple commands because Redis is a single-thread model.

```
local n = tonumber(ARGV[1])
if not n or n == 0 then
return 0
end
local vals = redis.call("HMGET", KEYS[1], "Total", "Booked");
local total = tonumber(vals[1])
local blocked = tonumber(vals[2])
if not total or not blocked then
return 0
end
if blocked + n <= total then
redis.call("HINCRBY", KEYS[1], "Booked", n)
return n;
end
return 0
```

Run the `SCRIPT LOAD` command to cache the Lua script to the ApsaraDB for Redis instance in advance, and then run the `EVALSHA` command to execute the script. This method requires less network bandwidth than directly running the `EVAL` command.

```
redis 127.0.0.1:6379>SCRIPT LOAD "lua code"
"438dd755f3fe0d32771753eb57f075b18fed7716"
redis 127.0.0.1:6379>EVALSHA 438dd755f3fe0d32771753eb57f075b18fed7716 1 goodsId 1
```

If the ApsaraDB for Redis instance returns the value n as the number of commodities that buyers have ordered, the flash sales system determines that the current inventory deduction is successful.

Use a master-replica instance of ApsaraDB for Redis to asynchronously write order data to the database based on message queues

The flash sales business system writes order data to the database after successful inventory deduction. For a few commodities, the system can directly perform operations in the database. If the number of commodities for promotion is more than 10,000 or 100,000, lock conflicts may occur and cause performance bottlenecks in the database. Therefore, to avoid direct operations in the database, the flash sales system writes order data to message queues to complete the order process.

1. The ApsaraDB for Redis instance provides message queues in a list structure.

```
orderList {
[0] = {Order content}
[1] = {Order content}
[2] = {Order content}
...
}
```

2. The flash sales business system writes order content to the ApsaraDB for Redis instance.

```
LPUSH orderList {Order content}
```

3. The asynchronous order module sequentially retrieves order data from the ApsaraDB for Redis instance and writes order data to the database.

```
BRPOP orderList 0
```

The ApsaraDB for Redis instance provides message queues and asynchronously writes order data to the database to complete the order process.

The data control module manages synchronization of promotion data

At the start, the flash sales business system uses the read/write splitting instance of ApsaraDB for Redis to block invalid traffic and allows a fraction of valid traffic to continue the order process. Afterward, the flash sales business system has to process more traffic caused by order authentication failures and returning orders. Therefore, the data control module regularly computes data in the database, and synchronizes the data to the master-replica instance and then to the read/write splitting instance.

2.10. Read/write splitting in Redis

ApsaraDB for Redis read/write splitting instances support multiple read replicas, providing high-performance service for more-reading and less-writing scenarios.

Background

In ApsaraDB for Redis, whether in the master-replica edition or the cluster edition, replica serves as a standby database and does not provide external services. When high availability is enabled and the primary master fails, the replica can be promoted to the master to take over read and write operations. In this architecture, read and write requests are completed on the master node with high consistency, but the performance is limited by the number of master nodes. Often, even when the user data is small, the cluster specification still needs to be updated because the traffic and the concurrency is too high.

In business scenarios where there are more reads than writes, ApsaraDB for Redis provides a read/write splitting specification that is transparent, flexible, highly available, and high-performance. This specification helps users minimize the cost.

Archietecture

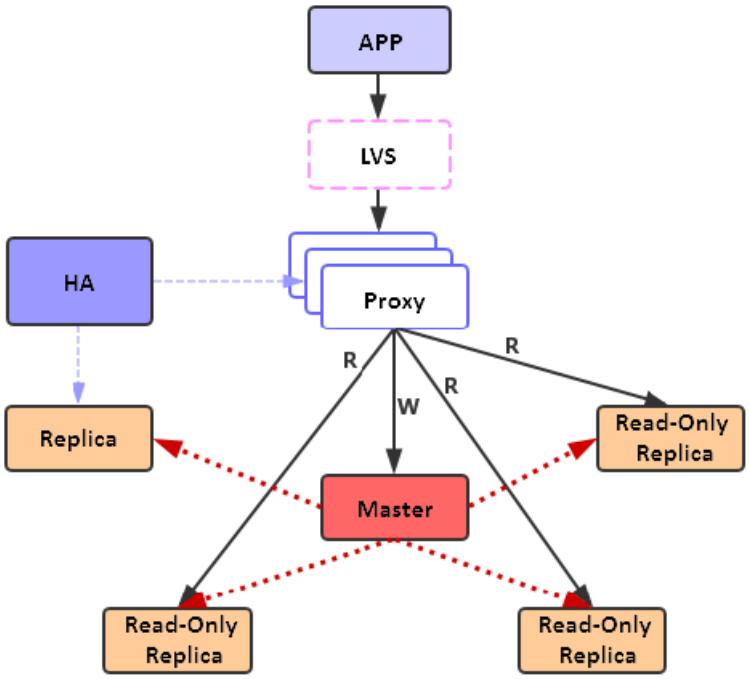
Redis cluster mode has several roles, including redis-proxy, Master, replica, and HA. In a read/write splitting instance, the read-only replica role is added to take over the read traffic. The replica serves as a hot standby and does not provide services. This architecture remains compatible with existing cluster specifications. The proxy forwards the read and write requests to the master node or a read-only replica accordingly by weight. The highly available (HA) cluster is responsible for monitoring the health status of nodes. When an exception occurs, the replica will take over or the read-only replica will be rebuilt to perform critical operations, and the route will be updated.

Typically, according to the data synchronization methods of master nodes and read-only replicas, there are two replication types: star replication and cascading replication.

Star replication

In the star replication, data volumes are replicated on multiple nodes in parallel. Since the master node is connected to all other read-only replica nodes, there is no need to failover a replica node in the event of a failure thus reducing the duration of recovery.

Redis uses a single-thread and single-process model. The data replication between the master node and the replica node is processed in the main thread. The CPU utilization on the master node due to data synchronization increases with the number of read-only replicas. Therefore, the write performance of the cluster is diminished by the increasing number of read-only replica nodes. In the star replication, the outbound bandwidth of the master node also increases with the number of read-only replicas. The tradeoffs between these two replication types is one of latency and throughput. Due to the high CPU utilization on the master node and the heavy network load, the low-latency star replication delivers lower throughput than the cascading replication. The performance of the entire cluster is limited by the master node.

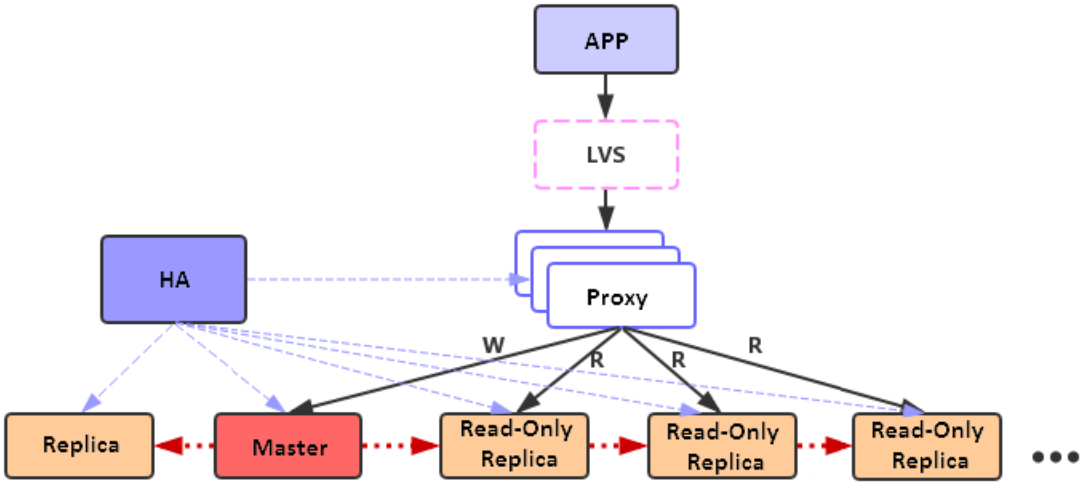


Cascading replication

All read-only replica nodes are replicated sequentially on intermediate and tail nodes, as shown in the following figure. The master node only needs to synchronize the data to the replica node and the first read-only replica on the replication chain.

Cascading replication solves the extension problem of star replication. In theory, the number of read-only replicas can increase infinitely, and the performance of the entire cluster will increase accordingly.

In a chain replication, the longer the replication chain, the greater the delay between the original master node and the read-only replica at the end of the chain.This shortcoming is usually acceptable, since that the read/write splitting is mainly used in scenarios that have low requirements on consistency. However, if a node in the replication chain fails, all data on the downstream nodes will be delayed significantly. What's worse, this may lead to a full synchronization that is passed to the end of the replication chain, and reduce the service performance. To solve this problem, the Redis read/write splitting uses an optimized binlog replication provided by Alibaba Cloud to minimize the probability of full synchronization.



In light of the preceding discussions and comparisons, Redis chooses a cascading replication architecture for read/write splitting.

Advantages of Redis read/write splitting

Transparent and compatible

Redis read/write splitting uses redis_proxy to forward requests. There are certain restrictions on the use of multi-sharding commands. This feature is fully compatible with the upgrade from the master-replica edition to the single-sharding read/write splitting, and the upgrade from the cluster specification to the multi-sharding read/write splitting.

The user establishes a connection with redis-proxy, a Redis proxy that supports read/write splitting. The proxy recognizes whether the request sent by the client is read or write, and then performs load balancing according to the weight. The proxy forwards write requests to the master and read requests to the read-only replica. The master also supports read requests by default, which can be controlled by weight.

You can purchase instances of read/write splitting specifications and use them directly with any client, with no modification to the business. You can enjoy an improved service performance almost at no cost.

Highly available

The high availability module (HA) monitors the health of all nodes to ensure instance availability. If the master node fails, the HA module redirects the requests to a new master node. If a read-only replica fails, the HA module can detect it promptly, create a new read-only replica, and turn the failed node offline.

In addition to the HA module, redis-proxy can also detect the state of each read-only replica in real time. During a read-only replica failure, redis_proxy automatically reduces the weight of this node. If a read-only replica fails multiple times, redis-proxy will temporarily block this node. After the node recovers, its weight will be resumed to a normal level.

HA and redis_proxy work together to minimize the business awareness of backend exceptions and improve service availability.

High performance

In business scenarios where there are more reads than writes, using the cluster edition directly is not the best solution. The read/write splitting provides more options, and you can choose the best specification based on the business scenario to make full use of the read-only replicas.

Multiple specifications are available: 1 master + 1 read-only replica, 1 master + 3 read-only replicas, and 1 master + 5 read-only replicas. You can submit a ticket if you need a different specification. This service provides 0.6 million QPS and 192 MB/s service capability. This service breaks the resource limit of a single machine since it is fully compatible with all commands. In the following versions, there will be no specification limit, and users can increase or decrease the number of read-only replicas based on the business traffic.

Specification	QPS	Bandwidth
1 master	80 to 100 thousand reads and writes	10 to 48 MB
1 master + 1 read-only replica	0.1 million writes + 0.1 million reads	20 to 64 MB
1 master + 3 read-only replicas	0.1 million writes + 0.3 million reads	40 to 128 MB
1 master + 5 read-only replicas	0.1 million writes + 0.5 million reads	60 to 192 MB

Concluding remarks

The asynchronous replication of the Redis master-replica edition may read old data from the read-only replica, so read/write splitting feature requires the business to tolerate a certain degree of data inconsistency. The following editions will grant users more flexibility in parameter configurations, such as the allowed maximum delay time.

2.11. JedisPool optimization

Proper values of JedisPool parameters allow you to improve the performance of Redis. This topic describes how to use JedisPool and configure the resource pool parameters. This topic also describes recommended parameter configurations to optimize JedisPool.

How to use JedisPool

Jedis 2.9.0 is used in this example. The following sample code shows the Maven dependency:

```
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<version>2.9.0</version>
<scope>compile</scope>
</dependency>
```

Jedis manages the resource pool by using Apache Commons-pool2. When you define JedisPool, we recommend that you consider the GenericObjectPoolConfig parameter (resource pool). The following sample code shows how to use this parameter.

```
GenericObjectPoolConfig jedisPoolConfig = new GenericObjectPoolConfig();
jedisPoolConfig.setMaxTotal(...);
jedisPoolConfig.setMaxIdle(...);
jedisPoolConfig.setMinIdle(...);
jedisPoolConfig.setMaxWaitMillis(...) ;
...
```

The following example shows how to initialize JedisPool:

```
// redisHost specifies the IP address of the instance. redisPort specifies the port of the instance. redi
sPassword specifies the password of the instance. The timeout parameter specifies both the connecti
on timeout and the read/write timeout.
JedisPool jedisPool = new JedisPool(jedisPoolConfig, redisHost, redisPort, timeout, redisPasswor//d);
// Run the following command:
Jedis jedis = null;
try {
jedis = jedisPool.getResource();
// Specific commands
jedis.executeCommand()
} catch (Exception e) {
logger.error(e.getMessage(), e);
} finally {
// In JedisPool mode, the Jedis resource is returned to the resource pool.
if (jedis ! = null)
jedis.close();
}
```

Parameters

The Jedis connection is a resource managed by JedisPool in the connection pool. JedisPool is a thread-safe pool of connections. It allows you to keep all resources within a manageable range. Specify the GenericObjectPoolConfig parameter properly can improve the performance of Redis and reduce the resource consumption. The following two tables list some important parameters and provide recommended configurations.

Parameters related to resource settings and resource usage

Parameter	Description	Default value	Recommendation
maxTotal	The maximum number of connections that are supported by the pool.	8	For more information, see Recommendations for key parameters.
maxIdle	The maximum number of idle connections in the pool.	8	For more information, see Recommendations for key parameters.
minIdle	The minimum number of idle connections in the pool.	0	For more information, see Recommendations for key parameters.
blockWhenExhausted	Specifies whether the client must wait when the resource pool is exhausted. The following maxWaitMillis parameter takes effect only when this parameter is set to true.	true	We recommend that you use the default value.
maxWaitMillis	The maximum number of milliseconds that the client needs to wait when no connection is available.	A value of -1 specifies that the connection will never time out.	We recommend that you do not use the default value.
testOnBorrow	Specifies whether connections will be validated by using the PING command before they are borrowed from the pool. Invalid connections will be removed from the pool.	false	We recommend that you set this parameter to false when the workload is heavy. This allows you to reduce the consumption caused by a ping test.
testOnReturn	Specifies whether connections will be validated using the PING command before they are returned to the pool. Invalid connections will be removed from the pool.	false	We recommend that you set this parameter to false when the workload is heavy. This allows you to reduce the consumption caused by a ping test.

Parameter	Description	Default value	Recommendation
jmxEnabled	Specifies whether to enable JMX monitoring.	true	We recommend that you enable JMX monitoring. Note that the feature of your application also needs to be enabled.

Idle Jedis object detection consists of the following four parameters. testWhileIdle is the switch of this feature.

Parameters related to idle resource detection

Parameter	Description	Default value	Recommendation
testWhileIdle	Specifies whether to enable the idle resource detection.	false	true
timeBetweenEvictionRunsMillis	Specifies the cycle of idle resources detection. Unit: milliseconds.	A value of -1 specifies that no idle resource are detected.	We recommend that you specify this parameter and set it to a proper value as needed. You can also use the default configuration in JedisPoolConfig.
minEvictableIdleTimeMillis	The minimum idle time in milliseconds of a resource in the resource pool. When the upper limit is reached, the idle resource will be evicted.	180000 (30 minutes)	The default value is suitable for most cases. You can also use the configuration in JeidsPoolConfig based on your business requirements.
numTestsPerEvictionRun	The number of resources to be detected at each idle resource detection.	3	You can change the value based on your application connections. A value of -1 specifies that idle resource detection will be performed on all connections.

Jedis provides JedisPoolConfig that uses some configurations of GenericObjectPoolConfig for idle resource detection.

```
public class JedisPoolConfig extends GenericObjectPoolConfig {
public JedisPoolConfig() {
// defaults to make your life with connection pool easier :)
setTestWhileIdle(true);
//
setMinEvictableIdleTimeMillis(60000);
//
setTimeBetweenEvictionRunsMillis(30000);
setNumTestsPerEvictionRun(-1);
}
}
```

Note You can view all default values in org.apache.commons.pool2.impl.BaseObjectPoolConfig.

Recommendations for key parameters

maxTotal: The maximum number of connections.

To set a proper value of maxTotal, you need to take into account the following factors:

- The required concurrent connections based on business requirements.
- The amount of time that is consumed by the client to run the command.
- The limit of Redis resources. For example, the product of multiplying maxTotal by the number of nodes (applications) must be smaller than the supported maximum number of connections in Redis. You can view the maximum connections on the Instance Information page in the ApsaraDB for Redis console.
- The resource that is cost to create and release connections. When the number of connections created and released is large for each request, the creation and release process takes a heavy toll.

The time for running a command to obtain a resource consists of the time to borrow or return the resource, the time consumed for JedisPool to run the command, and the time for network connection. For example, the average time consumed to run a command to obtain a resource is about 1 ms, the queries per second (QPS) of a connection is about 1,000, and the expected QPS is 50,000. In this case, the required theoretical pool size is 50 (50,000/1,000 = 50).

But this is only a theoretical value. To reserve some resources, the value of the maxTotal parameter can be larger than the theoretical value. However, if this value is too large, the connections will consume a large amount of client and server resources. On the other hand, for servers like Redis that has a high QPS, if a large number of commands block, even a large resource pool cannot solve this problem.

maxIdle and minIdle

maxIdle is the actual maximum number of connections required by the business. maxTotal includes the number of idle connections as a surplus. If the value of maxIdle is too small on heavily loaded systems, `new Jedis` (extra connections) will be created to serve the requests. Therefore, minIdle specifies the minimum number of established connections that need to be kept in the pool.

The connection pool reaches its best performance when maxTotal = maxIdle. This way, the performance is not affected by the scaling of the connection pool. We recommend that you set the values of both parameters to the same value during peak hours of your services. However, if the number of concurrent connections is small or the value of the maxIdle parameter is too large, the connection resources will be wasted.

You can evaluate the size of the connection pool used by each node based on the actual total QPS and the number of clients that call Redis.

Retrieve proper values based on monitoring data

In actual scenarios, a more reliable method is to try to retrieve optimal values based on monitoring data. You can use JMX monitoring or other monitoring tools to discover proper values.

FAQ

Insufficient resources

In the following cases, you cannot obtain resources from the resource pool.

- Timeout:

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool
...
Caused by: java.util.NoSuchElementException: Timeout waiting for idle object
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:449)
```

- When you set the blockWhenExhausted parameter to false, the time specified by borrowMaxWaitMillis is not used and the borrowObject call will block until an idle connection is available.

```
redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool
...
Caused by: java.util.NoSuchElementException: Pool exhausted
at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:464)
```

This exception may not be caused by limited pool size. For more information, see Recommendations for key parameters. To fix this issue, we recommend that you check the network, parameters of the resource pool, the resource pool monitoring (JMX monitoring), the code (for example, the reason may be that `jedis.close()` is not executed), slow queries, and DNS.

Preload JedisPool

The project may quickly time out after it is started if you specify a small timeout value. JedisPool does not create a Jedis connection in the connection pool when it defines the maximum number of resources and the minimum number of idle resources. If no idle connection exists in the pool, a `new Jedis` connection is created. This connection will be released to the pool after it is used. However, it may take a long period of time to create a new connection and release it every time. Therefore, we recommend that you preload JedisPool with the minimum number of idle connections after JedisPool is defined. The following example shows how to reload JedisPool:

```
List<Jedis> minIdleJedisList = new ArrayList<Jedis>(jedisPoolConfig.getMinIdle());

for (int i = 0; i < jedisPoolConfig.getMinIdle(); i++) {
Jedis jedis = null;
try {
jedis = pool.getResource();
minIdleJedisList.add(jedis);
jedis.ping();
} catch (Exception e) {
logger.error(e.getMessage(), e);
} finally {
}
}

for (int i = 0; i < jedisPoolConfig.getMinIdle(); i++) {
Jedis jedis = null;
try {
jedis = minIdleJedisList.get(i);
jedis.close();
} catch (Exception e) {
logger.error(e.getMessage(), e);
} finally {

}
}
```

2.12. Analyze hotkeys in a specific sub-node of a cluster instance

You can use the imonitor command developed by Alibaba Cloud to monitor the request status of a specific node in the Redis cluster, and use redis-faina to discover hotkeys and commands from the monitoring data.

Background information

When you use the ApsaraDB for Redis cluster edition, if the hotkey traffic on a specific node is too large, other services in the server may fail to continue. If the cache of the hotkey exceeds the current cache capacity, the sharding service of the cache will crash.

You can use Performance monitoring and Alert settings to monitor the cluster status in real time and set alert rules. When you discover an overloaded sub-node, you can use the imonitor command to view the client request of the node, and use redis-faina to analyze the hotkey.

Prerequisites

- You have activated an ECS instance that can interconnect with the ApsaraDB for Redis cluster edition.
- You have installed Python and Telnet in the ECS instance.

> **Note** The sample environment in this topic is CentOS 7.4 and Python 2.7.5.

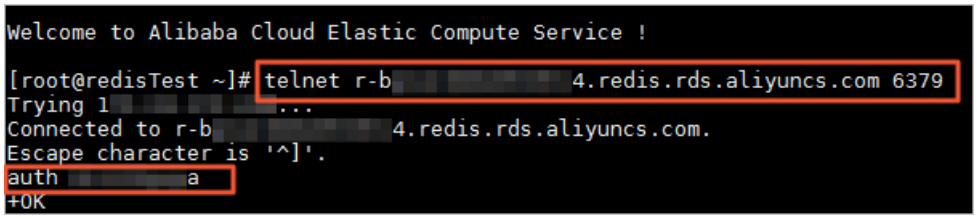
Procedure

1. In the ECS instance, use Telnet to connect to the Redis cluster.
   i. Use `# telnet <host> <port>` to connect to the Redis cluster.

   > **Note** `host` is the connection address of the Redis cluster. `port` is the connection port (the default port number is 6379).

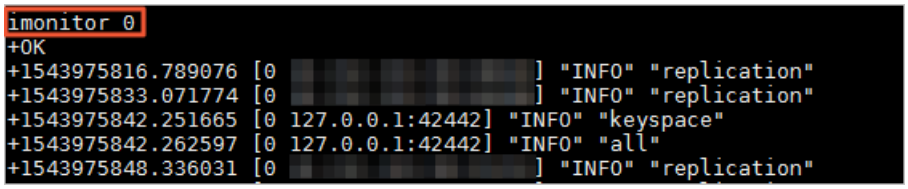
   ii. Enter `auth <password>` for verification.

   > **Note** `password` is the password for the Redis cluster.



> **Note** If `+OK` is returned, the connection is successful.

2. Use `imonitor <db_idx>` to collect the request data of the target node.



> **Note**
>
> The **imonitor** command is similar to the **info** command and the **iscan** command. This command added a parameter to the **monitor** command, and the user can specify the node to run the **monitor** command. In this command, the value range of db_idx is [0, nodecount). You can obtain the value of nodecount by running the **info** command or viewing the instance topology in the console.
>
> In this example, the value of db_idx of the target node is 0.
>
> If `+OK` is returned, the output of of monitored request records continues.

3. Collect the monitoring data based on your business requirements and enter the **QUIT** command. Press Enter to close the Telnet connection.
4. Store the monitoring data to a *.txt* file, and delete the plus sign (+) at the beginning of the line. You can replace this sign by using the text editing tool. The stored file is as follows:

```
[root@redisTest ~]# cat imonitorOut.txt
1543995847.659482 [0                    ] "INFO" "replication"
1543995856.057381 [0 127.0.0.1:58802] "INFO" "keyspace"
1543995856.070002 [0 127.0.0.1:58802] "INFO" "all"
1543995861.653458 [0                    ] "INFO" "ALL"
1543995862.782848 [0                    ] "INFO" "ALL"
1543995862.799096 [0                    ] "INFO" "ALL"
1543995862.863230 [0                    ] "INFO" "CLUSTER"
1543995862.876389 [0                    ] "scan" "0" "MATCH" "*" "COUNT" "3000"
1543995862.942649 [0                    ] "INFO" "replication"
1543995862.943303 [0                    ] "TYPE" "customer:18016"
1543995862.955943 [0                    ] "TYPE" "customer:17167"
```

5. Create a Python script for request analysis, and save it as *redis-faina.py*. The code is as follows:

```
#! /usr/bin/env python
import argparse
import sys
from collections import defaultdict
import re

line_re_24 = re.compile(r"""
^(? P<timestamp>[\d\.]+)\s(\(db\s(? P<db>\d+)\)\s)?"(? P<command>\w+)"(\s"(? P<key>[^(? <! \\
)"]+)(? <! \\)")?( \s(? P<args>. +))? $
""", re.VERBOSE)

line_re_26 = re.compile(r"""
^(? P<timestamp>[\d\.]+)\s\[(? P<db>\d+)\s\d+\.\d+\.\d+\.\d+:\d+]\s"(? P<command>\w+)"(\s"(?
P<key>[^(? <! \\)"]+)(? <! \\)")?( \s(? P<args>. +))? $
""", re.VERBOSE)

class StatCounter(object):

def __init__(self, prefix_delim=':', redis_version=2.6):
self.line_count = 0
self.skipped_lines = 0
self.commands = defaultdict(int)
self.keys = defaultdict(int)
self.prefixes = defaultdict(int)
self.times = []
self._cached_sorts = {}
self.start_ts = None
self.last_ts = None
```

```
self.last_entry = None
self.prefix_delim = prefix_delim
self.redis_version = redis_version
self.line_re = line_re_24 if self.redis_version < 2.5 else line_re_26

def _record_duration(self, entry):
ts = float(entry['timestamp']) * 1000 * 1000 # microseconds
if not self.start_ts:
self.start_ts = ts
self.last_ts = ts
duration = ts - self.last_ts
if self.redis_version < 2.5:
cur_entry = entry
else:
cur_entry = self.last_entry
self.last_entry = entry
if duration and cur_entry:
self.times.append((duration, cur_entry))
self.last_ts = ts

def _record_command(self, entry):
self.commands[entry['command']] += 1

def _record_key(self, key):
self.keys[key] += 1
parts = key.split(self.prefix_delim)
if len(parts) > 1:
self.prefixes[parts[0]] += 1

@staticmethod
def _reformat_entry(entry):
max_args_to_show = 5
output = '"%(command)s"' % entry
if entry['key']:
output += ' "%(key)s"' % entry
if entry['args']:
arg_parts = entry['args'].split(' ')
ellipses = ' ...' if len(arg_parts) > max_args_to_show else ''
output += ' %s%s' % (' '.join(arg_parts[0:max_args_to_show]), ellipses)
return output
```

```
def _get_or_sort_list(self, ls):
key = id(ls)
if not key in self._cached_sorts:
sorted_items = sorted(ls)
self._cached_sorts[key] = sorted_items
return self._cached_sorts[key]

def _time_stats(self, times):
sorted_times = self._get_or_sort_list(times)
num_times = len(sorted_times)
percent_50 = sorted_times[int(num_times / 2)][0]
percent_75 = sorted_times[int(num_times * .75)][0]
percent_90 = sorted_times[int(num_times * .90)][0]
percent_99 = sorted_times[int(num_times * .99)][0]
return (("Median", percent_50),
("75%", percent_75),
("90%", percent_90),
("99%", percent_99))

def _heaviest_commands(self, times):
times_by_command = defaultdict(int)
for time, entry in times:
times_by_command[entry['command']] += time
return self._top_n(times_by_command)

def _slowest_commands(self, times, n=8):
sorted_times = self._get_or_sort_list(times)
slowest_commands = reversed(sorted_times[-n:])
printable_commands = [(str(time), self._reformat_entry(entry)) \
for time, entry in slowest_commands]
return printable_commands

def _general_stats(self):
total_time = (self.last_ts - self.start_ts) / (1000*1000)
return (
("Lines Processed", self.line_count),
("Commands/Sec", '%. 2f' % (self.line_count / total_time))
)
```

```
def process_entry(self, entry):
self._record_duration(entry)
self._record_command(entry)
if entry['key']:
self._record_key(entry['key'])

def _top_n(self, stat, n=8):
sorted_items = sorted(stat.iteritems(), key = lambda x: x[1], reverse = True)
return sorted_items[:n]

def _pretty_print(self, result, title, percentages=False):
print title
print '=' * 40
if not result:
print 'n/a\n'
return

max_key_len = max((len(x[0]) for x in result))
max_val_len = max((len(str(x[1])) for x in result))
for key, val in result:
key_padding = max(max_key_len - len(key), 0) * ' '
if percentages:
val_padding = max(max_val_len - len(str(val)), 0) * ' '
val = '%s%s\t(%. 2f%%)' % (val, val_padding, (float(val) / self.line_count) * 100)
print key,key_padding,'\t',val
print


def print_stats(self):
self._pretty_print(self._general_stats(), 'Overall Stats')
self._pretty_print(self._top_n(self.prefixes), 'Top Prefixes', percentages = True)
self._pretty_print(self._top_n(self.keys), 'Top Keys', percentages = True)
self._pretty_print(self._top_n(self.commands), 'Top Commands', percentages = True)
self._pretty_print(self._time_stats(self.times), 'Command Time (microsecs)')
self._pretty_print(self._heaviest_commands(self.times), 'Heaviest Commands (microsecs)')
self._pretty_print(self._slowest_commands(self.times), 'Slowest Calls')

def process_input(self, input):
for line in input:
self.line_count += 1
line = line.strip()
```

```
match = self.line_re.match(line)
if not match:
if line ! = "OK":
self.skipped_lines += 1
continue
self.process_entry(match.groupdict())

if __name__ == '__main__':
parser = argparse.ArgumentParser()
parser.add_argument(
'input',
type = argparse.FileType('r'),
default = sys.stdin,
nargs = '?',
help = "File to parse; will read from stdin otherwise")
parser.add_argument(
'--prefix-delimiter',
type = str,
default = ':',
help = "String to split on for delimiting prefix and rest of key",
required = False)
parser.add_argument(
'--redis-version',
type = float,
default = 2.6,
help = "Version of the redis server being monitored",
required = False)
args = parser.parse_args()
counter = StatCounter(prefix_delim = args.prefix_delimiter, redis_version = args.redis_version)
counter.process_input(args.input)
counter.print_stats()
```

Note The preceding script is from redis-faina.

6. Run the `python redis-faina imonitorOut.txt` command to parse the monitoring data. *imonitorOut.txt* is the monitoring data stored in the example.

```
[root@redisTest ~]# python redis-faina.py imonitorOut.txt
Overall Stats
============================================
Lines Processed         311
Commands/Sec            0.88

Top Prefixes
============================================
customer                132     (42.44%)
user_agent              24      (7.72%)
simple_registration     12      (3.86%)
detailed_registration   9       (2.89%)
company                 4       (1.29%)

Top Keys
============================================
customer:1446           122     (39.23%)
ALL                     68      (21.86%)
replication             29      (9.32%)
all                     15      (4.82%)
keyspace                15      (4.82%)
user_agent:17358        8       (2.57%)
user_agent:10722        4       (1.29%)
customer:4968           1       (0.32%)

Top Commands
============================================
INFO    128     (41.16%)
HGET    121     (38.91%)
TYPE    50      (16.08%)
HLEN    3       (0.96%)
TTL     3       (0.96%)
HSCAN   3       (0.96%)
scan    1       (0.32%)
GET     1       (0.32%)

Command Time (microsecs)
============================================
Median          603448.0
75%             1556677.0
90%             5215846.0
99%             8019603.0

Heaviest Commands (microsecs)
============================================
INFO    231775519.75
HGET    103355620.75
GET     7377767.75
HLEN    6155302.75
HSCAN   2166953.0
TYPE    2031287.75
scan    66260.0
TTL     35047.25

Slowest Calls
============================================
8397898.75      "INFO" "replication"
8101143.0       "INFO" "ALL"
8079963.75      "INFO" "ALL"
```

Note In the preceding analysis result, Top Keys displays the most requested keys during this time period, and Top Commands displays the most frequently used commands. You can solve the hotkey problem based on the analysis results.

2.13. Use ApsaraDB for Redis to build a broadcasting channel information system

You can use ApsaraDB for Redis to build a broadcasting channel information system that has a low latency and can handle high traffic volumes.

Background information

The broadcasting channel is one of the key features of the live broadcasting system. Except for the broadcasting window, online users, online gifts, comments, likes, rankings, and other data generated in the live broadcast is time-limited, highly interactive, and delay-sensitive. Redis cache service is a suitable solution to handle such data.

The best practice in this topic introduces how to use ApsaraDB for Redis to build a broadcasting channel information system. This topic describes the construction methods for three information types:

- Real-time ranking information
- Counting information
- Timeline information

Real-time ranking information

The real-time ranking information includes an online user list, a list of online gifts, and live comments. Live comments are similar to a message ranking list that is sorted based on the message dimension. The sorted set structure in Redis is suitable to handle the real-time ranking information.

The Redis set is stored in a hash table. The time complexity of the insert, delete, edit, and search operations is O(1). Each member in the set is associated with a score to facilitate sorting and other operations. The following example describes the added and returned live comments to explain how the sorted set works to build a broadcasting channel information system.

- Uses "unix timestamp + milliseconds" as the score to record the last five live comments in the user55 broadcasting channel.

```
redis> ZADD user55:_danmu 1523959031601166 message111111111111
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959031601266 message222222222222
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959088894232 message33333
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959090390160 message444444
(integer) 1
11.160.24.14:3003> ZADD user55:_danmu 1523959092951218 message5555
(integer) 1
```

- Returns the last three live comments:

```
redis> ZREVRANGEBYSCORE user55:_danmu +inf -inf LIMIT 0 3
1) "message5555"
2) "message444444"
3) "message33333"
```

- Returns three live comments within the specified time period:

```
redis> ZREVRANGEBYSCORE user55:_danmu 1523959088894232 -inf LIMIT 0 3
1) "message33333"
2) "message222222222222"
3) "message111111111111"
```

Counting information

In case of the user-related data, the counting information includes the number of unread messages, followers, and fans, and the experience value. The hash structure in Redis is suitable to process this type of data. For example, the number of followers can be processed as follows:

```
redis> HSET user:55 follower 5
(integer) 1
redis> HINCRBY user:55 follower 1 //The number of followers +1
(integer) 6
redis> HGETALL user:55
1) "follow"
2) "6"
```

Timeline information

The timeline information is a list of information sorted in time order. For example, the broadcaster moments and new posts. This type of information is arranged in a fixed chronological order and can be stored using a Redis list or an ordered list. The example is as follows:

```
redis> LPUSH user:55_recent_activitiy '{datetime:201804112010,type:publish,title:The show starts, con
tent:Come on}'
(integer) 1
redis> LPUSH user:55_recent_activitiy '{datetime:201804131910,type:publish,title: Ask for a leave, cont
ent: Sorry, I have plans today.}'
(integer) 2
redis> LRANGE user:55_recent_activitiy 0 10
1) "{datetime:201804131910,type:publish,title:\xe8\xaf\xb7\xe5\x81\x87\",content:\xe6\x8a\xb1\xe6\
xad\x89\xef\xbc\x8c\xe4\xbb\x8a\xe5\xa4\xa9\xe6\x9c\x89\xe4\xba\x8b\xe9\xb8\xbd\xe4\xb8\x80\
xe5\xa4\xa9}"
2) "{datetime:201804112010,type:publish,title:\xe5\xbc\x80\xe6\x92\xad\xe5\x95\xa6,content:\xe5\x8
a\xa0\xe6\xb2\xb9}"
```

Related resources

- For more information about how to eliminate potential risks and locate business performance bottlenecks, see Analyze memory usage of ApsaraDB for Redis.
- For more information about how to handle high concurrency, see ApsaraDB for Redis cluster edition.

2.14. Parse AOFs

Records of command executions and key changes are stored in append-only files (AOFs). You can parse AOFs to track these records.

Redis persistence modes

- Redis Database (RDB) snapshot mode: This mode creates point-in-time snapshots of your dataset at specified intervals. Keys and values are encoded as Redis strings and stored in RDB snapshots.
- AOF persistence mode: Similar to the binlog, AOFs keep a record of data changes that occur by writing each change to the end of the file. You can restore the entire dataset by replaying the AOF from the beginning to the end.

Details of the AOF persistence mode

A Redis client communicates with the Redis server through a protocol called REdis Serialization Protocol (RESP). RESP can serialize the following types of data:

- Simple strings:

  A string that starts with a plus sign (+) and ends with rn. Example: +OKrn.

- Error messages:

  A string that starts with a minus sign (-) and ends with rn. Example: -ERR Readonlyrn.

- Integers

  A data structure that starts with a colon (:), ends with rn, and contains an integer between the beginning and the end. Example: (:1rn).

- Large strings

  A string structure that starts with a dollar sign ($)followed by the string length (less than 512 MB) and rn, and ends with the string content and rn. Example: $0rnrn.

- Arrays

  A data structure that starts with an asterisk symbol (*), followed by array elements that are separated by rn. The above four data types can be used as array elements. Example: *1rn$4rnpingrn.

The Redis client sends an array command to the server. The server responds based on the implementation method of the command and records the responses in the AOF.

Parse AOFs

The following example shows how to parse an AOF by invoking hiredis with Python:

```
#! /usr/bin/env python

""" A Redis appendonly file parser
"""

import logging
import hiredis
import sys

if len(sys.argv) ! = 2:
print sys.argv[0], 'AOF_file'
sys.exit()
file = open(sys.argv[1])
line = file.readline()
cur_request = line
while line:
req_reader = hiredis.Reader()
req_reader.setmaxbuf(0)
req_reader.feed(cur_request)
command = req_reader.gets()
try:
if command is not False:
print command
cur_request = ''
except hiredis.ProtocolError:
print 'protocol error'
line = file.readline()
cur_request += line
file.close
```

The AOF is parsed into the following format, where you can check the operations performed on a specific key. After you obtain the following results, you can view the operations related to a specific key at any time.

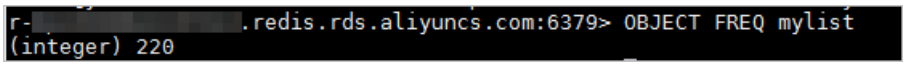
```
['PEXPIREAT', 'RedisTestLog', '1479541381558']
['SET', 'RedisTestLog', '39124268']
['PEXPIREAT', 'RedisTestLog', '1479973381559']
['HSET', 'RedisTestLogHash', 'RedisHashField', '16']
['PEXPIREAT', 'RedisTestLogHash', '1479973381561']
['SET', 'RedisTestLogString', '79146']
```

2.15. How to discover hotkeys in Redis 4.0

High performance is the most prominent feature of Redis. A robust Redis performance is crucial to ensure the service availability. A reduced Redis performance can be caused by multiple reasons. The hotkey problem is one of the most common reasons. The discovery of hotkeys is the first step to improve Redis performance. This topic describes how to use the new features of Redis 4.0 to discover the hotkeys.

Background

Redis 4.0 added two data eviction strategies: allkey-lfu and volatile-lfu. You can also run the **OBJECT** command to obtain the access frequency of a specific key, as shown in the following figure.



The native Redis client also added the *--hotkeys* option to help you discover hotkeys in your business.

> **Note** This topic describes how to discover hotkeys to optimize the performance of Redis. This topic is suitable for users who are familiar with the basic features of ApsaraDB for Redis and are seeking advanced skills. If you are not familiar with Redis, we recommend that you read Product Overview.

Prerequisites

- You have activated an ECS instance that can interconnect with the ApsaraDB for Redis instance.
- You have installed a Redis version later than Redis 4.0 on the ECS instance.

  > **Note** You can use the redis-cli tool based on these prerequisites.

- The maxmemory-policy parameter of the ApsaraDB for Redis instance is set to *volatile-lfu* or *allkeys-lfu*.

  > **Note** For more information about how to modify the parameters, see Parameter overview and configuration guide.

Procedure

1. When there is an ongoing business, use the following command to query the hotkey.

```
redis-cli -h r-***************.redis.rds.aliyuncs.com -a <password> --hotkeys
```

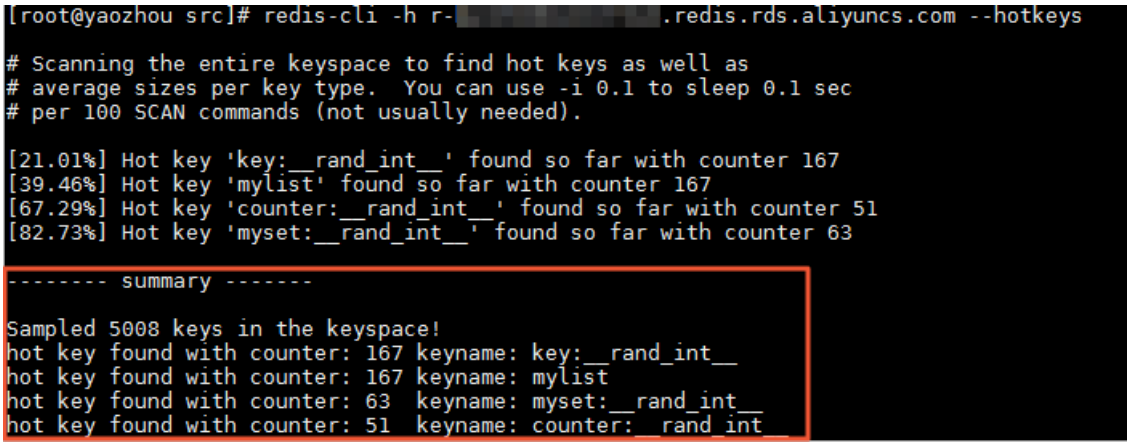
> **Note** This topic uses redis-benchmark to simulate a scenario featuring a high volume of writes.

Option descriptions

Option	Description
-h	Specifies the server hostname.
-a	Specifies the password for Redis Auth.
--hotkeys	Used to query hotkeys.

Results

The following example shows the result of running this command.



The summary part in the result is the hotkey.

2.16. Analyze memory usage of ApsaraDB for Redis

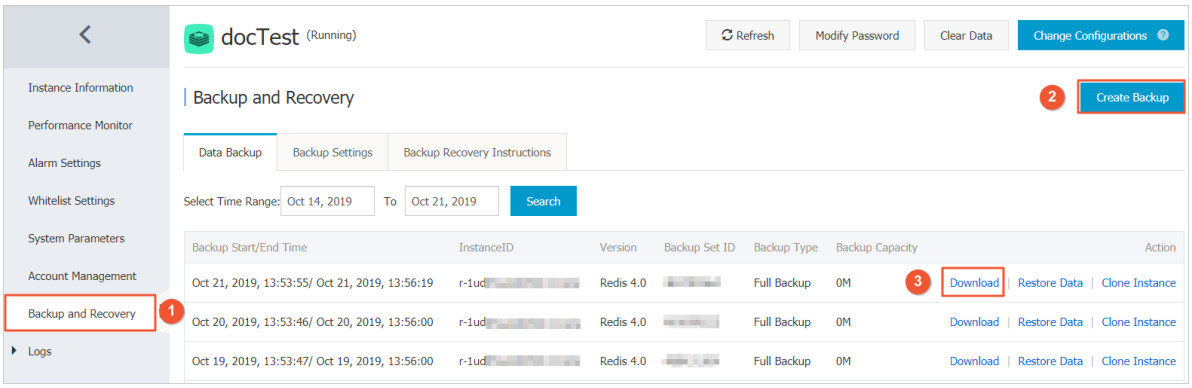
The data structure in ApsaraDB for Redis has a significant impact on service performance. If the number of big keys is large, the service performance or even service stability may deteriorate. Regular memory analysis and optimization can ensure service stability and efficiency. To avoid impacts on online services, you can run the **BGSAVE** command to generate an RDB file and use redis-rdb-tools and SQLite to analyze the file offline.

Prerequisites

- A Linux-based Elastic Compute Service (ECS) instance is created.
- The SQLite database is installed on the ECS instance.

Create an RDB file

- For an ApsaraDB for Redis instance, back up the instance and download the backup data as an RDB file in the ApsaraDB for Redis console. For more information, see Back up and restore data in the console.



- For an on-premises Redis database, run the **BGSAVE** command on the client to generate an RDB file.

Introduction to redis-rdb-tools

You need to use redis-rdb-tools to generate a memory snapshot from the RDB file that is obtained. redis-rdb-tools is a Python tool used to parse RDB files. It supports the following features:

- Generate a memory snapshot.
- Convert data in an RDB file to JSON format.
- Compare two RDB files to find their differences.

Install redis-rdb-tools

You can install redis-rdb-tools in either of the following ways:

- Install it from Python Package Index (PyPI) on the ECS instance.

```
pip install rdbtools
```

- Install it from source code on the ECS instance.

```
git clone https://github.com/sripathikrishnan/redis-rdb-tools
cd redis-rdb-tools
sudo python setup.py install
```

Use redis-rdb-tools to generate a memory snapshot

Run the following command on the ECS instance to generate a memory snapshot in CSV format:

```
rdb -c memory dump.rdb > memory.csv
```

The memory snapshot contains the following data:

- Database ID
- Data type
- Key
- Memory usage (in bytes), including the memory occupied by the key-value pair and other values

  > **Note** The memory usage is a theoretical approximation. Generally, it is slightly lower than the actual value.

- Encoding

A sample CSV file is as follows:

```
$head memory.csv
database,type,key,size_in_bytes,encoding,num_elements,len_largest_element
0,string,"orderAt:377671748",96,string,8,8,
0,string,"orderAt:413052773",96,string,8,8,
0,sortedset,"Artical:Comments:7386",81740,skiplist,479,41,
0,sortedset,"pay:id:18029",2443,ziplist,84,16,
0,string,"orderAt:452389458",96,string,8,8
```

Import the CSV file to the SQLite database

SQLite is a lightweight relational database. After importing the CSV file to the SQLite database, you can use SQL statements to analyze the data in the CSV file.

> **Note**
> - The SQLite version must be 3.16.0 or later.
> - Before importing the CSV file, delete the comma (,) at the end of each line in the CSV file.

Run the following commands to import the CSV file:

```
sqlite3 memory.db
sqlite> create table memory(database int,type varchar(128),key varchar(128),size_in_bytes int,encodin
g varchar(128),num_elements int,len_largest_element varchar(128));
sqlite>.mode csv memory
sqlite>.import memory.csv memory
```

Analyze the memory snapshot generated by redis-rdb-tools

After importing the CSV file to the SQLite database, you can use SQL statements to analyze the data in the CSV file. For example:

- Query the number of keys in the memory.

```
sqlite>select count(*) from memory;
```

- Query the total memory usage.

```
sqlite>select sum(size_in_bytes) from memory;
```

- Query the top 10 keys with the highest memory usage.

```
sqlite>select * from memory order by size_in_bytes desc limit 10;
```

- Query lists with over 1,000 elements.

```
sqlite>select * from memory where type='list' and num_elements > 1000;
```