

Alibaba Cloud

ApsaraDB for Redis Best Practices

Document Version: 20220620

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 .
<code>Courier font</code>	Courier font is used for commands	Run the <code>cd /d C:/window</code> command to enter the Windows system folder.
<i>Italic</i>	Italic formatting is used for parameters and variables.	<code>bae log list --instanceid</code> <i>Instance_ID</i>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>
{ } or {a b}	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Table of Contents

1. Development and O&M standards for ApsaraDB for Redis	06
2. Retry mechanisms for Redis clients	16
3. Usage of Lua scripts	20
4. Best Practices for Redis Enhanced Edition	28
4.1. Monitor user trajectories by using TairGIS	28
4.2. Implement high-performance distributed locks by using Ta...	30
4.3. Implement high-performance optimistic locking by using T...	35
4.4. Implement bounded counters by using TairString	38
4.5. Implement multidimensional leaderboards by using TairZse...	39
4.6. Implement fine-grained monitoring by using TairTS	44
4.7. Implement distributed leaderboards by using TairZset	47
4.8. Select users by using TairRoaring	52
5. Best Practices for All Editions	54
5.1. Migrate MySQL data to ApsaraDB for Redis	54
5.2. Rankings of online game players sorted by score	59
5.3. Correlation analysis on E-commerce store items	63
5.4. Publish and subscribe to messages	65
5.5. Pipeline	70
5.6. Process transactions	75
5.7. Discover and resolve the hotkey issue	78
5.8. ApsaraDB for Redis supports Double 11 Shopping Festival	81
5.9. Use ApsaraDB for Redis to build a business system that c...	85
5.10. Read/write splitting in Redis	88
5.11. JedisPool optimization	92
5.12. Analyze hotkeys in a specific sub-node of a cluster instan..	97
5.13. Use ApsaraDB for Redis to build a live-streaming channe...	104

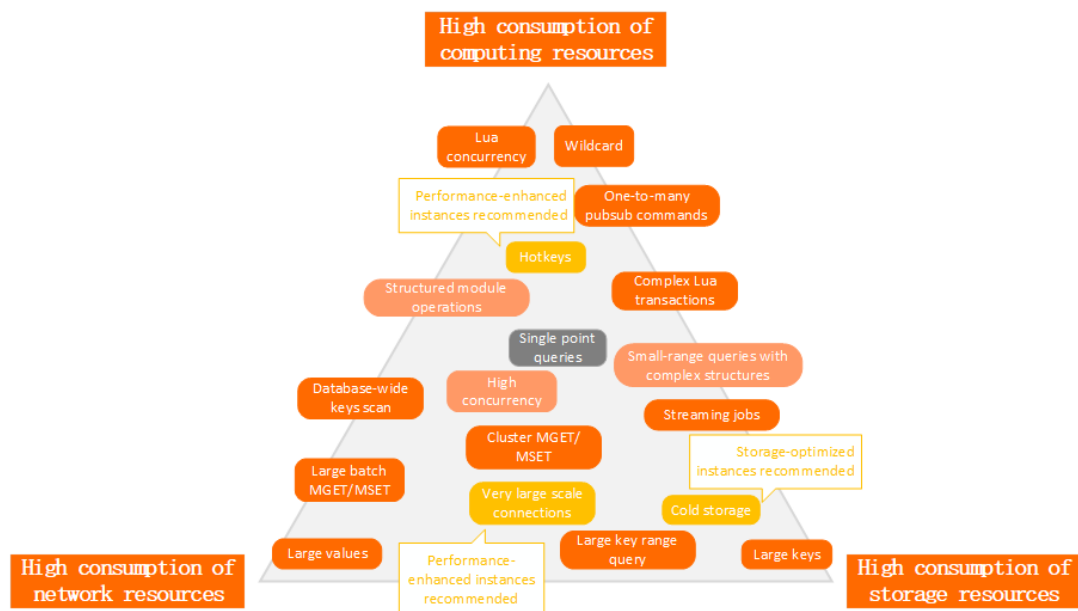
5.14. Parse AOFs	106
5.15. Query hotkeys in Redis 4.0	107
5.16. Automatically add or remove ECS instances to or from a ...-----	109

1. Development and O&M standards for ApsaraDB for Redis


ApsaraDB for Redis is a high-performance database service. This topic describes the development and O&M standards that you can follow to design a more efficient business system and better use ApsaraDB for Redis. The standards are developed by Alibaba Cloud based on years of Q&M experience and are applicable to the following scenarios: business deployment, key design, SDK usage, command usage, and O&M management.

Understand the performance limits of ApsaraDB for Redis

Performance limits of ApsaraDB for Redis



Resource type	Description
Computing resources	Wildcard characters, concurrent Lua scripts, one-to-many PubSub commands, and hotkeys consume a large amount of computing resources. For cluster instances, these items can also cause skewed requests and underutilization of data shards. For more information about cluster instances, see Cluster master-replica instances .
Storage resources	Streaming jobs and large keys consume a large amount of storage resources. For cluster instances , these items can also cause data skew and underutilization of data shards.



Resource type	Description
Network resources	<p>Database-wide scans (by running the KEYS command) and range queries of big values and large keys (by running the HGETALL command) consume a large amount of network resources and often cause thread congestion.</p> <div> Notice The high-concurrency capability of ApsaraDB for Redis does not significantly improve access performance as expected but does affect the overall performance of ApsaraDB for Redis. For example, the storage of big values in ApsaraDB for Redis does not improve access performance to a large degree.</div>

For **cluster instances**, hotkeys, large keys, or big values can also cause In a production environment, you must prevent reaching the performance limits of ApsaraDB for Redis. The following tables describe the business deployment, key design, SDK usage, command usage, and O&M management standards for ApsaraDB for Redis. These standards help you design a more efficient business system and better use the capabilities of ApsaraDB for Redis. skewed storage or skewed requests

- **Business deployment standards**
- **Key design standards**
- **SDK usage standards**
- **Command usage standards**
- **O&M management standards**

Business deployment standards


Importance	Standard	Description
------------	----------	-------------

Importance	Standard	Description
★★★★★	Determine whether the scenario is or high-speed cache in-memory databases	<ul style="list-style-type: none"> High-speed cache: We recommend that you disable append-only file (AOF) in cache-only scenarios to reduce overheads and prevent strong dependence on the data in a cache because the data may be evicted. For more information about AOF, see Disable AOF persistence. For example, after an ApsaraDB for Redis database is full, the data eviction policy is triggered to reclaim space for writing new data. For more information about the data eviction policy, see How does ApsaraDB for Redis evict data by default? The latency increases with the amount of data that is written. <div>  Notice To use the data flashback feature, you must enable AOF. For more information, see Use data flashback to restore data by point in time. </div> <ul style="list-style-type: none"> In-memory databases: We recommend that you choose Persistent memory-optimized instances of ApsaraDB for Redis Enhanced Edition (Tair). Persistent memory-optimized instances offer command-level persistence. In addition, you can monitor memory usage by configuring alerts in the databases. For more information, see Alert settings.
★★★★★	Deploy your business close to ApsaraDB for Redis instances. For example, you can deploy your business in an Elastic Compute Service (ECS) instance that resides in the same virtual private cloud (VPC) as your ApsaraDB for Redis instances.	<p>ApsaraDB for Redis is a high-performance database service. However, if you deploy your business server far from ApsaraDB for Redis instances and the business server and instances are connected over the Internet, the performance of ApsaraDB for Redis is greatly reduced due to network latency.</p> <div>  Note For cross-region deployment, you can use the geo-replication capability of Global Distributed Cache for Redis to implement geo-disaster recovery or active geo-redundancy, reduce network latency, and simplify business design. For more information, see Overview. </div>
★★★★★ ☆	Create an ApsaraDB for Redis instance for each service.	Do not use an ApsaraDB for Redis instance for different services. For example, do not use an ApsaraDB for Redis instance for both high-speed cache and in-memory database services. Otherwise, the eviction policies, slow queries, and FLUSHDB command of one service affect other services.

Importance	Standard	Description
★★★★☆	Configure appropriate eviction policies to evict expired keys.	The default expired key eviction policy is . For more information about eviction policies, see Supported parameters . volatile-lru
★★★★☆	Manage stress testing data and duration.	ApsaraDB for Redis does not delete stress testing data. To prevent impacts on your business, you must manage stress testing data and duration by yourself.


Key design standards

Importance	Standard	Description
★★★★★	Configure key values to an appropriate size. We recommend that you configure key values to a size smaller than 10 KB.	Excessively large values can cause data skew, hotkeys, high bandwidth, or high CPU utilization. You can prevent these issues from the beginning by making sure that key values are of proper size.
★★★★★	Configure proper key names that have proper length.	<ul style="list-style-type: none">Key names:<ul style="list-style-type: none">Use readable strings as key names. If you want to combine a database name, table name, and field name into a key name, we recommend that you use colons (:) to separate them. Example: <code>project:user:001</code> .Shorten key names without compromising their ability to describe your business. For example, <code>username</code> can be shortened to <code>u</code> .In ApsaraDB for Redis, braces {} are recognized as hash tags. In this case, if you use cluster instances, you must correctly use braces in key names to prevent For more information, see Keys hash tags. <div><p>Note For a cluster instance, if you want to manage multiple keys by running a command such as the RENAME command and do not use hash tags to ensure that the keys reside in the same data shard, the command cannot be run.</p></div> <p>data skew</p> <ul style="list-style-type: none">Length: We recommend that you configure key names to be no more than 128 bytes in length. The shorter, the better.

Importance	Standard	Description
★★★★★	<p>For complex data structures that support sub-keys, you must avoid including excessive sub-keys in one key. We recommend that you include less than 1,000 sub-keys in a key.</p> <div>  Note Common complex data structures include hashes, sets, Zsets, GEO structures, streams, and structures that are provided only by Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair), such as TairHash, TairBloom, and TairGIS. </div>	<p>The time complexity of some commands, such as HGETALL, is directly related to the number of sub-keys. Excessive sub-keys increase the time complexity of a command. If you frequently run commands whose time complexity is O(N) or higher, many issues occur, such as slow queries, data skew, and hotkeys.</p>
★★★★★ ☆	<p>Use the serialization method to convert values into readable structures.</p>	<p>The bytecode of a programming language may change when the version of the language changes. If you store naked objects (such as Java objects and C# objects) in ApsaraDB for Redis instances, the software stack may be difficult to upgrade. We recommend that you use the serialization method to convert values into readable structures.</p>


SDK usage standards

Importance	Standard	Description
------------	----------	-------------

Importance	Standard	Description
★★★★★	<p>Use JedisPool or JedisCluster clients to connect to ApsaraDB for Redis instances.</p> <div><p> Note We recommend that you use TairJedis clients to connect to Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair), because TairJedis clients support the encapsulation of new data structures. For more information, see TairJedis client.</p></div>	<p>If you use a single connection, the client cannot automatically reconnect to ApsaraDB for Redis instances after a connection times out. For more information about how to use JedisPool clients to connect to ApsaraDB for Redis instances, see Jedis client, JedisPool optimization, and JedisCluster.</p>
★★★★★ ☆	<p>Do not use Lettuce clients.</p>	<p>Lettuce clients do not automatically reconnect to ApsaraDB for Redis instances after multiple requests time out. If failures occur in an ApsaraDB for Redis instance and cause switchovers on proxy nodes or data shards, connections may time out and Lettuce clients cannot reconnect to the ApsaraDB for Redis instance. To prevent such risks, we recommend that you use a Jedis client to connect to ApsaraDB for Redis instances. For more information, see Jedis client.</p>
★★★★★ ☆	<p>Design proper fault tolerance mechanisms for your clients.</p>	<p>Network fluctuations and high usage of resources may cause connection timeouts or slow queries. To prevent these risks, you must design proper fault tolerance mechanisms for your clients.</p>
★★★★★ ☆	<p>Set longer retry intervals for your clients.</p>	<p>If retry intervals are shorter than required, such as shorter than 200 milliseconds, a large number of retries may occur in a short period of time. This can result in a service avalanche. For more information, see Retry mechanisms for Redis clients.</p>

Command usage standards


Importance	Standard	Description
★★★★★	Avoid range queries, such as those by running the KEYS * command. Instead, use multiple point queries or run the SCAN command to reduce latency.	Range queries may cause service interruptions, slow queries, or congestion.
★★★★★	Use extended data structures to perform complex operations. For more information, see Integration with multiple Redis modules . Do not use Lua scripts.	Lua scripts consume a large amount of computing and memory resources and do not support multi-threading acceleration. Overly complex or improper Lua scripts may result in the exhaustion of resources.
★★★★★ ☆	Use pipelines to reduce the round-trip time (RTT) of data.	<p>If you want to send multiple commands to a server and your client does not depend on each response from the server, you can use a pipeline to send the commands at a time. Take note of the following items when you use pipelines:</p> <ul style="list-style-type: none"> • A client that uses pipelines exclusively connects to a server. We recommend that you establish a dedicated connection for pipeline operations to separate them from regular operations. • Each pipeline must contain a proper number of commands. We recommend that you use each pipeline to send no more than 100 commands.
★★★★★ ☆	Use transaction commands. For more information, see Transaction command group .	<p>When you use transaction commands, take note of the following limits:</p> <ul style="list-style-type: none"> • Transactions cannot be rolled back. • If you want to run transaction commands on cluster instances, use hash tags to ensure that the keys to be managed are distributed to the same hash slot. You must also prevent skewed storage that hash tags may cause. • Do not encapsulate transaction commands in Lua scripts, because the compilation and loading of these commands consume a large amount of computing resources.

Importance	Standard	Description
★★★★☆	Do not use the Pub and Sub command group to perform a large number of message distribution tasks. For more information, see Pub and Sub command group .	<p>The Pub and Sub command group does not support data persistence or acknowledge mechanisms that ensure data reliability. We recommend that you do not use Pub or Sub commands to perform a large number of message distribution tasks. For example, if you use these commands to distribute a message whose size is greater than 1 KB to more than 100 subscriber clients, server resources may be exhausted and subscriber clients may not receive the message.</p> <div><p> Note To improve performance and balance, ApsaraDB for Redis is optimized for Pub and Sub commands. In cluster instances, proxy nodes calculate the hash values of commands based on channel names and allocate commands to corresponding data nodes.</p></div>

O&M management standards

Importance	Standard	Description
★★★★★	Understand the impacts of different instance management operations.	Configuration changes or restarts affect the state of an ApsaraDB for Redis instance. For example, transient connections may occur for the instance. Before you perform the preceding operations, make sure that you understand the impacts. For more information, see Instance states and impacts .
★★★★★	Verify the error handling capabilities or disaster recovery logic of a client.	ApsaraDB for Redis can monitor the health status of nodes. If a master node in an instance becomes unavailable, ApsaraDB for Redis automatically triggers a master-replica switchover. The roles of master and replica nodes are switched over to ensure the high availability of the instance. Before a client is officially released, we recommend that you manually trigger the master-replica switchover. This can help you verify the error handling capabilities or disaster recovery logic of the client. For more information, see Manually switch workloads from a master node to a replica node .
★★★★★	Disable time-consuming or high-risk commands.	In a production environment, abuse of commands may cause problems. For example, the FLUSHALL command can delete all data. The KEYS command may cause network congestion. To improve the stability and efficiency of services, you can disable these commands to minimize risks. For more information, see Disable high-risk commands .


Importance	Standard	Description
★★★★☆	Handle pending events at the earliest opportunity.	To enhance user experience and provide improved service performance and stability, Alibaba Cloud occasionally generates pending events to upgrade the hardware and software of specific servers or replace network facilities. For example, a pending event is generated when the minor version of databases needs to be updated. After you receive an event notification from Alibaba Cloud, you can check the impacts of the event and change the scheduled time of the event to meet your business requirements. For more information, see Query and manage pending events .
★★★★☆	Configure alerts for core metrics and better monitor the status of your instances.	Configure alerts for core metrics such as CPU utilization, memory usage, and bandwidth usage to monitor the status of your instances in real time. For more information, see Alert settings .
★★★★☆	Use O&M features provided by ApsaraDB for Redis to check the status of instances or troubleshoot resource usage exceptions on a regular basis.	<ul style="list-style-type: none"> • Use slow logs to troubleshoot timeout issues: Slow logs help you locate slow queries and the IP addresses of the clients that send the query requests. Slow logs provide a reliable basis for addressing timeout issues. • View monitoring data: ApsaraDB for Redis supports a variety of performance metrics. These metrics allow you to gain insights into the status of ApsaraDB for Redis instances and troubleshoot issues at the earliest opportunity. • Create a diagnostic report: Diagnostic reports help you evaluate the status of ApsaraDB for Redis instances, such as performance level, skewed requests, and slow logs. Diagnostic reports also help you identify exceptions on ApsaraDB for Redis instances. • Use the offline key analysis feature to display details about big keys: You can use the offline key analysis feature to identify large keys of ApsaraDB for Redis instances. You can also learn the memory usage, distribution, and expiration time of large keys. • Use the real-time key statistics feature: The real-time key statistics feature helps you identify hotkeys of ApsaraDB for Redis instances and allows you to further optimize your databases.

Importance	Standard	Description
★★★★☆ ☆	Enable the audit log feature and evaluate audit logs.	<p>After you enable the audit log feature, the audit statistics about write operations are recorded. ApsaraDB for Redis also allows you to query, analyze online, and export audit logs. These features help you monitor the security and performance of your ApsaraDB for Redis instances. For more information, see Enable the new audit log feature.</p> <div> Notice After you enable the audit log feature, the performance of ApsaraDB for Redis instances may degrade by 5% to 15%. The actual performance degradation varies based on the number of write operations or audit operations. If your business expects a large number of write operations, we recommend that you enable the audit log feature only when you perform O&M operations, such as troubleshooting. This helps you prevent performance degradation.</div>

2. Retry mechanisms for Redis clients

Due to network and running environments, applications may encounter temporary faults, such as transient network jitter, temporary unavailability of services, and timeout caused by busy services. You can configure automatic retry mechanisms to avoid temporary failures and ensure successful operations.

Causes for temporary failures

Cause	Description
The high availability mechanism triggered	<p>ApsaraDB for Redis can monitor the health status of nodes. If a master node in an instance fails, ApsaraDB for Redis automatically triggers a master-replica switchover. The roles of master and replica nodes are switched to ensure high availability of the instance. At this time, the client may encounter the following temporary failures:</p> <ul style="list-style-type: none">• Transient connections in seconds• Read-only state within 30 seconds (to avoid potential risks of data loss and dual writes caused by primary/secondary failover). <div><p> Note For more information, see Causes and impacts of master-replica switchovers.</p></div>
Request jams caused by slow queries	<p>Request jams and slow queries occur when operations with time complexity of $O(n)$ are executed. In this case, other requests initiated by the client may experience temporary failures.</p>
Complex network environments	<p>Complex network environments between the client and Redis server may cause problems such as occasional network jitter and data retransmission. In this case, requests initiated by the client may temporarily fail.</p>

Recommended retry rules


Retry rule	Description
Only retry idempotent operations	<p>A timeout event may occur at the following phases:</p> <ul style="list-style-type: none">• A command is sent by the client but has not reached ApsaraDB for Redis.• The command reaches ApsaraDB for Redis, but the execution times out.• The command is executed on ApsaraDB for Redis, but a timeout event occurs when the result is returned to the client. <p>A retry may cause an operation to be repeated on ApsaraDB for Redis. Therefore, not all operations are suitable for a retry mechanism. We recommend that you retry only idempotent operations, such as SET commands. After you run the SET a b command multiple times, the value of a can only be b or failed executions. When you run the LPUSH mylist a command which is not idempotent multiple times, mylist may contain multiple a elements.</p>

Retry rule	Description
------------	-------------

Appropriate number and interval of retries	<p>Adjust the number and interval of retries based on business requirements and actual scenarios. Otherwise, the following issues may occur:</p> <ul style="list-style-type: none">• If the number of retries is very low or the interval is very long, the application may fail because operations cannot be performed.• If the number of retries is very high or the interval is very short, the application may consume more system resources and request jams may cause the server to fail. <p>Common retry interval methods include immediate retry, fixed-time retry, exponentially increasing time retry, and random retry.</p>
Avoid retry nesting	Retry nesting may cause repeated or even unlimited retries.
Record retry exceptions and generate failure reports	During the retry process, we recommend that you configure the system to generate retry logs at the WARN level and only when the retry fails.

Jedis client

- In JedisPool mode, Jedis does not provide retry mechanisms. We recommend that you use [TairJedis](#) which is based on Jedis encapsulation and encapsulates the Jedis retry class to quickly implement retry policies.

 **Note** If [Performance-enhanced instances](#) instances of ApsaraDB for Redis Enhanced Edition (Tair) are used, this client allows you to use the data structures developed by Alibaba Cloud. For more information about the data structures, see [Commands supported by extended data structures of ApsaraDB for Redis Enhanced Edition \(Tair\)](#).

- In JedisCluster mode, you can specify the maxAttempts parameter to define the number of retries in case of a failure. The default value is 5.

An example of retry settings on the Jedis client:

```
//Add a dependency.
<dependency>
  <groupId>com.aliyun.tair</groupId>
  <artifactId>alibabacloud-tairjedis-sdk</artifactId>
  <version>Enter the latest version number</version>
</dependency>
//Set the key value command to automatically retry five times and the maximum overall retry
period to 10 seconds. For each retry, the system waits for a while between class indexes. If
the command fails, an exception is thrown.
int maxRetries = 5; //Specify the maximum number of retries.
Duration maxTotalRetriesDuration = Duration.ofSeconds(10); //Specify the maximum retry peri
od. Unit: seconds.
try {
    String ret = new JedisRetryCommand<String>(jedisPool, maxRetries, maxTotalRetriesDurati
on) {
        @Override
        public String execute(Jedis connection) {
            return connection.set("key", "value");
        }
    }.runWithRetries();
} catch (JedisException e) {
    // Indicates that maxRetries attempts have been made or the maximum query time maxTota
lRetriesDuration reached.
    e.printStackTrace();
}
```

Redisson client

The Redisson client provides two parameters to control the retry logic:

- **retryAttempts**: the number of retries. Default value: 3.
- **retryInterval**: the retry interval. Default value: 1,500 milliseconds.

An example of retry settings on the Jedis client:

```
Config config = new Config();
config.useSingleServer()
    .setTimeout(1000)
    .setRetryAttempts(3)
    .setRetryInterval(1500) //ms
    .setAddress("redis://127.0.0.1:6379");
RedissonClient connect = Redisson.create(config);
```

StackExchange.Redis client

The StackExchange.Redis client only supports connection retries. An example of retry settings on the StackExchange.Redis client:

```
var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");
```

 **Note** For more information about API-level retry policies, see [Polly](#).

Lettuce client

Although the Lettuce client does not provide parameters for retries after a command times out, you can use the following parameters to implement retry policies:

- at-most-once execution: The command can be executed once at most. If the client is disconnected and then reconnected, the command may be lost.
- at-least-once execution (default): A minimum of one successful command execution is ensured. This means that multiple attempts may be made to ensure a successful execution. If this method is used and a primary/secondary switchover for an ApsaraDB for Redis instance occurs, a large number of retry commands may be accumulated on the client. After the primary/secondary switchover is complete, the CPU utilization of the ApsaraDB for Redis instance may surge.

 **Note** For more information, see [Client Options](#) and [Command execution reliability](#).

An example of retry settings on the Lettuce client:

```
clientOptions.setAutoReconnect() ? Reliability.AT_LEAST_ONCE : Reliability.AT_MOST_ONCE;
```

Related information

- [Use a client to connect to an ApsaraDB for Redis instance](#)
- [Use a client to connect to an ApsaraDB for Redis instance that has SSL encryption enabled](#)


3.Usage of Lua scripts

ApsaraDB for Redis instances support commands related to Lua scripts. Lua scripts can be used to efficiently process check-and-set (CAS) commands. This improves the performance of ApsaraDB for Redis and simplifies the implementation of features that used to be difficult to implement. This topic describes the syntax and usage of Lua scripts in ApsaraDB for Redis.

Precautions

Commands related to Lua scripts cannot be used in the Data Management (DMS) console. For more information about DMS, see [Overview](#). You can use a client or redis-cli to connect to ApsaraDB for Redis instances and use Lua scripts.

Basic syntax

Command	Syntax	Description
EVAL	<pre>EVAL script numkeys [key [key ...]] [arg [arg ...]]</pre>	<p>Executes a specified script that takes parameters and returns the output.</p> <p>Parameter description:</p> <ul style="list-style-type: none"> script: the Lua script. numkeys: the number of arguments in the KEYS array. The number is a non-negative integer. KEYS[]: the Redis keys that you want to pass to the script as arguments. ARGV[]: the additional arguments that you want to pass to the script. The indexes of the KEYS[] and ARGV[] parameters start from 1. <div> <p> Note</p> <ul style="list-style-type: none"> The EVAL command loads a script into the script cache of ApsaraDB for Redis in a similar way as the SCRIPT LOAD command. Mixed use or misuse of the KEYS[] and ARGV[] parameters may cause ApsaraDB for Redis instances to run not as expected, especially for ApsaraDB for Redis cluster instances. For more information, see Limits on Lua scripts in cluster instances. </div>
EVALSHA	<pre>EVALSHA sha1 numkeys key [key ...] arg [arg ...]</pre>	<p>Evaluates a cached script by its SHA1 digest and runs the script.</p> <p>If the script is not cached in ApsaraDB for Redis when you use the EVALSHA command, ApsaraDB for Redis returns the NOSCRIPT error. Cache the script in ApsaraDB for Redis by using the EVAL or SCRIPT LOAD command and try again. For more information, see Handle the NOSCRIPT error.</p>
SCRIPT LOAD	<pre>SCRIPT LOAD script</pre>	<p>Caches a specified script in ApsaraDB for Redis and returns the SHA1 digest of the script.</p>

Command	Syntax	Description
SCRIPT EXISTS	<pre>SCRIPT EXISTS script [script ...]</pre>	Returns information about the existence of one or more scripts in the script cache by using their corresponding SHA1 digests. If a specified script exists, a value of 1 is returned. Otherwise, a value of 0 is returned.
SCRIPT KILL	<pre>SCRIPT KILL</pre>	Terminates a Lua script in execution.
SCRIPT FLUSH	<pre>SCRIPT FLUSH</pre>	Removes all the Lua scripts from the script cache in the Redis server.

For more information about Redis commands, visit the [Redis official website](#).

Some Redis commands are demonstrated in the following examples. Before the following commands are run, the `SET foo value_test` command is run.

- Sample EVAL command:

```
EVAL "return redis.call('GET', KEYS[1])" 1 foo
```

Sample output:

```
"value_test"
```

- Sample SCRIPT LOAD command:

```
SCRIPT LOAD "return redis.call('GET', KEYS[1])"
```

Sample output:

```
"620cd258c2c9c88c9d10db67812ccf663d96bdc6"
```

- Sample EVALSHA command:

```
EVALSHA 620cd258c2c9c88c9d10db67812ccf663d96bdc6 1 foo
```

Sample output:

```
"value_test"
```

- Sample SCRIPT EXISTS command:

```
SCRIPT EXISTS 620cd258c2c9c88c9d10db67812ccf663d96bdc6 ffffffffffffffffffffffffffffffffff
ffffffff
```

Sample output:

```
1) (integer) 1
2) (integer) 0
```

Optimize memory and network overheads

Issue:

A large number of scripts that serve the same purposes are cached in ApsaraDB for Redis. These scripts take up large amounts of memory and may cause the out of memory (OOM) error. Example of invalid usage:

```
EVAL "return redis.call('set', 'k1', 'v1')" 0
EVAL "return redis.call('set', 'k2', 'v2')" 0
```

Solution:

- Do not pass parameters to Lua scripts as constants to reduce memory usage.

```
# The following commands serve the same purposes as the preceding sample commands but cache scripts only once.
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 k1 v1
EVAL "return redis.call('set', KEYS[1], ARGV[1])" 1 k2 v2
```

- Use the following command syntax to reduce memory and network overheads:

```
SCRIPT LOAD "return redis.call('set', KEYS[1], ARGV[1])" # After this command is run,
the following output is returned: "55b22c0d0cedf3866879ce7c854970626dcef0c3"
EVALSHA 55b22c0d0cedf3866879ce7c854970626dcef0c3 1 k1 v1
EVALSHA 55b22c0d0cedf3866879ce7c854970626dcef0c3 1 k2 v2
```

Flush the Lua script cache


Issue:

Used memory of an ApsaraDB for Redis instance may be higher than expected because the Lua script cache takes up memory of the instance. When the used memory of the instance approaches or exceeds the upper limit and Lua scripts are used, the OOM error is returned. Error example:

```
-OOM command not allowed when used memory > 'maxmemory'.
```

Solution:

Flush the Lua script cache by running the `SCRIPT FLUSH` command on the client. Different from the `FLUSHALL` command, the `SCRIPT FLUSH` command is synchronous. If ApsaraDB for Redis caches an large number of Lua scripts, the `SCRIPT FLUSH` command can block ApsaraDB for Redis for an extended period of time and an instance may become unavailable. Proceed with caution. We recommend that you perform this operation during off-peak hours.

 **Note** If you click **Clear Data** in the ApsaraDB for Redis console, data can be cleared but the Lua script cache cannot be flushed.

Do not write large Lua scripts that may take up excessive amount of memory. Moreover, do not write large amounts of data to Lua scripts. Otherwise, memory usage significantly increases and the OOM error may even occur. To reduce memory usage, we recommend that you enable data eviction by using the volatile-lru policy. By default, data eviction is enabled in ApsaraDB for Redis. For more information about data eviction, see [How does ApsaraDB for Redis evict data by default?](#) However, ApsaraDB for Redis does not evict the Lua script cache regardless of whether data eviction is enabled.

Handle the NOSCRIPT error

Issue:


If the script is not cached in ApsaraDB for Redis when you use the EVALSHA command, ApsaraDB for Redis returns the NOSCRIPT error. Error example:

```
(error) NOSCRIPT No matching script. Please use EVAL.
```

Solution:

Run the EVAL or SCRIPT LOAD command to cache the script in ApsaraDB for Redis and try again. In some scenarios such as instance migrations and configuration changes, ApsaraDB for Redis still flushes the Lua script cache because ApsaraDB for Redis cannot ensure the persistence and replicability of Lua scripts. For this reason, your client must have the ability to handle this error. For more information, see [Caching, persistence, and replication of scripts](#).

The following sample Python code shows a method for handling the NOSCRIPT error. The sample code prepends strings by using Lua scripts.

 **Note** You can also use redis-py to handle this error. redis-py provides the Script class that encapsulates the judgement logic for Lua scripts of ApsaraDB for Redis, such as a catch statement for the NOSCRIPT error.

```

import redis
import hashlib
# strin indicates a string in Lua scripts. This function returns the sha1 value of strin in
the string format.
def calcSha1(strin):
    sha1_obj = hashlib.sha1()
    sha1_obj.update(strin.encode('utf-8'))
    sha1_val = sha1_obj.hexdigest()
    return sha1_val
class MyRedis(redis.Redis):
    def __init__(self, host="localhost", port=6379, password=None, decode_responses=False):
        redis.Redis.__init__(self, host=host, port=port, password=password, decode_response
s=decode_responses)
    def prepend_inLua(self, key, value):
        script_content = """\
        local suffix = redis.call("get", KEYS[1])
        local prefix = ARGV[1]
        local new_value = prefix..suffix
        return redis.call("set", KEYS[1], new_value)
        """
        script_sha1 = calcSha1(script_content)
        if self.script_exists(script_sha1)[0] == True:      # Check whether ApsaraDB for Re
dis already caches the script.
            return self.evalsha(script_sha1, 1, key, value) # If the script is already cach
ed, the EVALSHA command is used to run the script.
        else:
            return self.eval(script_content, 1, key, value) # Otherwise, use the EVAL comma
nd to run the script. Note that the EVAL command can cache scripts in ApsaraDB for Redis. A
nother method is to use the SCRIPT LOAD and EVALSHA commands.
r = MyRedis(host="r-*****.redis.rds.aliyuncs.com", password="***:***", port=6379, decode_r
esponses=True)
print(r.prepend_inLua("k", "v"))
print(r.get("k"))

```

Handle timeouts of Lua scripts

- Issue:

Slow Lua requests may block ApsaraDB for Redis because Lua script execution is atomic in ApsaraDB for Redis. One Lua script can block ApsaraDB for Redis for up to 5 seconds when the script is being executed. After 5 seconds, ApsaraDB for Redis returns the BUSY error for other commands until the script execution is complete.

```
BUSY Redis is busy running a script. You can only call SCRIPT KILL or SHUTDOWN NOSAVE.
```

Solution:

Run the SCRIPT KILL command to terminate the Lua script or wait until the Lua script execution is complete.

 **Note**

- During the first 5 seconds when a slow Lua script is being executed, the SCRIPT KILL command does not take effect because ApsaraDB for Redis is being blocked.
- To prevent ApsaraDB for Redis from being blocked for an extended period of time, we recommend that you estimate the amount of time required to execute a Lua script when you write the Lua script, check for infinite loop, and split the Lua script if necessary.

● **Issue:**

If a Lua script has already run write commands against the dataset, the SCRIPT KILL command does not take effect. Error example:

```
(error) UNKILLABLE Sorry the script already executed write commands against the dataset.
You can either wait the script termination or kill the server in a hard way using the SHUTDOWN NOSAVE command.
```

Solution:

On the **Instances** page of the ApsaraDB for Redis console, click **restart** in the Actions column corresponding to the instance. If the issue persists, .

Caching, persistence, and replication of scripts

Issue:


ApsaraDB for Redis keeps caching the Lua scripts in an instance that have been executed if the instance is not restarted or the SCRIPT FLUSH command is not run for the instance. However, ApsaraDB for Redis cannot ensure the persistence of Lua scripts or the synchronization of Lua scripts from the current node to other nodes in scenarios such as instance migrations, configuration changes, version upgrades, and instance switchovers.

Solution:

Store all Lua scripts in your on-premise device. Recache the Lua scripts in ApsaraDB for Redis by using the EVAL or SCRIPT LOAD command if necessary to prevent the NOSCRIPT error from occurring when Lua scripts are cleared during an instance restart or a high availability (HA) switchover.

Limits on Lua scripts in cluster instances

Redis clusters impose limits on the usage of Lua scripts. The following additional limits exist for ApsaraDB for Redis cluster instances:

 **Note** If an error message indicating that the EVAL command fails to run is returned, such as `ERR command eval not support for normal user`, update the minor version of the ApsaraDB for Redis instance to the latest version. For more information, see [Update the minor version](#).

- All keys that a script uses must be allocated to the same hash slot. Otherwise, the following error message is returned:

```
-ERR eval/evalsha command keys must be in same slot\r\n
```

 **Note** You can run the CLUSTER KEYSLOT command to obtain the hash slot of a key.

- A Lua script may not be stored in other nodes when you run the `SCRIPT LOAD` command on one node.
- The following Pub/Sub commands are not supported: `PSUBSCRIBE`, `PUBSUB`, `PUBLISH`, `PUNSUBSCRIBE`, `SUBSCRIBE`, and `UNSUBSCRIBE`.
- The `UNPACK` function is not supported.

If all the operations can be performed in the same hash slot and you want to break through the limits that the cluster architecture imposes on your Lua script, you can set the `script_check_enable` parameter to `0` in the ApsaraDB for Redis console. This way, the system does not check your Lua script at the backend. In this case, you still need to specify at least one key in the `KEYS` array so that proxy nodes can route commands in the Lua script. If you cannot make sure that all the operations are performed in the same hash slot, an error is returned. For more information, see [Modify parameters of an instance](#).

Additional limits on the proxy mode

- Lua scripts use the `redis.call` or `redis.pcall` function to run Redis commands. For Redis commands, all keys must be specified by using the `KEYS` array, which cannot be replaced by Lua variables. If you do not use the `KEYS` array to specify the keys, the following error message is returned:

```
-ERR bad lua script for redis cluster, all the keys that the script uses should be passed using the KEYS array\r\n
```

Examples of valid and invalid usage:

```
# The following two commands must be run in advance.
SET foo foo_value
SET {foo}bar bar_value
# Example of valid usage
EVAL "return redis.call('mget', KEYS[1], KEYS[2])" 2 foo {foo}bar
# Examples of invalid usage
EVAL "return redis.call('mget', KEYS[1], '{foo}bar')" 1 foo
EVAL "return redis.call('mget', KEYS[1], ARGV[1])" 1 foo {foo}bar
```


- Keys must be included in all the commands that you want to run. Otherwise, the following error message is returned:

```
-ERR for redis cluster, eval/evalsha number of keys can't be negative or zero\r\n
```

Examples of valid and invalid usage:

```
# Example of valid usage
EVAL "return redis.call('get', KEYS[1])" 1 foo
# Example of invalid usage
EVAL "return redis.call('get', 'foo')" 0
```

- You cannot run the `EVAL`, `EVALSHA`, or `SCRIPT` command in the `MULTI` or `EXEC` transactions.

 **Note** If you want to use the features that are unavailable for the proxy mode, you can enable the direct connection mode for an ApsaraDB for Redis cluster instance. However, migrations or configuration changes fail for cluster instances when Lua scripts that do not conform to the requirements of the proxy mode are executed in direct connection mode. This is because cluster instances rely on proxy nodes to migrate data during migrations and configuration changes. To prevent subsequent migrations and configuration changes based on Lua scripts from failing, we recommend that you conform to the usage limits of Lua scripts in proxy mode when you use Lua scripts in direct connection mode.

4. Best Practices for Redis Enhanced Edition

4.1. Monitor user trajectories by using TairGIS

This topic describes how to use the TairGIS data structure provided by ApsaraDB for Redis Enhanced Edition (Tair) to monitor user trajectories based on points, lines, and planes.

Background information

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

The COVID-19 pandemic that emerged in 2020 has posed grave health threats to mankind and put countries around the world on pause. To control the spread of the COVID-19 pandemic, China has mobilized the whole country and galvanized the people into a nationwide response. Gradually, cities across China begin to recover from the COVID-19 pandemic. Employees go back to work, enterprises resume production, and students go back to schools. While the spread of the pandemic in China is under control, many other countries are still fighting to flatten the curve of COVID-19 cases. Epidemic prevention and control remains challenging. 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 provided by open source Redis. You can use these native Redis GEO commands to describe location data. However, these commands offer limited support for LBS applications because these commands provide only limited precision and features. TairGIS commands available for performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provide more features than the native Redis GEO commands. For more information, see [TairGIS commands](#).

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 these users. You can use the following two methods to obtain the location data:

- Use the Global Positioning System (GPS) service on user mobile phones. 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 users have been to high-risk areas such as those with epidemic outbreaks. In most cases, you do not need to store the historical trajectory data of users. Alerts can be sent 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 as TairGIS data. You can use points, lines, or polygons to indicate user trajectories based on WKT, and store the points, lines, or polygons as TairGIS data. Then, you can run TairGIS commands to query the intersections between the user trajectories and these high-risk areas to determine whether a user has been to these high-risk areas.

 **Note** WKT is a text markup language for representing vector geometry objects on a map, spatial reference systems of spatial objects, and transformations between spatial reference systems.

The methods to process location data vary 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 `GIS.CONTAINS` command to check whether the user location is in a high-risk area. For more information about the `GIS.CONTAINS` command, see [GIS.CONTAINS](#). 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 approaching a high-risk area. If the user approaches a high-risk area, alerts are sent. For more information about the `GIS.INTERSECTS` command, see [GIS.INTERSECTS](#).


You can use WKT to describe the GPS data of a user as a point, such as `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 demonstrates 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 the TairGIS data structure.
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, such as `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. For more information about the `GIS.INTERSECTS` command, see [GIS.INTERSECTS](#). Sample code:

```
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 structure.
GIS.INTERSECTS your_province 'POLYGON ((10 22, 30 45, 16 53, 10 22))'
```

 **Note** For more information about TairGIS commands, see [TairGIS](#).

Summary

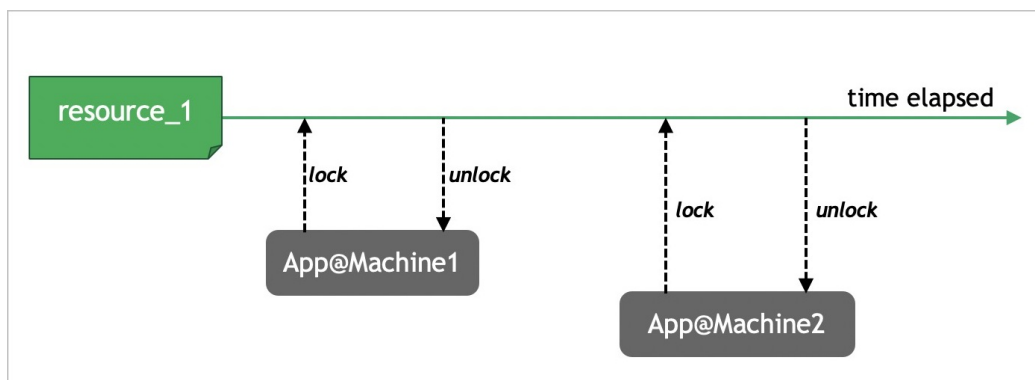
Performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provide the TairGIS data structure. TairGIS provides an easy method for you to store and process geographic data by using LBS applications. TairGIS can also deliver high performance in high-concurrency scenarios.

4.2. Implement high-performance distributed locks by using TairString

Distributed locks are one of the most widely adopted features in large applications. You can implement distributed locks based on Redis by using a variety of methods. This topic describes the common methods to implement distributed locks and the best practices for implementing distributed locks by using ApsaraDB for Redis Enhanced Edition (Tair). These best practices are developed based on the accumulated experience of Alibaba Group in using ApsaraDB for Redis Enhanced Edition (Tair) and distributed locks.

Distributed locks and their use scenarios

If a specific resource needs to be concurrently accessed by multiple threads in the same process during application development, you can use mutexes (also known as mutual exclusion locks) and read/write locks. If a specific resource needs to be concurrently accessed by multiple processes on the same host, you can use interprocess synchronization primitives such as semaphores, pipelines, and shared memory. However, if a specific resource needs to be concurrently accessed by multiple hosts, you must use distributed locks. Distributed locks are mutual exclusion locks that have global presence. You can apply distributed locks to resources in distributed systems to prevent logical failures that may be caused by resource contention.



Features of distributed locks

- Mutually exclusive

At any given moment, only one client can hold a lock.

- Deadlock-free

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

- Consistent

Switchovers in ApsaraDB for Redis may be triggered by external or internal errors. External errors include hardware failures and network exceptions, and internal errors include slow queries and system defects. After a switchover is triggered, a replica node is promoted to be the new master node to ensure high availability (HA). In this scenario, if your business has high requirements for mutual exclusion, locks must remain the same after a switchover.

Implement distributed locks based on open source Redis

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

- Acquire a lock

In Redis, you need to only run the **SET** command to acquire a lock. The following section provides a command example and describes the parameters or options used in the command:

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

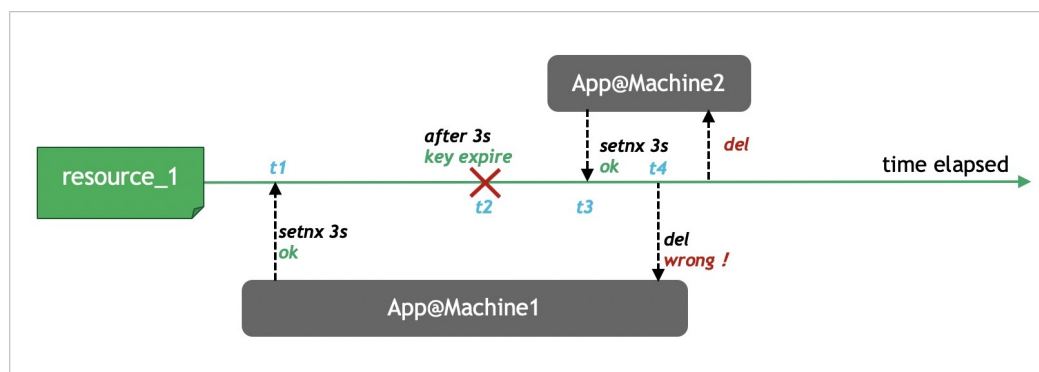
Parameters or options

Parameter/option	Description
resource_1	The key of the distributed lock. If the key exists, the corresponding resource is locked and cannot be accessed by other clients.
random_value	A random string. The value must be unique across clients.
EX	The validity period of the key. Unit: seconds. You can also use the PX option to set a validity period accurate to the millisecond.
NX	Specifies to set the key only if the key does not exist in Redis.

In the sample code, the validity period of the resource_1 key is set to 5 seconds. If the client does not release the key, the key expires after 5 seconds and the lock is reclaimed by the system. Then, 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.



- At the t1 time point, the key of the distributed lock is resource_1 for application 1, and the validity period for the resource_1 key is set to 3 seconds.

- ii. Application 1 remains blocked for more than 3 seconds due to specific reasons, such as long response time. 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 being blocked 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 needs to 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 first run the `GET` command to check whether the lock was set by itself. In most cases, a client uses the following Lua script in Redis to release the lock that was 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 lease time of the lock, the client must renew the lock. 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)

If your instance is a performance-enhanced or persistent memory-optimized instance of the ApsaraDB for Redis Enhanced Edition (Tair), you can run string-enhanced commands to implement distributed locks without the help of Lua scripts. For more information about performance-enhanced and persistent memory-optimized instances, see [Performance-enhanced instances](#) and [Persistent memory-optimized instances](#).

- Acquire a lock

The method to acquire a lock in ApsaraDB for Redis Enhanced Edition (Tair) is the same as that used in open source Redis. The method is to run the `SET` command. Sample command:

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

- Release a lock


The `CAD` command of ApsaraDB for Redis Enhanced Edition (Tair) provides an elegant and efficient way for you to release a lock. For more information about the `CAD` command, see [CAD](#). Sample command:

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

- Renew a lock

You can run the CAS command to renew a lock. For more information, see [CAS](#). Sample command:

```
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 expireTime) {
    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
expireTime) {
    jedis.getClient().sendCommand(TairCommand.CAS,resourceKey,randomValue,randomValue,"EX
",String.valueOf(expireTime));
    Long ret = jedis.getClient().getIntegerReply();
    return 1 == ret;
}
```

Methods to ensure lock consistency

The replication between a master node and a replica node is asynchronous. If a master node crashes after data changes are written to the master node and an HA switchover is triggered, the data changes in the buffer may not be replicated to the new master node. This results in data inconsistency. Note that the new master node is the original replica node. If the lost data is related to a distributed lock, the locking mechanism becomes faulty and service exceptions occur. This section describes three methods that you can use to ensure lock consistency.

- Use the **Redlock** algorithm

The Redlock algorithm is proposed by the founders of the open source Redis project to ensure lock consistency. The Redlock algorithm is based on the calculation of probabilities. A single master-replica Redis instance may lose a lock during an HA switchover, and the probability is $k\%$. If you use the Redlock algorithm to implement distributed locks, you can calculate the probability at which N independent master-replica Redis instances all lose locks at the same time based on the following formula: Probability of losing locks = $(k\%)^N$. The more nodes an instance has, the higher the consistency is. Given the high stability of Redis, the probability can meet the service requirements.

Note When you use the Redlock algorithm, you do not need to ensure that all the locks in N Redis instances take effect at the same time. In most cases, the Redlock algorithm 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 .

The Redlock algorithm has the following issues:

- A client takes a long time to acquire or release a lock.
 - You cannot use the Redlock algorithm in cluster or standard master-replica instances.
 - The Redlock algorithm consumes large amounts of resources. To use the Redlock algorithm, you must create multiple independent ApsaraDB for Redis instances or self-managed Redis instances.
- Use the **WAIT** command

The **WAIT** command of Redis blocks the current client until all the previous write commands are synchronized from a master node to a specific number of replica nodes. In the **WAIT** command, you can specify a timeout period in milliseconds. The **WAIT** command is used in ApsaraDB for Redis to ensure the consistency of distributed locks. Sample command:

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

When you run the **WAIT** command, the client will only continue to perform other operations in two scenarios after the client acquires a lock. One scenario is that data is synchronized to the replica nodes. The other scenario is that the timeout period is reached. In this example, the timeout period is 5,000 milliseconds. If the output of the **WAIT** command is 1, data is synchronized between the master node and the replica nodes. In this case, data consistency is ensured. The **WAIT** command is far more cost-effective than the Redlock algorithm.

Before you use the **WAIT** command, take note of the following items:

- The **WAIT** command only blocks the client that sends the **WAIT** command and does not affect other clients.

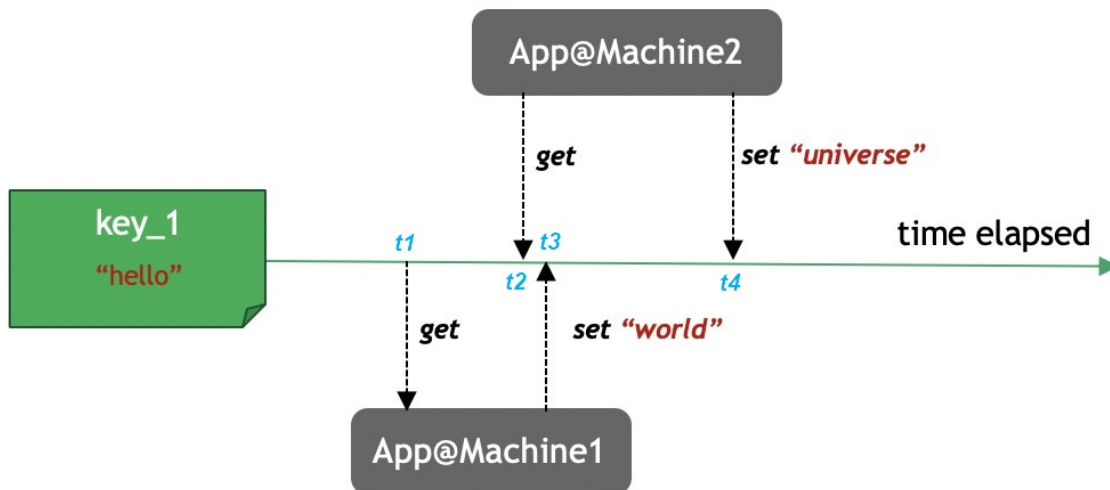
- If the **WAIT** command returns a valid value, the lock is synchronized from the master node to the replica nodes. However, if an HA switchover is triggered before the command returns a successful response, data may be lost. In this case, the output of the **WAIT** command only indicates a possible synchronization failure, and data integrity cannot be ensured. After the **WAIT** command returns an error, you can acquire a lock again or verify the data.
- You do not need to run the **WAIT** command to release a lock. This is because distributed locks are mutually exclusive. Logical failures do not occur even if you release the lock after a period of time.
- Use ApsaraDB for Redis Enhanced Edition (Tair)
 - The **CAS** and **CAD** commands help you reduce the costs of developing and managing distributed locks and improve lock performance.
 - Performance-enhanced instances of the ApsaraDB for Redis Enhanced Edition (Tair) provide three times the performance of open source Redis. Service continuity is ensured even if you use performance-enhanced instances to implement high-concurrency distributed locks. For more information about performance-enhanced instances, see [Performance-enhanced instances](#).
 - Persistent memory-optimized instances of the ApsaraDB for Redis Enhanced Edition (Tair) adopt Intel® Optane™ Persistent Memory to ensure real-time data persistence. A response is returned for each write operation after a successful data persistence attempt. Data loss is prevented even if power failures occur. For more information about persistent memory-optimized instances, see [Persistent memory-optimized instances](#). You can also specify the semi-synchronous mode for master-replica synchronization in persistent memory-optimized instances. In this mode, a successful response is returned to the client only if data is written to the master node and synchronized to the replica node. This prevents data loss after HA switchover. The semi-synchronous mode is degraded to the asynchronous mode if a replica node failure or network exception occurs during data synchronization.

4.3. Implement high-performance optimistic locking by using TairString

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 mechanisms is optimistic locking. Compared with open source Redis, performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair) provide the TairString data structure that allows you to implement optimistic locking to deliver higher performance at lower costs.

Concurrency and last-writer-wins

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



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 key_1 value `hello`.
3. At the t2 time point, application 2 reads the key_1 value `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 access and update operations on 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 performance-enhanced instances of ApsaraDB for Redis Enhanced Edition (Tair).

Implement optimistic locking by using 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. TairStrings consist of keys, values, and version numbers. For this reason, TairString is more suitable for optimistic locking. For more information about TairString commands, see [TairString](#).

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 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.
- When 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 Redis String except bit operations.

Due to these features, the locking mechanism is native to TairString data. Therefore, TairString provides an easy method for you to implement optimistic locking. Example:

```
while(true){
    {value, version} = EXGET(key);           // Retrieve the value and version number of the key
    .
    value2 = update(...);                   // Save the new value as value 2.
    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.contains("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 a TairString and create a TairString that has the same key as the deleted TairString, the key version of the new TairString is 1. The new TairString does not inherit the key version of the deleted TairString.
- You can specify the ABS option to skip version verification and forcibly overwrite the current version to update a TairString. 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 version number 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, you need only to send one access request in each while loop by using the **EXCAS** command of TairString. For more information about the **EXCAS** command, see [EXCAS](#). 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 succeeds, the TairString value is updated. If the verification fails, the following elements are returned:

- `update version is stale`
- `value`
- `version`

If the update fails, the command returns the current version number of the TairString. You do not need to run another query to retrieve the current version number, and only one access request is required for each while loop. Sample code:

```
while(true){
    {ret, value, version} = excas(key, new_value, old_version)    // Use the CAS command to
    replace the original 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.contains("update version is stale"))    // If the return value contains the
    "update version is stale" error message, the update fails. The values of the value and old_
    version variables are updated.
        update(value);
        old_version = version;
}
```

4.4. Implement bounded counters by using TairString

In flash sale scenarios where the sales period or product quantity is limited, you must handle traffic peaks that occur before, during, and after the sale period. You must also make sure that the number of purchase orders accepted does not exceed the number of products in stock. 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 way to implement bounded counters. You can use bounded counters to ensure that the accepted purchase orders do not exceed the upper limit. The solutions described in this topic are also applicable to other scenarios where rate limiting or throttling is required.

Bounded counters for flash sales

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

The **EXINCRBY** and **EXINCRBYFLOAT** commands for TairStrings have similar functions to the **INCRBY** and **INCRBYFLOAT** commands for 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 for native Redis strings. These options include *EX*, *NX*, *VER*, *MIN*, and *MAX*. For more information, see **TairString**. The solution described in this topic uses the *MIN* and *MAX* options. The following table describes the two 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, the required code is complex and difficult to manage. This may lead to excess purchase orders, where users are able to make successful purchases of items even after these items have already been sold out. TairString allows you to compile and run simple code to limit the exact number of purchase orders. Sample pseudocode:

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

Bounded counters for throttling

As with [Bounded counters for flash sales](#), you can specify the MAX option of the EXINCRBY command to implement bounded counters for throttling. Sample pseudocode:

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

Bounded counters for throttling can be used for various purposes such as limiting the number of concurrent requests, access frequency, and number of password changes. For example, in concurrency limiting scenarios, the number of concurrent requests suddenly exceeds the system performance threshold. To prevent service failures that cause severe consequences, you can use a bounded counter as a temporary solution to control the number of concurrent requests. This solution can respond to concurrent requests in a timely manner. If you want to limit the number of queries per second (QPS), you can compile and run simple code by using the EXINCRBY command for TairStrings to set a bounded counter for concurrent requests.

```
public boolean tryAcquire(Jedis jedis,String rateLimiter,int limiter){
    try {
        jedis.sendCommand(TairCommand.EXINCRBY,rateLimiter,"1","EX","1","MAX",String.valueOf(limiter), "KEEPTTL");
        // Set a bounded counter. EX 1 indicates that the rate limiter expires after 1 second. MAX limiter indicates that the upper limit is limiter. KEEPTTL indicates that the time-to-live (TTL) of an existing exstring is not modified.
        return true;
    }catch (Exception e){
        if(e.getMessage().contains("increment or decrement would overflow")){ // Check whether the returned result contains error messages.
            return false;
        }
        throw e;
    }
}
```

4.5. Implement multidimensional leaderboards by using TairZset

TairZset is a data structure developed by Alibaba Cloud. It allows you to sort score data of the DOUBLE type with respect to 256 dimensions.

Issues with Redis ZSET

The Sorted Set (or ZSET) data structure of open source Redis allows you to sort elements only in one dimension instead of multiple dimensions based on the DOUBLE-typed score data. For example, you can use the IEEE Standard for Floating-Point Arithmetic (IEEE 754) standard to concatenate score data to implement multidimensional sorting. However, this method is limited by complex logic, reduced precision, and the unavailability of the `ZINCRBY` command.

Introduction to TairZset

To help you implement multidimensional sorting, Alibaba Cloud developed the TairZset data structure. Compared with the preceding method, TairZset provides the following advantages:

- Allows DOUBLE-typed scores to be sorted based on a maximum of 256 dimensions. The scores are displayed from left to right based on their priorities.

In a multidimensional sorting, a left score has higher priority than a right score. Take the comparison of three-dimensional scores in the `score1#score2#score3` format as an example. TairZset compares the `score1s` of multiple three-dimensional scores and moves on to `score2s` only when `score1s` are equal. If `score1s` are not equal, the ranking of `score1s` represents the ranking of the three-dimensional scores involved. By the same logic, `score3s` are compared only if `score2s` are equal. If all `score1s` are equal and the same holds true for `score2s` and `score3s`, the involved multidimensional scores are ranked in ASCII sort order.

For easier understanding, you can imagine number signs (#) as decimal points (.). This way, `0#99 < 99#90 < 99#99` can be seen as `0.99 < 99.90 < 99.99`.

- Supports the `EXZINCRBY` command. You no longer need to perform the following operations: retrieve current data, apply the increments to the data, and then write the data back to Redis databases.
- Supports APIs similar to those available for native Redis ZSET.
- Allows you to implement and regular leaderboardsdistributed leaderboards
- Supports the open source TairJedis client. For more information about the TairJedis client, visit [alibabacloud-tairjedis-sdk](#). You can use the TairJedis client without the need to encode, decode, or encapsulate data. You can also encapsulate clients for other programming languages by referring to the open source code.




 **Note** For more information about the TairZset commands used in this topic, see [TairZset](#).




Scenarios

The following ranking requirements are common for various games, applications, and medals:

- Support for member query based on the specified score range, create, read, update, delete (CRUD) operations, and reverse sorting.
- Quick retrieval of sorting results.
- Scalability to implement Workloads can be offloaded to other data shards when the current data shard has insufficient storage or computing power. distributed leaderboards

Use TairZset to implement medal leaderboards

Rank	Participant	 Gold medal	 Silver medal	 Bronze medal
1	A	32	21	16

Rank	Participant	 Gold medal	 Silver medal	 Bronze medal
2	B	25	29	21
3	C	20	7	12
4	D	14	4	16
5	E	13	21	18
6	F	13	17	14

In the medal leaderboard, participants are sorted by the numbers of gold, silver, and bronze medals that they win. If the number of gold medals is the same, they are sorted by the number of silver medals. If the number of silver medals is also the same, they are sorted by the number of bronze medals. For example, Participants E and F have the same number of gold medals, but Participant E has more silver medals than Participant F. In this case, Participant E ranks higher than Participant F. You can use simple APIs to implement this multidimensional sorting with the help of the TairZset data structure.

You can run the following code to install the dependency. Alibaba Cloud SDK for TairJedis is used in this example. For more information, visit [alibabacloud-tairjedis-sdk](#).

```
<dependency>
  <groupId>com.aliyun.tair</groupId>
  <artifactId>alibabacloud-tairjedis-sdk</artifactId>
  <version>1.6.0</version>
</dependency>
```

The following sample code provides an example:

```

JedisPool jedisPool = new JedisPool();
// Create a leaderboard.
LeaderBoard lb = new LeaderBoard("leaderboard", jedisPool, 10, true, false);
// Rank the participants by the number of their gold medals. If the number of gold medals is
// the same, rank the participants by the number of their silver medals. If the number of silver
// medals is also the same, rank the participants by the number of their bronze medals.
//
//      Gold medal Silver medal Bronze medal
lb.addMember("A",      32,  21, 16);
lb.addMember("D",      14,   4, 16);
lb.addMember("C",      20,   7, 12);
lb.addMember("B",      25,  29, 21);
lb.addMember("E",      13,  21, 18);
lb.addMember("F",      13,  17, 14);
// Retrieve the rank of Participant A.
lb.rankFor("A"); // 1
// Retrieve the top 3 participants.
lb.top(3);
// [{"member":"A","score":"32#21#16","rank":1},
// {"member":"B","score":"25#29#21","rank":2},
// {"member":"C","score":"20#7#12","rank":3}]
// Retrieve the entire leaderboard.
lb.allLeaders();
// [{"member":"A","score":"32#21#16","rank":1},
// {"member":"B","score":"25#29#21","rank":2},
// {"member":"C","score":"20#7#12","rank":3},
// {"member":"D","score":"14#4#16","rank":4},
// {"member":"E","score":"13#21#18","rank":5},
// {"member":"F","score":"13#17#14","rank":6}]

```

Use TairZset to implement leaderboards by hour, day, week, or month or in real time

If you want to implement a monthly leaderboard for a key, the month information must be used as the index.

Leaderboards of various time ranges can be implemented by using multi-level indexing provided by the TairZset data structure. In this example, all data for the month of July is stored in a key named `julyZset`. The following code shows how to write the sample data to the key:


```

EXZINCRBY julyZset 7#2#6#16#22#100 7#2#6#16#22_user1
EXZINCRBY julyZset 7#2#6#16#22#50 7#2#6#16#22_user2
EXZINCRBY julyZset 7#2#6#16#23#70 7#2#6#16#23_user1
EXZINCRBY julyZset 7#2#6#16#23#80 7#2#6#16#23_user1

```

Note

- `7#2#6#16#22#100` indicates that the score was updated to 100 at 16:22 on 6 July. The date belongs to the second week of July.
- `7#2#6#16#22_user1` indicates the user whose score was updated at this point in time. A prefix indicates that time is added to the username.

Leaderboard type	Command and output
<p>Real-time hourly leaderboard. This type of leaderboards includes the members whose scores were updated within an hour before the current time. For example, the current time is 16:23, the leaderboard includes members whose scores were updated within the range of 15:23 to 16:23.</p> <div>  Note If the ranking results are frequently accessed, we recommend that you cache the ranking results. </div>	<p>Query command:</p> <pre>EXZREVRANGEBYSCORE julyZset 7#2#6#16#23#0 7#2#6#15#23#0</pre> <p>Command output:</p> <pre>1) "7#2#6#16#22_user1" 2) "7#2#6#16#22_user2"</pre>
<p>Leaderboard for a specific hour. For example, you can query the leaderboard that includes the members whose scores are updated within the time range of 16:00 to 17:00.</p>	<p>Query command:</p> <pre>EXZREVRANGEBYSCORE julyZset 7#2#6#17#0#0 7#2#6#16#0#0</pre> <p>Command output:</p> <pre>1) "7#2#6#16#22_user1" 2) "7#2#6#16#22_user2"</pre>
<p>Daily leaderboard. For example, you can query the leaderboard whose data was generated on July 5.</p>	<p>Before the query, use the following command to insert a data record that was generated on July 5:</p> <pre>EXZINCRBY julyZset 7#2#5#10#23#70 7#2#5#10#23_user1</pre> <p>Command output:</p> <pre>"7#2#5#10#23#70"</pre> <p>Query command:</p> <pre>EXZREVRANGEBYSCORE julyZset 7#2#6#0#0#0 7#2#5#0#0#0</pre> <p>Command output:</p> <pre>1) "7#2#5#10#23_user1"</pre>

Leaderboard type	Command and output
Weekly leaderboard. For example, you can query the leaderboard for the second week of July.	<p>Query command:</p> <pre>EXZREVRANGEBYSCORE julyZset 7#3#0#0#0#0 7#2#0#0#0#0</pre> <p>Command output:</p> <pre>1) "7#2#6#16#22_user1" 2) "7#2#6#16#22_user2" 3) "7#2#5#10#23_user1"</pre>
Monthly leaderboard. For example, you can query the leaderboard of July.	<p>Before the query, insert a data record that was generated on July 20.</p> <pre>EXZINCRBY julyZset 7#4#20#12#20#50 7#4#20#12#20_user1</pre> <p>Command output:</p> <pre>"7#4#20#12#20#50"</pre> <p>Query command:</p> <pre>EXZREVRANGEBYSCORE julyZset 7#6#0#0#0#0 7#0#0#0#0#0</pre> <p>Command output:</p> <pre>1) "7#4#20#12#20_user1" 2) "7#2#6#16#22_user1" 3) "7#2#6#16#22_user2" 4) "7#2#5#10#23_user1"</pre>

4.6. Implement fine-grained monitoring by using TairTS

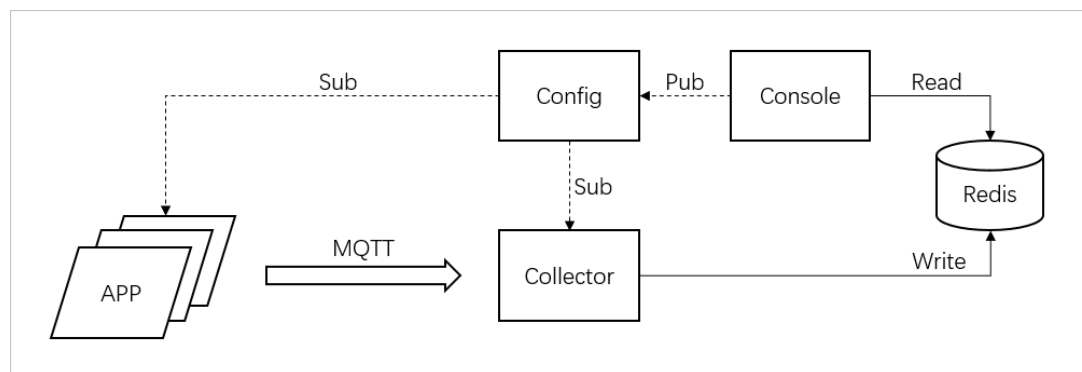
As the number of monitoring metrics and the amount of data traffic increase, monitoring systems become more complex and require higher time efficiency. This topic describes how to build a high concurrency fine-grained monitoring system by using TairTS.

Overview of TairTS

TairTS is a self-developed module of Tair that supports real-time and high concurrency queries and writes based on the multi-threading model of ApsaraDB for Redis Enhanced Edition (Tair). With TairTS, you can update or add to existing time series data, use the gorilla compression algorithm and specific storage to drastically reduce storage costs, and specify time to live (TTL) settings for keys to make them automatically roll based on time windows. For more information, see [TairTS](#).

Overview of fine-grained monitoring

Architecture of fine-grained monitoring



The preceding figure shows the architecture of a fine-grained monitoring system. The console sends fine-grained monitoring configurations to the application, the application writes the configurations to the collector by using the MQ Telemetry Transport (MQTT) protocol, and the collector processes the configuration data and then writes the data to ApsaraDB for Redis databases.

- High concurrency queries

During high concurrency queries, TairTS ensures query performance and supports aggregate operations in scenarios such as downsampling, attribute-based filtering, batch query, and the use of multiple numerical functions for multi-level filtering and query. With TairTS, you can perform batch query and aggregation by using a single command to reduce network interaction, receive responses in milliseconds, and identify issues at the earliest opportunity.

- High concurrency writes

One collector may be insufficient to handle high concurrency writes as applications become larger. In this regard, TairTS allows you to update or add to existing time series data to ensure the accuracy of concurrent writes to multiple collectors and reduce memory usage. The following code provides an example on how to concurrently write data:

```

import com.aliyun.tair.tairts.TairTs;
import com.aliyun.tair.tairts.params.ExtsAggregationParams;
import com.aliyun.tair.tairts.params.ExtsAttributesParams;
import com.aliyun.tair.tairts.results.ExtsSkeyResult;
import redis.clients.jedis.Jedis;

public class test {
    protected static final String HOST = "127.0.0.1";
    protected static final int PORT = 6379;
    public static void main(String[] args) {
        try {
            Jedis jedis = new Jedis(HOST, PORT, 2000 * 100);
            if (!"PONG".equals(jedis.ping())) {
                System.exit(-1);
            }
            TairTs tairTs = new TairTs(jedis);

```

```

//Use the following code if you want to work with a cluster instance:
//TairTsCluster tairTsCluster = new TairTsCluster(jedisCluster);
String pkey = "cpu_load";
String skey1 = "app1";
long startTs = (System.currentTimeMillis() - 100000) / 1000 * 1000;
long endTs = System.currentTimeMillis() / 1000 * 1000;
String startTsStr = String.valueOf(startTs);
String endTsStr = String.valueOf(endTs);
tairTs.extsdel(pkey, skey1);
long num = 5;
//Concurrently update data in Collector A.
for (int i = 0; i < num; i++) {
    double val = i;
    long ts = startTs + i*1000;
    String tsStr = String.valueOf(ts);
    ExtsAttributesParams params = new ExtsAttributesParams();
    params.dataEt(1000000000);
    String addRet = tairTs.extsrawincr(pkey, skey1, tsStr, val, params);
}
ExtsAggregationParams paramsAgg = new ExtsAggregationParams();
paramsAgg.maxCountSize(10);
paramsAgg.aggAvg(1000);
System.out.println("Updated result of Collector A:");
ExtsSkeyResult rangeByteRet = tairTs.extsrange(pkey, skey1, startTsStr, endTsStr, paramsAgg);
for (int i = 0; i < num; i++) {
    System.out.println("    ts: " + rangeByteRet.getDataPoints().get(i).getTs() + ", value: " + rangeByteRet.getDataPoints().get(i).getDoubleValue());
}
//Concurrently update data in Collector B.
for (int i = 0; i < num; i++) {
    double val = i;
    long ts = startTs + i*1000;
    String tsStr = String.valueOf(ts);
    ExtsAttributesParams params = new ExtsAttributesParams();
    params.dataEt(1000000000);
    String addRet = tairTs.extsrawincr(pkey, skey1, tsStr, val, params);
}
System.out.println("Updated result of Collector B:");
rangeByteRet = tairTs.extsrange(pkey, skey1, startTsStr, endTsStr, paramsAgg);
;
for (int i = 0; i < num; i++) {
    System.out.println("    ts: " + rangeByteRet.getDataPoints().get(i).getTs() + ", value: " + rangeByteRet.getDataPoints().get(i).getDoubleValue());
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Execution results:

```
Updated result of Collector A:
  ts: 1597049266000, value: 0.0
  ts: 1597049267000, value: 1.0
  ts: 1597049268000, value: 2.0
  ts: 1597049269000, value: 3.0
  ts: 1597049270000, value: 4.0
Updated result of Collector B:
  ts: 1597049266000, value: 0.0
  ts: 1597049267000, value: 2.0
  ts: 1597049268000, value: 4.0
  ts: 1597049269000, value: 6.0
  ts: 1597049270000, value: 8.0
```

4.7. Implement distributed leaderboards by using TairZset

TairZset is a data structure developed by Alibaba Cloud. It allows you to sort score data of the DOUBLE type with respect to 256 dimensions. You can use the Tair-based clients developed in-house to implement distributed leaderboards where computing tasks can be distributed to multiple keys (also called sub-leaderboards). For example, if you specify 10 keys, data is distributed to the 10 keys for computing.

Context

The precise ranking and imprecise ranking (also called linear interpolation) methods can be used to implement distributed leaderboards.

Methods to implement distributed leaderboards

Method	Description
Precise ranking (recommended)	<p>In this method, you can distribute data to multiple keys for computing, and query the ranks of the same member in multiple keys to obtain a total rank of the member.</p> <p>For example, if you specify three keys and create a leaderboard that has 3,000 members, Tair distributes these members to the three keys (or sub-leaderboards). During a data query, the FindRank(x) command is used to retrieve three ranks of the x member from the three keys. Assume the retrieved ranks are 124, 183, and 156. In this case, the actual rank of the x member is 463, which is the sum of 124, 183, and 156.</p> <ul style="list-style-type: none">• Benefits: This method yields precise ranks.• Drawbacks: The time complexity of this method is $m \cdot O(\log(N))$.
Linear interpolation (unavailable for now)	<p>In this method, you can classify members into different ranges by member score, record the number of members and the highest rank in each range, and then use linear interpolation to estimate the ranks of members whose scores fall between the largest and the smallest values in each range.</p> <ul style="list-style-type: none">• Benefits: This method is fast in rank retrieval and has a time complexity of $O(m)$.• Drawbacks: This method retrieves estimated ranks that may differ from the actual ranks.

This topic describes how to use precise ranking to implement distributed leaderboards.

 **Note** For information about the TairZset commands that are used in this topic, see [TairZset](#).

Prerequisites

The Tair-based client developed by Alibaba Cloud is used. For more information, visit [alibabacloud-tairjedis-sdk](#).

Implement distributed leaderboards

The following table compares the methods to implement basic features for common leaderboards and distributed leaderboards.

Basic feature	Common leaderboard		Distributed leaderboard	
	Implementation method	Time complexity	Implementation method	Time complexity
Insertion of a member	Run the EXZADD command.	$O(\log(N))$	Use the <code>crc(key) & m</code> syntax to specify the key into which you want to insert a member, and then run the EXZADD command to insert the member into the key.	$O(\log(N))$
Update of a member score	Run the EXZINCRBY command.	$O(\log(N))$	Use the <code>crc(key) & m</code> syntax to specify the key whose member score you want to update, and then run the EXZINCRBY command to update the score of a member in the key.	$O(\log(N))$
Removal of a member	Run the EXZREM command.	$O(M \cdot \log(N))$	Use the <code>crc(key) & m</code> syntax to specify the key whose member you want to remove, and then run the EXZREM command to remove a member from the key.	$O(\log(N))$
Query of the number of members in a key	Run the EXZCARD command.	$O(1)$	Run the EXZCARD command several times to individually query the number of members in multiple keys and add the numbers to obtain a total number.	$O(m)$ <div>  Note In this column, m indicates the number of shards. </div>

Basic feature	Common leaderboard		Distributed leaderboard	
	Implementation method	Time complexity	Implementation method	Time complexity
Query of the total number of pages	Run the EXZCARD command to query the number of members in a key, and then divide the number by the number of entries that can be displayed on each page.	$O(1)$	Run the EXZCARD command several times to individually query the number of members in multiple keys and add the numbers to obtain the total number. Then, divide the total number by the number of entries that can be displayed on each page.	$O(m)$
Query of the total number of members whose scores are within a specific range	Run the EXZCOUNT command.	$O(\log(N))$	Run the EXZCOUNT command several times to individually query the number of members whose scores are within a specific range in multiple keys, and then add the numbers to obtain the total number.	$m \cdot O(\log(N))$
Removal of the members whose scores are within a specific range	Run the EXZREMRANGEBYSCORE command.	$O(\log(N) + M)$	Run the EXZREMRANGEBYSCORE command several times to individually remove the members whose scores are within a specific range from multiple keys.	$m \cdot O(\log(N))$
Retrieval of a member score	Run the EXZSCORE command.	$O(1)$	Use the <code>crc(key) & m</code> syntax to specify the key whose member score you want to retrieve, and then run the EXZSCORE command to retrieve the score of a member in the key.	$O(1)$
Retrieval of a member rank	Run the EXZRANK command.	$O(\log(N))$	Run the EXZRANKBYSCORE command to individually retrieve the rank of the same member in multiple keys, and then add the ranks to obtain the total rank of the member.	$m \cdot O(\log(N))$

Basic feature	Common leaderboard		Distributed leaderboard	
	Implementation method	Time complexity	Implementation method	Time complexity
Retrieval of a member score and rank	Run the EXZSCORE and EXZRANK commands.	$O(\log(N))$	<ol style="list-style-type: none"> 1. Use the <code>crc(key) & m</code> syntax to specify the key whose member score and rank you want to retrieve, and then run the EXZSCORE command to retrieve the score and rank of a member in the key. 2. Run the EXZRANKBYScore command to individually retrieve the rank of the same member in multiple keys, and then add the ranks to obtain the total rank of the member. 	$m * O(\log(N))$
Query of the top i members	Run the EXZRANGE command.	$O(\log(N) + M)$	Run the EXZRANGE command several times to individually retrieve the top i members from multiple keys, and then obtain the top i members among all retrieved members.	$m * O(\log(N))$
Query of the top i pages of a leaderboard	Run the EXZRANGE command.	$O(\log(N))$	Retrieve the members displayed before the ith page in each sub-leaderboard, rank the retrieved members of all sub-leaderboards, and then obtain the total top i pages of all retrieved members.	$m * O(\log(N))$
Configuration of an expiration time	Run the EXPIRE command.	$O(1)$	Specify an expiration time for each member.	$O(m)$
Deletion of a leaderboard	Run the DEL command.	$O(N)$	Delete all members from a key.	$m * O(N)$

The following sample code provides an example:

```

public class DistributedLeaderBoardExample {
    private static final int shardKeySize = 10; // Number of sub-leaderboards.
    private static final int pageSize = 10;     // Number of entries that can be displayed
    on each page in a leaderboard.
    private static final boolean reverse = true; // In this example, members are ranked in
    descending order.
    private static final boolean useZeroIndexForRank = false; // In this example, ranks sta
    rt from 1.
    public static void main(String[] args) {
        JedisPool jedisPool = new JedisPool();
        // Create a distributed leaderboard.
        DistributedLeaderBoard dlb = new DistributedLeaderBoard("distributed_leaderboard",
        jedisPool,
            shardKeySize, pageSize, reverse, useZeroIndexForRank);
        // Rank the participants by the number of their gold medals. If the number of gold
        medals is the same, rank the participants by the number of their silver medals. If the numb
        er of silver medals is also the same, rank the participants by the number of their bronze m
        edals.
        //
        // Gold medal Silver medal Bronze medal
        dlb.addMember("A", 32, 21, 16);
        dlb.addMember("D", 14, 4, 16);
        dlb.addMember("C", 20, 7, 12);
        dlb.addMember("B", 25, 29, 21);
        dlb.addMember("E", 13, 21, 18);
        dlb.addMember("F", 13, 17, 14);
        // Retrieve the rank of Participant A.
        dlb.rankFor("A"); // 1
        System.out.println(dlb.rankFor("A"));
        // Retrieve the top 3 participants.
        dlb.top(3);
        System.out.println(dlb.top(3));
        // [{"member": "A", "score": "32#21#16", "rank": 1},
        // {"member": "B", "score": "25#29#21", "rank": 2},
        // {"member": "C", "score": "20#7#12", "rank": 3}]
    }
}

```

The following table describes the parameters.

Parameter	Type	Description
shardKeySize	int	The number of sub-leaderboards. The default value is 10. The number of sub-leaderboards cannot be dynamically scaled. Therefore, you must determine how many sub-leaderboards you need before you use sub-leaderboards.
pageSize	int	The number of entries that can be displayed on each page in a leaderboard. The default value is 10.
reverse	boolean	Valid values: <ul style="list-style-type: none"> false: Members are ranked in ascending order. This is the default value. true: Members are ranked in descending order.

Parameter	Type	Description
useZeroIndexForRank	boolean	Valid values: <ul style="list-style-type: none"> • true: Ranks start from 0. This is the default value. • false: Ranks start from 1.

4.8. Select users by using TairRoaring

You can provide a high-performance service for potential user selection by using the TairRoaring data structure available for ApsaraDB for Redis Enhanced Edition (Tair).

Introduction to TairRoaring

Tag-based user selection is applicable to business scenarios such as personalized recommendation and precision marketing. A variety of operational marketing strategies are implemented for users marked by different tags to maximize the interests of advertisers.

Tag-based user selection has the following characteristics:

- A large number of tags for users. This requires large storage space and high scalability.
- A large number of users. This indicates that a variety of dimensions are needed to generate tags and the data is discretized.
- A heavy computing burden. Applications can select users who are attached different tags based on a variety of strategies and have a high demand for performance and timeliness.

The bit map (or bitset) data structure is able to meet the preceding requirements. This data structure can use a small amount of storage to implement optimized query of large amounts of data. Bit map operations are supported by ApsaraDB for Redis Community Edition. However, the native bit map data structure may be overwhelmed by massive tagging needs.

- The native bit map data structure is limited by the size of keyspaces. This can lead to a significant reduction in space efficiency for sparse data.
- When bit map operations are performed by using strings, user code must be written to perform computing tasks and the round-trip time (RTT) increases threefold.
- When bit map data is stored in native Redis, big keys may be generated and cause instability to clusters.

TairRoaring commands are highly optimized bit map implementations. For more information, see [TairRoaring commands](#).

- TairRoarings can strike a balance between performance and space complexity in a large number of scenarios by means of two-level indexes and dynamic containers.
- TairRoarings use optimization techniques such as single instruction, multiple data (SIMD), vectorization, and popcount algorithms to improve computing efficiency and deliver efficient time and space complexity.
- TairRoarings provide powerful computing performance and high stability for a variety of business scenarios based on ApsaraDB for Redis Enhanced Edition (Tair).

Compared with the native bit map data structure, TairRoaring provides lower memory usage and higher computing efficiency for collections. TairRoaring also offers lower latency and higher throughput by virtue of the high-performance ApsaraDB for Redis Enhanced Edition (Tair) service.

Procedure of potential user selection

User selection consists of multiple steps, including model generation and selection.

1. Use row schemas to store user characteristics that are classified from different dimensions. In most cases, raw user data is stored in relational databases.
2. Process raw data on demand, and generate mappings between user identifiers (UIDs) and user tags.
3. Update these mappings on a regular basis to the TairRoaring data structure. In most cases, updates take place two days after the corresponding business data is generated.
4. Accelerate business data processing by using the TairRoaring data structure.

- You can query the relationship between a user and a user tag.

For example, you can run the following command to determine whether user1 is attached Tag-A. The serial number of Tag-A is 16161.

```
TR.GETBIT user1 16161
```

- You can create logical user groups by using operators such as `AND`, `OR`, and `DIFF` and process the information of these user groups.

For example, you can run the following command to obtain the users who are attached both Tag-B and Tag-C:

```
TR.BITOP result AND Tag-B Tag-C
```

- You can also use the TairRoaring data structure in some mapping scenarios such as risk control to check whether a tag is mapped to a UID.

For example, you can run the following command to query whether user1 is attached Tag-A:

```
TR.GETBIT Tag-A user1
```

5. Best Practices for All Editions


5.1. Migrate MySQL data to ApsaraDB for Redis

You can efficiently migrate data from ApsaraDB RDS for 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 by performing the steps described in this topic.

Scenario

In one of the classic use cases, ApsaraDB for Redis is used as a caching service between applications and databases to expand the capabilities of traditional relational databases. This also optimizes the ecosystem. ApsaraDB for Redis is used to store hot data. Applications can directly retrieve hot data from ApsaraDB for Redis. In addition, ApsaraDB for Redis can keep sessions alive for active users that use interactive applications. This reduces the load on the backend relational database and improves 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 you start, you must convert the source data to a specific structure. This topic describes how to use the open source tool to migrate tables from MySQL databases to ApsaraDB for Redis in an easy and efficient way. You can use the pipeline feature of ApsaraDB for Redis to transmit data in MySQL tables to hash tables of ApsaraDB for Redis.

 **Note** In this example, data is migrated from the source ApsaraDB RDS for MySQL instance to the destination ApsaraDB for Redis instance. A Linux environment that is deployed on an Elastic Compute Service (ECS) instance is used to run the command to migrate data. These instances are deployed in the same virtual private cloud (VPC), therefore they can communicate with each other.

You can follow the same procedure to migrate data from other relational databases to ApsaraDB for Redis. During the migration process, you must 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

- An ApsaraDB RDS for MySQL instance is created and stores the tables to be migrated.
- An ApsaraDB for Redis instance is created as the destination.
- An ECS instance that runs the Linux system is created.
- These instances are deployed in the same VPC and region.
- The private IP address of the ECS instance is added to the IP address whitelists of ApsaraDB RDS for MySQL and ApsaraDB for Redis instances.
- MySQL and Redis services are running on the ECS instance to extract, convert, and transmit data.

Note These prerequisites apply only when you migrate data on 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
d96b5		1986-05-23		.com	Michaelborough
662a7		1979-12-11		@example.net	Pamelaborough
db6c1		2001-09-06		example.net	Port Melodybury
38c2a		1979-06-08		ple.org	New Tracymouth
65613	Hernandez	1975-11-01	th.org	@example.net	North Matthewhaven
bb993	d Wagner	2004-06-29		le.net	East Angelamouth
132b1		2018-01-03	mith.com	ple.org	Bradleychester
8898c		1971-12-07		e.org	East Christianhaven
a5882		1976-07-14		e.org	Port Christopherberg

The table contains six columns. After the migration is complete, the values in the `id` column of the MySQL table are converted to hash keys in ApsaraDB for Redis. The names of other columns are converted to hash fields, and the values of these columns are converted to the values of the hash fields. You can modify the scripts and commands for the migration based on 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 structure of the MySQL table.
    '$', LENGTH('HMSET'), '\r\n', #The HMSET variable specifies the command that you run to write data to ApsaraDB for Redis.
    'HMSET', '\r\n',
    '$', LENGTH(id), '\r\n', #The id variable specifies the first field after you run the HMSET command for fields. This field is converted to the hash key in ApsaraDB for Redis
    id, '\r\n',
    '$', LENGTH('name'), '\r\n', #The name variable is passed to the hash table as a string field. Other fields such as sdate are processed in the same way.
    'name', '\r\n',
    '$', LENGTH(name), '\r\n', #The name variable specifies the company name in the MySQL table. This variable is converted 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 -
-skip-column-names --raw <mysql_to_redis.sql | redis-cli -h <Redis host> --pipe -a <Redis password>

```

Options

Name	Description	Example
------	-------------	---------

Name	Description	Example
-h	<p>The endpoint of the ApsaraDB RDS for MySQL database.</p> <p> Note This is the first <i>-h</i> in the command.</p>	<p>rm-bp1xxxxxxxxxxx.mysql.rds.aliyuncs.com</p> <p> Note Use the endpoint to connect the Linux server to the ApsaraDB RDS for MySQL database.</p>
-P	The service port of the ApsaraDB RDS for MySQL database.	3306
-u	The username of the ApsaraDB RDS for MySQL database.	testuser
-D	The database where the MySQL table that you want to migrate is stored.	mydatabase
-p	<p>The password of the ApsaraDB RDS for MySQL database.</p> <p> Note</p> <ul style="list-style-type: none">◦ If no password is set, you do not need to specify this parameter.◦ For higher security, you can enter only <i>-p</i>, run the command, and then enter the password as requested by the 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.

Name	Description	Example
-h	<p>The URL that is used to access the Redis database.</p> <p>Note This is the <code>-h</code> option that follows <code>redis-cli</code>.</p>	<p>r-bp1xxxxxxxxxxxxx.redis.rds.aliyuncs.com</p> <p>Note Use the endpoint to connect the Linux server to the ApsaraDB for Redis database.</p>
--pipe	Use the pipeline feature of ApsaraDB for Redis to transmit data.	No value is required.
-a	<p>The password that is used to access the Redis database.</p> <p>Note If no password is set, you can skip this parameter.</p>	Redispwd233

Sample code

```
[root@ ~]# mysql -h r-bp1xxxxxxxxxxxxx.mysql.rds.aliyuncs.com -P 3306 -u root -D custm_info -p --skip-column-names --raw < mysql_to_redis.sql | redis-cli -h r-bp1xxxxxxxxxxxxx.redis.rds.aliyuncs.com --pipe -a Redispwd233
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 the system returns, 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 data is migrated, one data entry in the MySQL table corresponds to one data entry in the hash table of ApsaraDB for Redis. You can run the `HGETALL` command to query a data entry and view the following result.

```
r-bp1xxxxxxxxxxxxx.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.

5.2. Rankings of online game players sorted by score

ApsaraDB for Redis is compatible with open source Redis. This topic provides an example on how to use ApsaraDB for Redis to create rankings of online game players sorted by score.

Environment settings

Cloud service	Description
Elastic Compute Service (ECS) instance	<ul style="list-style-type: none">The ECS instance runs the Ubuntu 16.04.6 operating system.The ECS instance and the ApsaraDB for Redis instance are deployed in the same virtual private cloud (VPC).
ApsaraDB for Redis instance	The ApsaraDB for Redis instance and the ECS instance are deployed in the same VPC.

 **Note** If the ApsaraDB for Redis instance and the ECS instance are deployed in different VPCs, you can migrate the ApsaraDB for Redis instance to the VPC of the ECS instance. For more information about how to change the VPC of an ApsaraDB for Redis instance, see [Change the VPC or vSwitch of an ApsaraDB for Redis instance](#). If the ApsaraDB for Redis instance and the ECS instance are deployed in different types of networks, see [Connect an ECS instance to an ApsaraDB for Redis instance in different types of networks](#).

Procedure


1. Configure the IP address whitelist of the ApsaraDB for Redis instance to make sure that the ECS instance and the ApsaraDB for Redis instance can communicate with each other.
 - i. Obtain the private IP address of the ECS instance. For more information, see [How do I query IP addresses of ECS instances?](#)
 - ii. Add the private IP address of the ECS instance to the whitelist of the ApsaraDB for Redis instance. For more information, see [Configure whitelists](#).
2. Log on to the ECS instance. For more information, see [Overview](#).
3. On the ECS instance, run the following commands to install the dependencies for the environment:

```
sudo apt-get update
sudo apt-get install openjdk-8-jdk
apt install maven
```

4. Run the following commands to download and decompress the sample code file:

```
wget https://docs.aliyun.cn-hangzhou.oss.aliyun-inc.com/assets/attach/120287/cn_zh/1615470698355/source.tar.gz
tar xvf source.tar.gz && cd source
```


5. Run the `vim src/main/java/test/GameRankSample.java` command to change the value of each parameter in the sample code based on your requirements.

 **Note** After you run the preceding command, the system opens the editor. Enter *a* to enter the editing mode.

Examples

```
public static void main(String[] args) {
    //Connection information. This information can be obtained from the console
    String host = "r-bp[REDACTED].redis.rds.aliyuncs.com";

    int port = 6379;
    Jedis jedis = new Jedis(host, port);
    try {
        //Instance password
        String authString = jedis.auth("F[REDACTED]"); //password
        if (!authString.equals("OK")) {
            System.err.println("AUTH Failed: " + authString);
            return;
        }
    }
}
```

Parameter	Description
String host	Enter the internal endpoint and port number of the ApsaraDB for Redis instance. For more information about how to obtain the internal endpoint and port number, see View endpoints .
port	
String authString	<p>The password of the account that has the read and write permissions. The password format varies based on the account that you select. For more information about how to create an account, see Create and manage database accounts.</p> <div><p> Note</p><ul style="list-style-type: none">◦ If you are using the default account, which is named after the instance ID, enter only the password.◦ If you are using a custom account, enter a password in the format of <code><user>:<password></code> . For example, if the username of a custom account is testaccount and the password is Rp829dlwa, you must enter testaccount:Rp829dlwa.</div>

- 6. To save the configuration file and exit the editor, press the Esc key to exit the edit mode, enter `:wq`, and press the Enter key.
- 7. To run the sample code, run the following commands.

```
mvn clean package assembly:single -DskipTests
java -classpath target/demo-0.0.1-SNAPSHOT.jar test.GameRankSample
```

Output

Comments on the sample code

```
package test;
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) {
        //The endpoint of the instance. You can view the endpoint in the ApsaraDB for Redis console.
        String host = "r-gs50a75e1968****.redis.hangzhou.rds.aliyuncs.com";
        int port = 6379;
        Jedis jedis = new Jedis(host, port);
        try {
            //The password of the instance.
            String authString = jedis.auth("Pass!123"); //password
            if (!authString.equals("OK")) {
                System.err.println("AUTH Failed: " + authString);
                return;
            }
            //The key.
            String key = "Game name: Keep Running, Alibaba Cloud!";
            //Clears all data.
            jedis.del(key);
            //Creates multiple player accounts.
            List<String> playerList = new ArrayList<String>();
            for (int i = 0; i < TOTAL_SIZE; ++i) {
                //Generates a random ID for each player.
                playerList.add(UUID.randomUUID().toString());
            }
            System.out.println("Inputs all players ");
            //Records the score of each player.
            for (int i = 0; i < playerList.size(); i++) {
                //Generates random numbers as the scores of players.
                int score = (int) (Math.random() * 5000);
                String member = playerList.get(i);
                System.out.println("Player ID:" + member + ", Player Score: " + score);
                //Adds the player IDs and scores to a specified sorted set.
                jedis.zadd(key, score, member);
            }
            //Prints the rankings 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 the specified sorted set.
        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()).intValue()
            );
        }
        //Prints information about 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()).intValue()
            );
        }
        //Prints 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 the list of players whose scores range from 1,000 to 2,000 from the specifi
ed sorted set.
        scoreList = jedis.zrangeByScoreWithScores(key, 1000, 2000);
        for (Tuple item : scoreList) {
            System.out.println(
                "Player ID:" +
                item.getElement() +
                ", Player Score:" +
                Double.valueOf(item.getScore()).intValue()
            );
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis.quit();
        jedis.close();
    }
}
}

```

5.3. 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 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(item.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 out put 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: Product: 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: Product: 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
```

5.4. 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

Messages published by 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, regardless of whether these messages were received. Thus, messages are lost after being published. Message subscribers can only receive messages after they have subscribed to the publisher. They will not receive the earlier messages in the channel.

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

Sample code

For the message sender (publisher 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);
        //The password of the ApsaraDB for Redis instance.
        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 ends > Channel: "+channel+" > Message:quit");
        //The message publisher stops sending by sending a quit message.
        jedis.publish(channel, "quit");
    }
}
```

For the message receiver (subscriber 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);
        //The password of the ApsaraDB for Redis instance.
        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 receiver is listening for subscribed messages, the process is blocked un
        til the quit message is received (in a passive manner) or the subscription is actively canc
        eled.
        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("-----SUBSCRIBE begins-----");
            subscribe();
            System.out.println("-----SUBSCRIBE 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 (in a passive manne
r).
        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 code block

```
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 fr
om the console.
    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 no clients have subscribed to
the channel, so the messages will not be received.
        pubClient.pub(channel, "Alibaba Cloud message 1: (No subscribers. This message
will not be received)");
        //The message receiver.
        KVStoreSubClient subClient = new KVStoreSubClient(host, port,password);
        JedisPubSub listener = new KVStoreMessageListener();
        subClient.setChannelAndListener(listener, channel);
        //The message receiver subscribes.
        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 receiver unsubscribes.
        subClient.unsubscribe(channel);
        Thread.sleep(1000);
        pubClient.pub(channel, "Alibaba Cloud message 2:(Subscription canceled. This me
ssage will not be received)");
        //The message publisher stops sending by sending a quit message.
        //When other message receivers receive quit in listener.onMessage(), the UNSUBS
CRIBE operation is performed.
        pubClient.close(channel);
    }
}
```

Returned result

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. This message will not be received)
-----SUBSCRIBE 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-77c7-4498-89a0-1dc5a2f65889
>>> PUBLISH > Channel: KVStore Channel-A> sends message: ed5924a9-016b-469b-8203-7db63d06f812
<<< SUBSCRIBE < Channel:KVStore Channel-A >receives message: ed5924a9-016b-469b-8203-7db63d06f812
>>> PUBLISH > Channel: KVStore Channel-A> sends message: f1f84e0f-8f35-4362-9567-25716b1531cd
<<< SUBSCRIBE < Channel:KVStore Channel-A >receives message: f1f84e0f-8f35-4362-9567-25716b1531cd
>>> PUBLISH > Channel: KVStore Channel-A> sends message: 746bde54-af8f-44d7-8a49-37d1a245d21b
<<< SUBSCRIBE< Channel:KVStore Channel-A >receives message: 746bde54-af8f-44d7-8a49-37d1a245d21b
>>> PUBLISH > Channel: KVStore Channel-A> sends message: 8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef
<<< SUBSCRIBE < Channel:KVStore Channel-A >receives message: 8ac3b2b8-9906-4f61-8cad-84fc1f15a3ef
>>> UNSUBSCRIBE > Channel: KVStore Channel-A
-----SUBSCRIBE 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 where only one publisher and one subscriber are involved. There can be multiple publishers, subscribers, and even multiple message channels. In such scenarios, you are required to change the code to fit the scenario.

5.5. Pipeline

ApsaraDB for Redis provides the pipeline feature similar to that of Redis.

Scenario

A client interacts with a server through one-way pipelines. One pipeline is used to send requests and the other is used to receive responses. You can send operation requests consecutively from the client to the server. However, during this period, the server does not send the response to each operation request. The client receives the response to each request from the server after 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 that uses 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);
        //The password of the ApsaraDB for Redis instance.
        String authString = jedis.auth(password); // password
        if (! authString.equals("OK")) {
            System.err.println("AUTH Failed: " + authString);
            jedis.close();
            return;
        }
        //Runs 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 a response.
            jedis.incr(key);
        }
        Date ts2 = new Date();
        System.out.println("Without Pipeline > value is: "+jedis.get(key)+" > Time
elapsed: " + (ts2.getTime() - ts1.getTime())+ "ms");
        //2 ----Using pipeline operations---
        jedis.del(key); //Initializes the key.
        Pipeline pl = jedis.pipelined();
        Date ts3 = new Date();
        for (int i = 0; i < COUNT; i++) {
            //Sends the request.
            pl.incr(key);
        }
        //Receives the response.
        pl.sync();
        Date ts4 = new Date();
        System.out.println("Using Pipeline > value is:"+jedis.get(key)+" > Time ela
psed:" + (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: 5,000 > Time elapsed: 5,844 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 = "xxxxxxx.m.cnha.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);
        //The password of the ApsaraDB for Redis instance.
        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); //Initializes the key.
        //----- 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 pipeline sends all requests, the client starts receiving responses.
        System.out.println("Sending requests completed. Start to receive responses");
    };

    List<Object> responses = p1.syncAndReturnAll();
    if (responses == null || responses.isEmpty()) {
        jedis.close();
        throw new RuntimeException("Pipeline error: no responses 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); //Initializes the key.
    Pipeline p2 = jedis.pipelined();
    //Declare the responses first.
    Response<Long> r1 = p2.incr(key);
    System.out.println("Pipeline sends requests");
```

```

        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 r
responses.

            }catch(Exception e){
                System.out.println(" <<< Pipeline error: the client has not started rec
eiving responses >>> ");
            }
        //After pipeline sends all requests, the client starts receiving responses.
        System.out.println("Sending requests completed. Start to receive responses"
);

        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 Java code, the following output is displayed:


```
----- Method 1 -----
Pipeline sends requests
Pipeline sends requests
Pipeline sends requests
Pipeline sends requests
Pipeline sends requests
After pipeline sends 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 requests
Pipeline sends requests
Pipeline sends requests
Pipeline sends requests
Pipeline sends requests
<Pipeline error: The client has not started receiving responses>
After pipeline sends 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
```

5.6. Process transactions

ApsaraDB for Redis supports the transaction mechanism defined in Redis.

Scenario

You can run **MULTI**, **EXEC**, **DISCARD**, **WATCH**, and **UNWATCH** commands to perform atomic operations in transactions.

 **Note** 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 transaction rollbacks.

Sample code 1: Two clients process 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);
    //The password of the ApsaraDB for Redis instance.
    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 the other client.
    new KVStoreTranscationTest().new OtherKVStoreClient().start();
    Thread.sleep(500);
    Transaction tx = jedis.multi();//Starts the transaction.
    //The following operations are 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 subsequent operations in a transaction. Other thread operations cannot be performed.
    tx.incr(client1_key);
    Thread.sleep(300);//The suspension of the thread does not affect the subsequent operations in a transaction. Other thread operations cannot be performed.
    tx.incr(client1_key);
    Thread.sleep(200);//The suspension of the thread does not affect the subsequent operations in a transaction. Other thread operations cannot be performed.
    tx.incr(client1_key);
    List<Object> result = tx.exec();//Performs the operations.
    //Parses and prints the results.
    for(Object rt : result){
        System.out.println("Client 1 > transaction in progress> "+rt.toString());
    }
    jedis.close();
}

class OtherKVStoreClient extends Thread{
    @Override
    public void run() {
        Jedis jedis = new Jedis(host, port);
        //The password of the ApsaraDB for Redis instance.
        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 transaction operations submitted by client 1 are sequentially implemented. Client 2 sends requests to perform an operation on another key during this period, but the operation is blocked. Client 2 must wait until all the transaction operations of client 1 are complete.

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

Sample code 2: Two clients process the same key

By modifying the preceding code, the two clients can process 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 following output is displayed: The two clients are in different threads but process the same key. However, while client 1 uses the transaction mechanism to process this key, client 2 is blocked and must wait until all the transaction operations of client 1 are completed.

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

5.7. Discover and resolve the hotkey issue

Keys that are frequently accessed in Redis are known as hotkeys. If hotkeys are improperly managed, Redis processes may be blocked and your service may be interrupted. This topic describes the solutions that use ApsaraDB for Redis to resolve the hotkey issue.

Overview

Causes

The hotkey issue can have the following two causes:

- The size of data consumed by users is much greater than that of produced data, as in the cases of hot sale items, hot news, hot comments, and celebrity live streaming.

The hotkey issue tends to occur unexpectedly, for example, the sales price promotion of popular commodities during Double 11. When one of these commodities is browsed or purchased tens of thousands of times, a large number of requests are processed, which causes the hotkey issue. Similarly, the hotkey issue tends to occur in scenarios where more read requests are processed than write requests. For example, hot news, hot comments, and celebrity live streaming.

- In these cases, hotkeys are accessed much more frequently than other keys. Therefore, most of the user 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 partitioning. During this process, the corresponding key is accessed on the server. When the load exceeds the performance threshold of the server, the hotkey issue occurs.

Impacts of the hotkey issue

- The traffic is aggregated and reaches the upper limit of the physical network adapter.
- Excessive requests queue up, and the partitioning service stops responding.
- The database is overloaded and the service is interrupted.

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 concentrated traffic. If hotkeys are densely distributed, a large number of hotkeys are cached. When the cache capacity is exhausted, the partitioning service stops responding. After the caching service stops responding, the newly generated requests are cached on the backend database. Due to its poor performance, this database is prone to exhaustion when the database handles a large number of requests. The exhaustion of the database leads to service interruption and a dramatic downgrading of the performance.

Common solutions

Rebuild the server or client to improve the performance.

Use a server cache

The client sends requests to the server. The server provides a multi-thread service, and a cache space is available based on the cache LRU policy. When the server is congested, it directly responds to the requests instead of forwarding them to the database. The server sends the requests from the client to the database and rewrite the data to the cache only after the congestion is cleared. By using this solution, the cache is accessed and rebuilt.

However, this solution has the following issues:

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

Use Memcache and Redis

In this solution, a separate cache is deployed on the client to resolve the hotkey issue. 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 speed, and no bandwidth limit. However, it has the following disadvantages:

- Wasted memory resources
- Dirty reading

Use a local cache

Using the local cache generates the following issues:

- hotkeys 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 issue be resolved?

ApsaraDB for Redis provides the solution to the hotkey issue

Read/write splitting solution

The nodes in the architecture serve the following purposes:

- Load balancing is implemented at the Server Load Balancer (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 replica nodes.

- 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, classify, and then distribute requests. For example, a proxy node sends all write requests to the master node and all read requests to the read replica nodes. But the read replica nodes in the module can be expanded to solve the hotkey reading issue. Read/write splitting supports flexible scaling for hotkey reading and can store a large number of hotkeys. It is client-friendly.

Hot data solution

In this solution, hotkeys are actively discovered and stored to resolve the hotkey issue. The client accesses an SLB instance and requests are distributed to a proxy node through the SLB instance. Then, the proxy node forwards the requests to the backend Redis instances.

A cache is added to the server. A local cache is added to each proxy node. This cache uses the LRU algorithm to cache hot data. A hotkey computing module is added to the backend data node to return the hot data.

The proxy architecture has the following benefits:

- The proxy nodes cache the hot data, and its reading capability can be scaled out.
- The database node computes the hot data set at a specified time.
- The database returns the hot data to the proxy nodes.
- The proxy architecture is transparent to the client, therefore, no compatibility is required.

Process hotkeys

Read hot 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 node. If K1 becomes a hotkey after the calculation conducted by the backend hotkey computing module, the proxy node caches the hotkey. In this way, the client can directly access K1 without using Redis. The proxy node can be scaled out. Therefore, the accessibility of the hot data can be enhanced.

Discover hot data

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

The database uses the following methods to compute the hot data:

- Hot data statistics based on statistical thresholds
- Hot data statistics based on statistical cycles
- Statistics collection method based on the version number without resetting the initial value
- Computing hotkeys on the database has a minor impact on the performance and occupies only 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 issue. The read/write splitting solution and the hot data solution can be extended. 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 hot data, while the proxy-based solution is more cost-effective.

5.8. 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

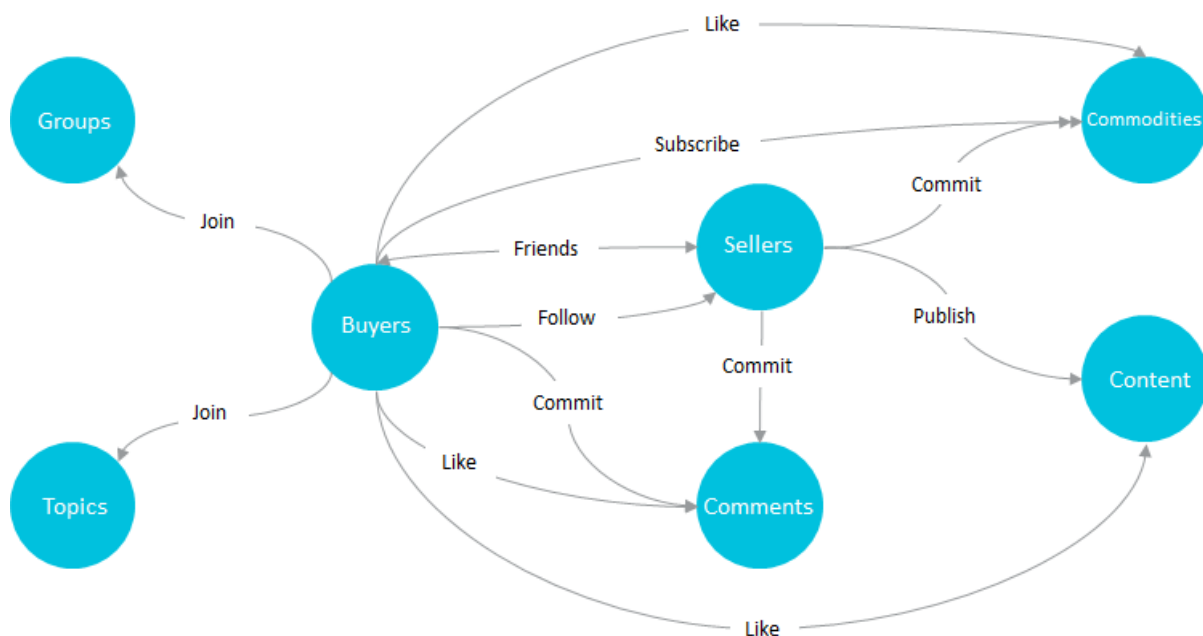
ApsaraDB for Redis provides multiple editions as follows: standard single-replica edition, standard dual-replica edition, and cluster edition.

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.

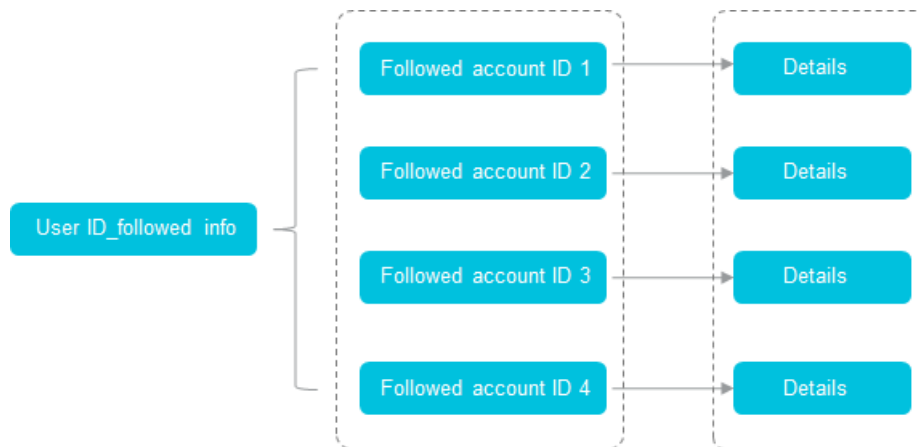
Store social relations for hundreds of millions of users in Weitao community

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

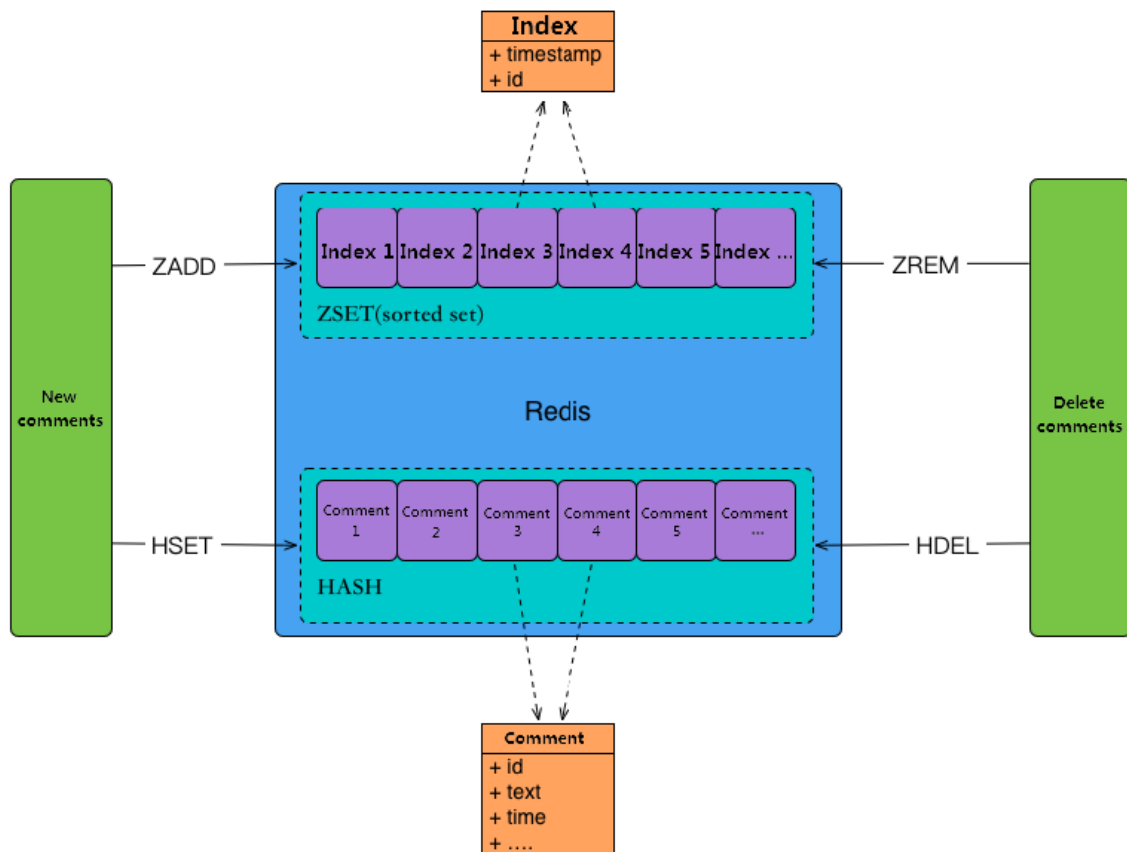


Paginate comments to live videos in Tmall based on a cursor

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

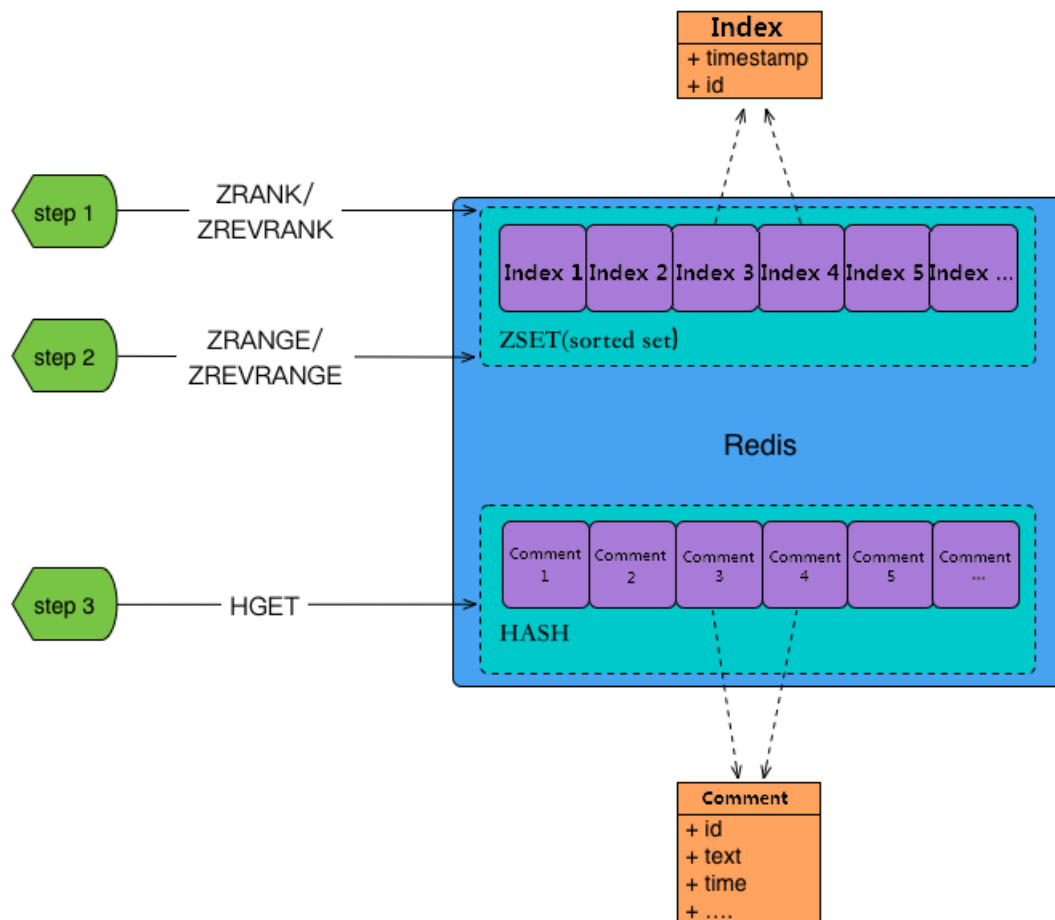
- Pull down for incremental comments: obtain a specified number of incremental comments from the specified position up.
- Pull-down refresh: obtain a specified number of the latest comments.
- Pull up for incremental comments: obtain a specified number of incremental comments from the specified position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

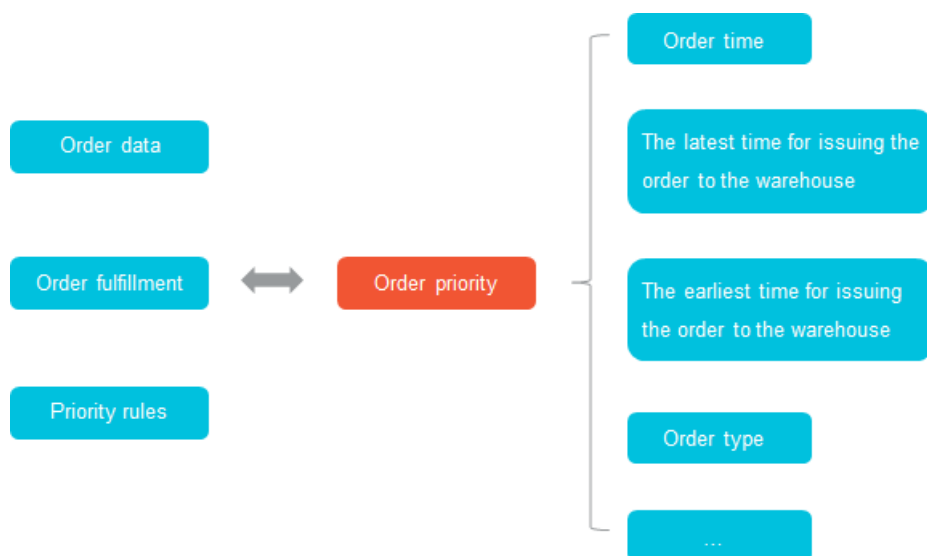
1. Obtain the current index ID.
2. Retrieve the index list.
3. Obtain the comments.



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.



5.9. Use ApsaraDB for Redis to build a business system that can handle flash sales

The flash sales strategy is commonly used for promotional events and brand marketing in the e-commerce industry. This strategy can help you increase the number of unique visitors and customer loyalty to 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 used to sell scarce or special commodities for specified quantities in a limited period of time. This attracts a large number of buyers. However, only a few buyers can place orders during the promotional event. A flash sales activity increases the number of unique visitors and order requests by dozens or hundreds of times that in regular sales activities on your platform within a short period of time.

A flash sales activity is divided into three phases:

- Before the promotional event: Buyers continuously refresh the commodity details page. As a result, the number of requests for this page spikes.
- During the promotional event: Buyers place orders. The number of order requests reaches a peak.
- After the promotional event: Specific buyers that have placed orders continue to query the status of orders or cancel orders. Most buyers continue to refresh 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 the requests that hold the lock to query inventory data and place orders. However, in these cases, the database cannot handle high concurrency. This may cause services to be blocked by a large number of requests and cause the server to stop responding 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. You can identify and block invalid requests in each phase in advance by using the hierarchy of the system architecture.

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

Before a flash sales activity, buyers continue to refresh the commodity details page. As a result, the number of requests for this page spikes. To resolve this issue, you must present details of commodities for flash sales and details of regular commodities on different web pages. Use static elements to present details of commodities for flash sales. Static data is cached in the browser and on CDN nodes, except for the place-order feature that requires interaction between the browser and server. This way, only a small fraction of the traffic that is caused by page refreshes before the promotion is redirected 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 retrieves data. The read/write splitting instance can handle more than 600,000 queries per second (QPS), which can meet 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 tag that indicates whether the flash sales activity begins:

```
"goodsId_count": 100 // The total number of commodities.  
"goodsId_start": 0 // The tag that indicates whether the flash sales activity begins.  
"goodsId_access": 0 // The number of order requests that are accepted.
```

1. Before the flash sales activity begins, the value of the goodsId_start parameter retrieved by the server cluster is 0. A value of 0 indicates that the flash sales activity has not begun.
2. After the data control module changes the value of the goodsId_start parameter to 1, the flash sales activity begins.
3. Then, the server cluster caches the goodsId_start tag and accepts order requests. The cluster updates the number of accepted order requests in goodsId_access. The number of remaining commodities is calculated in the following method: 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.

As a result, the business system accepts only a small fraction of the order requests. For high concurrency scenarios, a large amount of traffic is directed to the system. In this case, 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 speed up the removal of the item from the inventory

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

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

```
"goodsId" : {  
  "Total": 100  
  "Booked": 0  
}
```

Note The `goodsId` field indicates the commodity ID. The `Total` field indicates the number of the commodities in the inventory. The `Booked` field indicates the number of ordered commodities.

To remove the item from the inventory, the flash sales promotion server runs the following Lua script and connects to the ApsaraDB for Redis instance to obtain the order permission. The Lua script ensures the atomicity of multiple commands based on the Redis 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. Then, run the `EVALSHA` command to execute the script. This method requires less network bandwidth than directly running the `EVAL` command.

1. Cache the Lua script to the ApsaraDB for Redis instance.

```
SCRIPT LOAD "lua code"
```

The following result is returned:


```
"438dd755f3fe0d32771753eb57f075b18fed7716"
```

2. Run the Lua script.

```
EVALSHA 438dd755f3fe0d32771753eb57f075b18fed7716 1 goodsId 1
```

The following result is returned. The result indicates that an item is removed from the inventory.

```
(integer) 1
```

 **Note** In this case, if you run the `HGET goodsId Booked` command, the return value is `"1"`. The return value indicates that a commodity is ordered.

If the ApsaraDB for Redis instance returns the value `n` as the number of commodities that buyers ordered, the items are successfully removed from the inventory.

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

After the items are removed from the inventory, the flash sales business system writes order data to the database. The system can directly perform operations in the database for a few commodities. If the number of commodities for promotion is more than 10,000 or 100,000, lock conflicts may occur and can cause performance bottlenecks in the database. Therefore, to prevent directly writing data to the database, the flash sales system writes order data to message queues. Orders that are written to message queues are considered successfully placed orders.

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 speed up the order process.

Use the data control module to manage the synchronization of promotion data

At the start of the promotion, 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. After the promotion, the flash sales business system must process more traffic caused by order authentication failures and refund requests. 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.

5.10. 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.

Architecture

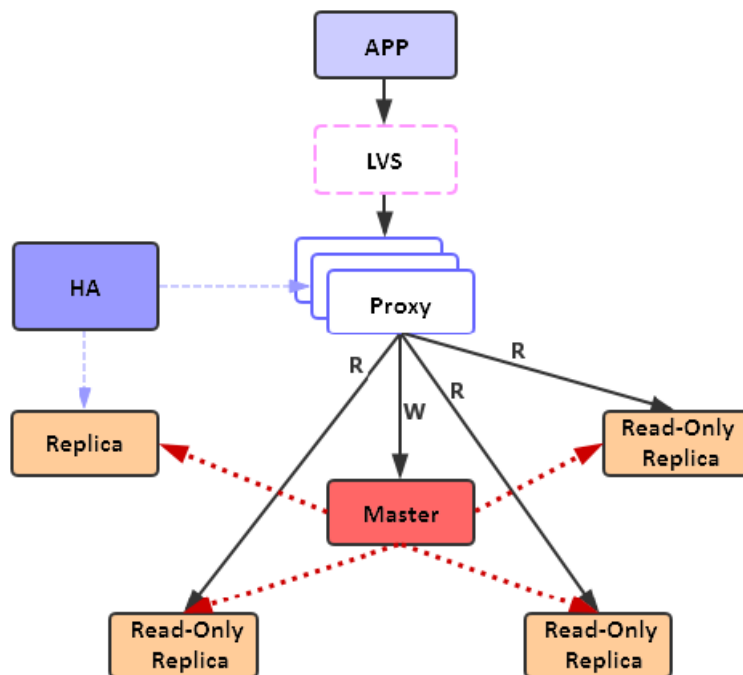
Redis cluster mode has several roles, including redis-proxy, Master, replica, and HA. In a read/write splitting instance, the read-only replica role is added to take over the read traffic. The replica serves as a hot standby and does not provide services. This architecture remains compatible with existing 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.

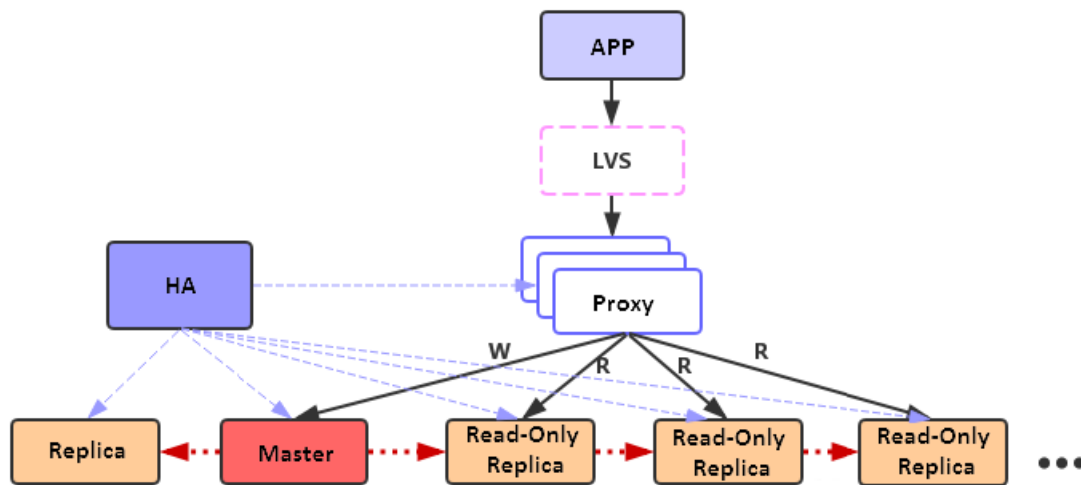


Cascading replication

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.

5.11. JedisPool optimization

You can set JedisPool parameters to proper values to improve Redis performance. This topic describes how to use JedisPool and configure the resource pool parameters. This topic also describes the recommended settings to optimize JedisPool.

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 pay attention to the GenericObjectPoolConfig parameter of the 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 i
nstance. redisPassword specifies the password of the instance. The timeout parameter specif
ies the connection timeout and the read/write timeout.
JedisPool jedisPool = new JedisPool(jedisPoolConfig, redisHost, redisPort, timeout, redisPa
ssword//);
//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. If you set the GenericObjectPoolConfig parameter to a proper value, you can improve the performance of Redis and reduce resource consumption. The following two tables describe important parameters and provide the recommended settings.

Parameters related to resource settings and resource usage

Parameter	Description	Default value	Recommended settings
maxTotal	The maximum number of connections that are supported by the pool.	8	For more information, see Recommended settings .
maxIdle	The maximum number of idle connections in the pool.	8	For more information, see Recommended settings .
minIdle	The minimum number of idle connections in the pool.	0	For more information, see Recommended settings .

Parameter	Description	Default value	Recommended settings
blockWhenExhausted	Specifies whether the client must wait when the resource pool is exhausted. Only when this parameter is set to true, the maxWaitMillis parameter takes effect.	true	We recommend that you use the default value.
maxWaitMillis	The maximum number of milliseconds that the client must wait when no connection is available.	A value of -1 specifies that the connection never times out.	We recommend that you do not use the default value.
testOnBorrow	Specifies whether to validate connections by using the PING command before the connections are borrowed from the pool. Invalid connections are 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 overhead of a ping test.
testOnReturn	Specifies whether to validate connections by using the PING command before the connections are returned to the pool. Invalid connections are 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 overhead of a ping test.
jmxEnabled	Specifies whether to enable Java Management Extensions (JMX) monitoring.	true	We recommend that you enable JMX monitoring. Take note that you must also enable the feature for your application.

Idle Jedis object detection provides the following four parameters.


Parameters related to idle resource detection

Parameter	Description	Default value	Recommended settings
testWhileIdle	Specifies whether to validate connections by running the PING command during the process of idle resource detection. Invalid connections are evicted.	false	true
timeBetweenEvictionRunsMillis	Specifies the cycle of idle resources detection. Unit: milliseconds.	A value of -1 specifies idle resource detection is disabled.	We recommend that you set this parameter to a proper value. You can also use the default configuration in JedisPoolConfig.

Parameter	Description	Default value	Recommended settings
minEvictableIdleTimeMillis	The minimum idle time of a resource in the resource pool. Unit: milliseconds. When the upper limit is reached, the idle resource is evicted.	1,800,000 (30 minutes)	The default value is suitable for most cases. You can also use the configuration in JedisPoolConfig based on your business requirements.
numTestsPerEvictionRun	The number of resources to be detected within each cycle.	3	You can change the value based on your application connections. A value of -1 specifies that the system checks all connections for idle resources.

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`.

Recommended settings

maxTotal: The maximum number of connections.

To set a proper value of maxTotal, take note of the following factors:

- The expected concurrent connections based on your business requirements.
- The amount of time that is consumed by the client to run the command.
- The limit of Redis resources. For example, if you multiply maxTotal by the number of nodes (ECS instances), the product 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 consumed to create and release connections. If the number of connections that are created and released for a request is large, the processes that are performed to create and release connections are adversely affected.

For example, the average time that is consumed to run a command, or the average time that is required to borrow or return resources and to run Jedis commands with network overhead, is approximately 1 ms. The queries per second (QPS) of a connection is about 1 second/1 millisecond = 1000. The expected QPS of an individual Redis instance is 50,000 (the total number of QPS divided by the number of Redis shards). The theoretically required size of a resource pool (maxTotal) is 50,000/1,000 = 50.

However, 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 the value of the maxTotal parameter is too large, the connections consume a large amount of client and server resources. For Redis servers that have a high QPS, if a large number of commands are blocked, the issue cannot be solved even by a large resource pool.

maxIdle and minIdle

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

The connection pool achieves 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 maxIdle and minIdle parameters to the same value if the user traffic fluctuates. If the number of concurrent connections is small or the value of the maxIdle parameter is too large, the connection resources are 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 Redis serves.

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 find proper values.

FAQ

Insufficient resources

You cannot obtain resources from the resource pool in the following cases:

- 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 blocks the connection 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 a limited pool size. For more information, see [Recommended settings](#). To fix this issue, we recommend that you check the network, the parameters of the resource pool, the resource pool monitoring (JMX monitoring), the code (for example, the reason is that `jedis.close()` is not executed), slow queries, and the domain name system (DNS).

Preload JedisPool

If you specify a small timeout value, the project may time out after it is started. JedisPool does not create a Jedis connection in the connection pool when JedisPool 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 is released to the pool after the connection is used. However, the process in which you create a connection and repeatedly release the connection may take a long period of 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 preload 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 {
    }
}
```

5.12. 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.

```
Welcome to Alibaba Cloud Elastic Compute Service !
[root@redisTest ~]# telnet r-b-4.redis.rds.aliyuncs.com 6379
Trying 1...
Connected to r-b-4.redis.rds.aliyuncs.com.
Escape character is '^'.
auth a
+OK
```

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

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

```
imonitor 0
+OK
+1543975816.789076 [0 ] "INFO" "replication"
+1543975833.071774 [0 ] "INFO" "replication"
+1543975842.251665 [0 127.0.0.1:42442] "INFO" "keyspace"
+1543975842.262597 [0 127.0.0.1:42442] "INFO" "all"
+1543975848.336031 [0 ] "INFO" "replication"
```

 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 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|!|\\)"|+)(? <!\s)"?( \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|!|\\)"|+)(? <!\s)"?( \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_co
unt) * 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 = Tr
ue)
    self._pretty_print(self._top_n(self.keys), 'Top Keys', percentages = True)
    self._pretty_print(self._top_n(self.commands), 'Top Commands', percentages = Tr
ue)
    self._pretty_print(self._time_stats(self.times), 'Command Time (microsecs)')
    self._pretty_print(self._heaviest_commands(self.times), 'Heaviest Commands (mic
rosecs)')
    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.

5.13. Use ApsaraDB for Redis to build a live-streaming channel information system

You can use ApsaraDB for Redis to build a live-streaming channel information system that has low latency and can withstand high traffic volumes.

Background information

Live-streaming channels are one of the main features of the live-streaming system. Except for the live-streaming window, live users, virtual gifts, comments, likes, rankings, and other data generated during the live streaming are time-limited, highly interactive, and delay-sensitive. The Redis caching service is a suitable solution to handle such data.

The best practice in this topic demonstrates how to use ApsaraDB for Redis to build a live-streaming channel information system. This topic describes how to build a live-streaming channel information system for three types of information:

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

Real-time ranking information

Real-time ranking information includes an online user list, a ranking of virtual gifts, and live comments. Live comments can be considered as a message ranking that is sorted based on message dimensions. The sorted set structure in Redis is suitable to store the real-time ranking information.

Redis sets are stored in hash tables. The time complexity of create, read, update, and delete (CRUD) operations is $O(1)$. Each member in a set is associated with a score to facilitate sorting and other operations. The following example shows how sorted sets work to build a live-streaming channel information system. The added and returned live comments are used in the example.

- Use unix timestamp + millisecond as the score format to record the last five live comments in the user55 live-streaming 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 message333333
(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
```

- Return the last three live comments:

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

- Return three live comments within the specified period of time:

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

Counting information

For 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 store this type of data. For example, the number of followers can be processed in the following way:

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

Timeline information

Timeline information is a list of information sorted in chronological order. Timeline information includes anchor moments and new posts. This information type is arranged in a fixed chronological order and can be stored by using a Redis list or an ordered list. Example:

```
redis> LPUSH user:55_recent_activity '{datetime:201804112010,type:publish,title: The show starts, content: Come on}'
(integer) 1
redis> LPUSH user:55_recent_activity '{datetime:201804131910,type:publish,title: Ask for a leave, content: Sorry, I have plans today.}'
(integer) 2
redis> LRANGE user:55_recent_activity 0 10
1) "{datetime:201804131910,type:publish,title:\xe8\xaf\xb7\xe5\x81\x87\",content:\xe6\x8a\x
b1\xe6\xad\x89\xef\xbc\x8c\xe4\xbb\x8a\xe5\xa4\xa9\xe6\x9c\x89\xe4\xba\x8b\xe9\xb8\xbd\xe4\x
b8\x80\xe5\xa4\xa9}"
2) "{datetime:201804112010,type:publish,title:\xe5\xbc\x80\xe6\x92\xad\xe5\x95\xa6,content:
\xe5\x8a\xa0\xe6\xb2\xb9}"
```

Related resources

- For more information about how to query hotkeys for a live-streaming system, see [Use the real-time key statistics feature](#).
- For more information about how to use offline key analysis to eliminate potential risks in workloads and identify performance bottlenecks, see [Offline key analysis](#).
- For more information about how to handle high concurrency, see [Cluster master-replica instances](#).

5.14. 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: `$0mrn`.
- **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']
```

5.15. Query hotkeys in Redis 4.0

High performance is the most prominent feature of Redis. Robust Redis performance is crucial to ensure the service availability. A reduced Redis performance can be caused by multiple reasons. The hot key issue 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.

 **Note** ApsaraDB for Redis now supports querying hotspot keys by using audit logs. This can help you query hotspot keys in the Redis service in an easy and accurate way. For more information, see [Query historical hotkeys](#).

Background information

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.

```
r-*****.redis.rds.aliyuncs.com:6379> OBJECT FREQ mylist  
(integer) 220
```

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

Note This topic describes how to discover hot keys 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](#) and [Quick Start](#).

Prerequisites

- You have activated an Elastic Compute Service (ECS) instance that can connect to an ApsaraDB for Redis instance.
- You have installed a Redis server whose version is later than 4.0 on the ECS instance.

Note You can use `redis-cli` after Redis is installed on the ECS instance.

- 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 [Modify parameters of an instance](#).

Procedure

- You can use the following command to query the hot keys when the ApsaraDB for Redis instance have running workloads.

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

Note This topic uses `redis-benchmark` to simulate a scenario that features a high volume of writes.

Option descriptions

Option	Description
-h	Specifies the endpoint of an ApsaraDB for Redis instance.
-a	Specifies the password of an ApsaraDB for Redis instance.
--hotkeys	Used to query hotkeys.

Results

The following example shows the result of running this command.

```
[root@yaozhou src]# redis-cli -h r-xxxxxx.redis.rds.aliyuncs.com --hotkeys
# Scanning the entire keyspace to find hot keys as well as
# average sizes per key type. You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[21.01%] Hot key 'key:__rand_int__' found so far with counter 167
[39.46%] Hot key 'mylist' found so far with counter 167
[67.29%] Hot key 'counter:__rand_int__' found so far with counter 51
[82.73%] Hot key 'myset:__rand_int__' found so far with counter 63

----- summary -----
Sampled 5008 keys in the keyspace!
hot key found with counter: 167 keyname: key:__rand_int__
hot key found with counter: 167 keyname: mylist
hot key found with counter: 63 keyname: myset:__rand_int__
hot key found with counter: 51 keyname: counter:__rand_int__
```

The summary part in the result displays the hotkeys.

5.16. Automatically add or remove ECS instances to or from a whitelist of an ApsaraDB for Redis instance

This topic describes how to use a lifecycle hook to put Elastic Compute Service (ECS) instances into the wait state and then use an Operation Orchestration Service (OOS) template to automatically add or remove the instances to or from a whitelist of an ApsaraDB for Redis instance.

Prerequisites

-
-
- An ApsaraDB for Redis instance is created.
-


Context

A scaling group can be associated with Server Load Balancer (SLB) or ApsaraDB for RDS instances, but cannot be associated with ApsaraDB for Redis instances. If your business data is stored in an ApsaraDB for Redis instance, you must manually add or remove ECS instances to or from a whitelist of the ApsaraDB for Redis instance. This is time-consuming and inefficient. You can use lifecycle hooks and OOS templates to automatically add or remove ECS instances to or from a whitelist of the ApsaraDB for Redis instance.

Procedure

In the following example, the ACS-ESS-LifeCycleModifyRedisIPWhitelist public template of OOS is used to demonstrate how to add ECS instances to a whitelist of an ApsaraDB for Redis instance during scale-out events. Perform the following steps to add ECS instances to a whitelist:

- [Step 1: Grant OOS permissions to the RAM role](#)
- [Step 2: Create a lifecycle hook for scale-out events and trigger a scale-out event](#)
- [Step 3: View the whitelist of the ApsaraDB for Redis instance](#)
- [Step 4: \(Optional\) View the execution status of the OOS template](#)

 **Note** If you want to remove ECS instances from a whitelist of an ApsaraDB for Redis instance during scale-in events, you can create lifecycle hooks that are applicable to scale-in events and then trigger the scale-in events.

Step 1: Grant OOS permissions to the RAM role

You must be granted the permissions to execute OOS templates. Resources of ECS, Auto Scaling, and ApsaraDB for Redis are involved when O&M operations specified in the ACS-ESS-LifeCycleModifyRedisIPWhitelist public template are performed.

- 1.
2. Create a policy.
 - i.
 - ii.
 - iii. On the **Create Custom Policy** page, configure parameters for the policy and click **OK**.

The following table describes the parameters used in this example. Use the default values for parameters that are not mentioned in the table.

Parameter	Description
Policy Name	Enter ESSHookPolicyForRedisWhitelist.
Configuration Mode	Select Script .

Parameter	Description
Policy Document	<div>Enter the following content:</div> <pre>{ "Version": "1", "Statement": [{ "Action": ["ecs:DescribeInstances"], "Resource": "*", "Effect": "Allow" }, { "Action": ["kvstore:ModifySecurityIps"], "Resource": "*", "Effect": "Allow" }, { "Action": ["ess:CompleteLifecycleAction"], "Resource": "*", "Effect": "Allow" }] }</pre>

3. Attach the policy to the OOSServiceRole RAM role.

- i.
- ii.
- iii. In the **Add Permissions** panel, configure the parameters and click **OK**.

The following table describes the parameters used in this example. Use the default values for parameters that are not mentioned in the table.

Parameter	Description
Authorized Scope	Select Alibaba Cloud Account .
Select Policy	Select Custom Policy and then the ESSHookPolicyForRedisWhitelist policy.


Step 2: Create a lifecycle hook for scale-out events and trigger a scale-out event

If you want ECS instances to be automatically added to a whitelist of an ApsaraDB for Redis instance when scale-out events are triggered, you can set the notification method to OOS Template and configure related parameters when you create lifecycle hooks.

- 1.
- 2.
- 3.
- 4.
5. Create a lifecycle hook for scale-out events.
 - i.
 - ii. Click **Create Lifecycle Hook**.


- iii. Configure parameters for the lifecycle hook and click **OK**.

The following table describes the parameters used in this example. Use the default values for parameters that are not mentioned in the table.

Parameter	Description
Name	Enter <code>ESSHookForAddRedisWhitelist</code> .
Applicable Scaling Activity Type	Select Scale-out Event .
Timeout Period	Enter an appropriate value, such as 300.  Note The timeout period is the period of time during which to perform customized operations. If the period is short, the operations may fail to be properly performed. Estimate the time required to perform the operations and set an appropriate timeout period.
Execution Policy	Select Continue .
Notification Method	Configure the following settings: <ul style="list-style-type: none">■ Notification method: Select OOS Template.■ OOS template type: Select Public Templates.■ Public template: Select <code>ACS-ESS-LifeCycleModifyRedisIPWhitelist</code> from the drop-down list. The parameters for the <code>ACS-ESS-LifeCycleModifyRedisIPWhitelist</code> public template: <ul style="list-style-type: none">■ dbInstanceId: Enter the ID of the ApsaraDB for Redis instance.■ modifyMode: Select Append. This value applies to scale-out events and allows ECS instances to be added to a whitelist of the ApsaraDB for Redis instance.■ Permissions: Select <code>OOSServiceRole</code>. In Step 1, the <code>OOSServiceRole</code> RAM role is granted permissions on resources of ECS, Auto Scaling, and ApsaraDB for Redis. OOS owns the preceding permissions after it assumes the RAM role.

6. Trigger a scale-out event.

In this example, a scale-out event is manually triggered by executing a scaling rule. You can also trigger scale-out events by using scheduled or event-triggered tasks.

 **Note** Lifecycle hooks take effect when scaling activities are manually triggered by executing scaling rules. Lifecycle hooks do not take effect when you manually add or remove ECS instances to or from a scaling group.

- i.

- ii. Click **Create Scaling Rule**.
- iii. In the Create Scaling Rule dialog box, configure the parameters and click **OK**.

The following table describes the parameters used in this example. Use the default values for parameters that are not mentioned in the table.

Parameter	Description
Rule Name	Enter Add1.
Rule Type	Select Simple Scaling Rule .
Operation	Set this parameter to Add 1 Instances.

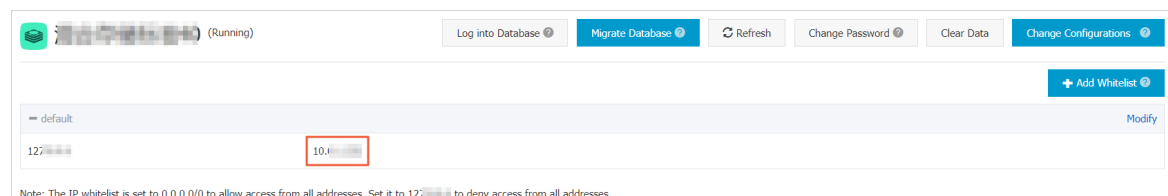
- iv. On the **Scaling Rules** page, find the created Add1 scaling rule and click **Execute** in the **Actions** column.
- v. Click **OK**.

After the scaling rule is executed, an ECS instance is automatically created. The ESSHookForAddRedisWhitelist lifecycle hook in the scaling group puts the ECS instance into the wait state. Auto Scaling automatically notifies OOS to perform the O&M operations specified in the ACS-ESS-LifeCycleModifyRedisIPWhitelist public template on the ECS instance.

Step 3: View the whitelist of the ApsaraDB for Redis instance

- Log on to the [ApsaraDB for Redis console](#).
- In the left-side navigation pane, click **Instances**.
- Find the ApsaraDB for Redis instance and click its ID in the **Instance ID/Name** column.
- In the left-side navigation pane, click **Whitelist Settings**.

The following figure shows that the private IP address of the ECS instance is added to the whitelist of the ApsaraDB for Redis instance as specified in the ACS-ESS-LifeCycleModifyRedisIPWhitelist public template.



If the ECS instance is created but its private IP address is not added to the whitelist of the ApsaraDB for Redis instance, log on to the OOS console to view the execution result of O&M tasks. For more information, see [Step 4: \(Optional\) View the execution status of the OOS template](#).

Step 4: (Optional) View the execution status of the OOS template

-
-
-
- On the page that appears, click the **Advanced View** tab.

The execution status is displayed on the **Execution Result** tab.

The screenshot displays the 'exec-5466173ae5674f2c885c1b' execution details. The 'Basic Information' tab is active, showing a successful execution status. The 'Visual Preview' tab is also visible, showing a flowchart with two tasks: 'getInstanceIpAddresses' and 'modifySecurityIps'. The 'modifySecurityIps' task is highlighted with a green checkmark, indicating success. The 'Execution Result' tab shows the output: 'ipAddresses: 10.10.10.10'.

Basic Information

Execution ID	Template Name	Execution Status	Start Time	End Time	Execution Mode
exec-5466173ae5674f2c885c1b	ACS-ESS-LifeCycleModifyRedis IPWhitelist (v9)	成功	2020年8月6日 16:09:56	2020年8月6日 16:09:57	自动执行

Input Parameters

Parameter	Value
OOSAssumeRole	OOSServiceRole
regionId	cn-hangzhou
lifecycleActionToken	02A31D28-4841-459C-80F8-05F0FFD45A69
instanceIds	['i-10000000000000000000']
dbInstanceId	r-bp10000000000000000000
lifecycleHook	展开

Execution Result

Execution Status	Result Output
成功	ipAddresses: 10.10.10.10

If the execution fails, the error message is also displayed on the **Execution Result** tab.

The screenshot displays the 'exec-5466173ae5674f2c885c1b' execution details. The 'Basic Information' tab is active, showing a failed execution status. The 'Visual Preview' tab is also visible, showing a flowchart with two tasks: 'getInstanceIpAddresses' and 'modifySecurityIps'. The 'modifySecurityIps' task is highlighted with a red exclamation mark, indicating failure. The 'Execution Result' tab shows the error message: 'Forbidden.RAM message: User not authorized to operate on the specified resource, or this API doesn't support RAM.'.

Basic Information

Execution ID	Template Name	Execution Status	Start Time	End Time	Execution Mode
exec-5466173ae5674f2c885c1b	ACS-ESS-LifeCycleModifyRedis IPWhitelist(v9)	Failed	Aug 6, 2020 4:17:15 PM	Aug 6, 2020 4:17:16 PM	Execution... Automatic

Input Parameters

Parameter	Value
OOSAssumeRole	OOSServiceRole
regionId	cn-hangzhou
lifecycleActionToken	15075FE3-142E-4D3F-AC34-D364192E2507
instanceIds	['i-10000000000000000000']
dbInstanceId	r-bp10000000000000000000
lifecycleHook	Expand

Execution Result

Execution Status	Status Information
Failed	requestId: 386BDF58-8B03-46EE-B23F-BE7C151F4746 code: Forbidden.RAM message: User not authorized to operate on the specified resource, or this API doesn't support RAM.