

Alibaba Cloud ApsaraDB for HBase Development Guide

Issue: 20200320

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 due to product version upgrades, adjustments, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and the updated versions of this document will be occasionally released through Alibaba Cloud-authorized channels. You shall 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 the document in the context that Alibaba Cloud products and services are provided on an "as is", "with all faults" and "as available" basis. 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 bear any liability for any errors or financial losses incurred by any organizations, companies, or individuals arising from their download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, bear responsibility for any indirect, consequent

ial, exemplary, incidental, 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 contact Alibaba Cloud directly if you discover any errors in this document

.

Document conventions

Style	Description	Example
	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.
	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.
	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.
	A note indicates supplemental instructions, best practices, tips, and other content.	 Note: You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click Settings > Network > Set network type.
Bold	Bold formatting is used for buttons, menus, page names, and other UI elements.	Click OK.
Courier font	Courier font is used for commands.	Run the <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 Instance_ID</code>
[] or [a b]	This format is used for an optional value, where only one item can be selected.	<code>ipconfig [-all -t]</code>

Style	Description	Example
<code>{}</code> or <code>{a b}</code>	This format is used for a required value, where only one item can be selected.	<code>switch {active stand}</code>

Contents

Legal disclaimer.....	I
Document conventions.....	I
1 HBase best practices.....	1
1.1 Data compression and encoding.....	1
1.2 Read optimization.....	2
1.3 Write optimization.....	5
1.4 Read/Write splitting.....	7
1.5 Pre-split.....	8
2 HBase table design.....	9
2.1 Rowkey design.....	9
2.2 Schema design.....	15

1 HBase best practices

1.1 Data compression and encoding

Currently, ApsaraDB for HBase platform supports the following compression algorithms: LZO, ZSTD, GZ, LZ4, SNAPPY, and NONE. NONE means that the compression is disabled.

The following table compares the compression rates and speeds of the compression algorithms in different scenarios.

Business type	Size of an uncompressed table	LZO (compression rate/decompression speed, Unit: MB/s)	ZSTD (compression rate/decompression speed, Unit: MB/s)	LZ4 (compression rate/decompression speed, Unit: MB/s)
Monitoring	419.75 TB	5.82/372	13.09/256	5.19/463.8
Logs	77.26 TB	4.11/333	6.0/287	4.16/496.1
Risk control	147.83 TB	4.29/297.7	5.93/270	4.19/441.38
Transaction records	108.04 TB	5.93/316.8	10.51/288.3	5.55/520.3

Recommendations:

- We recommend that you use the LZ4 compression algorithm for the scenarios with high response time (RT) requirements.
- We recommend that you use the ZSTD compression algorithm for scenarios with low RT requirements, such as monitoring and Internet of Things (IoT) scenarios.

Encoding

ApsaraDB for HBase supports DataBlockEncoding, which compresses data by reducing the duplicate parts in HBase KeyValue. We recommend that you use DIFF for DATA_BLOCK_ENCODING.

Procedure

Follow these steps to modify the compression encoding:

1. Modify the COMPRESSION property of the table.

```
alter 'test', {NAME => 'f', COMPRESSION => 'lz4', DATA_BLOCK_ENCODING => 'DIFF'}
```
2. The modifications do not take effect immediately. You must perform a major compaction for the modifications to take effect. Major

```
compactions are time consuming, and we recommend that you perform a
major compaction during off-peak hours.
major_compact 'test'
```

For more information, see [Exploration of ApsaraDB for HBase compression encoding](#).

1.2 Read optimization

ApsaraDB for HBase is highly flexible and adaptable to multiple scenarios. The following section describes optimizations for read operations. During production, you may encounter errors such as Full GC, Out of Memory (OOM), Region in Transition (RIT), and high read latency. While upgrading your hardware may solve some of these problems, a more feasible approach is to optimize HBase based on your needs.

We divide the optimizations into:

Client optimization, server optimization, and platform optimization (implemented in ApsaraDB for HBase).

Use batch GET requests instead of single GET requests.

This greatly decreases the number of RPC calls between the client and server, significantly increasing throughput.

```
Result[] re= table.get(List<Get> gets)
```

Check whether an appropriate cache size is set for SCAN operations

SCAN operations require the server to return a large amount of data per request . When the client calls a SCAN operation, the server returns data to the client in batches. This is designed to reduce workloads on both the server and client when a large amount of data is transmitted at a time. The data is cached locally. The default maximum number of cached records is 100. Some SCAN operations may return large amounts of data (hundreds or tens of thousands) over RPC requests. We recommend that you extend the size of the cache based on your needs.

```
scan.setCaching(int caching) // Set this parameter to 1000 if SCAN
operations may return large amounts of data.
```

Request a specified column family or column name

ApsaraDB for HBase is a column-oriented database. Data is stored based on column families, where each column family is a separate database. We recommend that you specify the column family or column name when you perform queries to reduce Input/Output (I/O).

Disable caching when you access ApsaraDB for HBase offline

When you access ApsaraDB for HBase offline, the entire data is read at one time. In this case, caching is not required. We recommend that you disable BlockCache for offline reads.

```
scan.setBlockCache(false)
```

Server optimization

Request balancing

Check the status of the read workload during peak hours. You can view the status in the ApsaraDB for HBase console. If there is obvious hotspotting, the ultimate solution is to redesign the rowkey to balance the workloads. An intermediate solution is to split the hotspotting regions.

Whether the BlockCache is set properly

BlockCache is important for read performance as a read cache. If a lot of data is requested, we recommend that you use a server with a core-to-memory ratio of 1:4. For example, a machine with 8 cores and 32 GB memory or a machine with 16 cores and 64 GB memory. You can increase the value of BlockCache and decrease the value of Memstore.

You can set `hfile.block.cache.size` to 0.5 and set `hbase.regionserver.global.memstore.size` to 0.3 in the console of ApsaraDB for HBase, and then click Restart.

Editable Parameter		Edit History					
Parameter	Default Value	Running Value	Unit	Restart Required	Value Range	Description of Parameter	Actions
hbase.dynamic.jars.enabled	false	false	STR	Yes	[true false]	Enable dynamic class loader	Edit
hbase.exporter.enabled	false	false	STR	Yes	[true false]		Edit
hbase.hregion.majorcompaction	604800000	604800000	INT	Yes	[0,604800000]	The time (in milliseconds) between major compactions of all HS...	Edit
hbase.ipc.server.callqueue.read.ratio	0.0	0.0	FLOAT	Yes	[0.0,0.7]	Split the call queues into read and write queues. The specified i...	Edit
hbase.ipc.server.callqueue.scan.ratio	0.0	0.0	FLOAT	Yes	[0.0,0.7]	Given the number of read call queues, calculated from the total ...	Edit
hbase.regionserver.global.memstore.lowerLimit	0.3	0.3	FLOAT	Yes	[0.24,0.475]	Maximum size of all memstores in a region server before flush...	Edit
hbase.regionserver.global.memstore.size	0.35	0.35	FLOAT	Yes	[0.3,0.5]	Maximum size of all memstores in a region server before new u...	Edit
hbase.regionserver.maxlogs	32	32	INT	Yes	[16,256]	If more than this many logs, force flush of oldest region to olde...	Edit
hbase.regionserver.thread.compaction.large	1	1	INT	No	[1,10]	Large compaction threads	Edit
hbase.regionserver.thread.compaction.small	1	1	INT	No	[1,10]	Small compaction threads	Edit

The number of HFiles

During read operations, HFiles need to be opened frequently. The number of I/O operations increases in proportion to the number of HFiles, increasing read latency. To reduce this effect, we recommend that you perform a major compaction during off-peak hours.

Whether compaction consumes a large amount of system resources

Compaction is mainly used to merge multiple smaller HFiles into one larger HFile. This operation improves the read performance of subsequent read operations, but also consumes large amounts of resources. Typically, a minor compaction does not consume a large amount of system resources unless the configuration is inappropriate. Do not perform a major compaction during peak hours. We recommend that you perform a major compaction during off-peak hours.

Whether the Bloomfilter is properly set

Bloomfilter is mainly used to filter HFiles during queries to avoid unnecessary I/O operations. Bloomfilter can improve the read performance. Generally, when you create a table, the default value of BLOOMFILTER is set to ROW.

Platform optimization

Whether the rate of data localization is too low (The platform has been optimized by Alibaba Cloud.)

If you have local HFiles, we recommend that you use Short-Circuit Local Read. During restart or expand operations, ApsaraDB for HBase automatically merges regions that have been moved. This ensures that the localization rate is not negatively affected. Furthermore, performing a regular major compaction helps improve the localization rate.

Short-Circuit Local Read (enabled by default)

Normally, Hadoop Distributed File System (HDFS) performs read operations through DataNode. Short-Circuit Local Read can be enabled to bypass DataNode and allow the client to directly read local data.

Hedged Read (enabled by default)

The system prioritizes Short-Circuit Local Read to read local data. However, in some special cases, the local read operation may fail for a short time due to disk problems or network problems. The Hedged Read operation has been developed to solve this problem. The basic working principle of Hedged Read is as follows: when a client initiates a local read request, and no result is returned after a period of time, the client will send the same request to DataNodes. If a result is returned, all subsequent results are discarded.

Disable swap (disabled by default)

When there is insufficient physical memory, part of the hard disk space is used as swap. This operation may cause high latency. Swap is disabled by default for ApsaraDB for HBase. However, disabling swap will cause high anon-rss, and the page reclaim operation cannot reclaim enough pages. This may cause the kernel to become unresponsive. To this effect, ApsaraDB for HBase has taken the relevant isolation measures to avoid this situation.

1.3 Write optimization

ApsaraDB for HBase writes data into HLog and Memory based in the Log-Structured Merge (LSM) mode. This means that ApsaraDB for HBase does not perform random input/output (I/O) operations, providing high performance and stability for

write operations. In most database solutions, write operations are optimized for reliability at the cost of performance.

Batch write

The batch write feature is provided to decrease the number of Remote Procedure Calls (RPC).

```
HTable.put(List<Put>)
```

Auto Flush

You can greatly improve write performance by setting Autoflush to false. However, we recommend that you wait until 2 MB of data is buffered (hbase.client.write.buffer) or the hbase.flushcommits() command is called before you call Autoflush on a RegionServer. The data is not written to the remote database.

HTable.setWriteBufferSize(writeBufferSize) can be used to set the buffer size.

Server optimization

WAL Flag

You can disable Write Ahead Log (WAL) to greatly improve write performance. This is because when WAL is disabled, no writes are performed to HLog, which reduces the number of I/O operations. Only in-memory writes to Memstore are performed.

This operation is applicable to scenarios where performance is prioritized over reliability.

Increase the memory size of Memstore

You can also increase the value of Memstore and reduce the value of BlockCache to improve write performance. This is the opposite of read optimization.

Check whether a large number of HFile files are generated

At high write speeds, a large number of HFiles are generated. This is because the merge speed of HFiles are slower than that of write operations.

To ensure efficient usage of resources, we recommend that you perform a major compaction during off-peak hours. If the number of HFiles cannot be reduced, we recommend that you add more nodes.

1.4 Read/Write splitting

ApsaraDB for HBase provides three basic read and write APIs: read (GET, SCAN), write (PUT). In actual use, there may be instances where you want to perform these operations at the same time without interfering with each other. However, read/write splitting is not enabled by default. To achieve this effect, perform the following steps:

- If your business handles high volumes of reads and writes and you want to prioritize reads over writes, we recommend that you implement read/write splitting.
- If your business handles a large volume of SCAN and GET requests, it is desired that SCAN requests do not affect the performance of GET requests.

Related configurations:

- `hbase.ipc.server.callqueue.read.ratio`
- `hbase.ipc.server.callqueue.scan.ratio`

Description:

- Setting the `hbase.ipc.server.callqueue.read.ratio` to 0.5 indicates that 50% of the threads are used for read requests.
- If you also set `hbase.ipc.server.callqueue.scan.ratio` to 0.5, 50% of the read threads are used for SCAN requests, which means that 25% of the total threads are used for SCAN requests.

Procedure

- Click the cluster in the ApsaraDB for HBase console and select **Parameter Configuration**.
- Modify the configuration based on the read/write status of the business.
- The modifications take effect only after the cluster is restarted. Restarting the cluster will not cause major business interruptions, but may cause network

jitters. We recommend that you restart the cluster during off-peak hours to prevent any potential negative effects.

Editable Parameter						
Edit History						
Parameter	Default Value	Running Value	Unit	Restart Required	Value Range	Description of Parameter
hbase.dynamic.jars.enabled	false	false	STR	Yes	[true false]	Enable dynamic class loader
hbase.exporter.enabled	false	false	STR	Yes	[true false]	
hbase.hregion.majorcompaction						The time (in milliseconds) between major compactions of all H
hbase.ipc.server.callqueue.read.ratio						Split the call queues into read and write queues. The specific
hbase.ipc.server.callqueue.scan.ratio						Given the number of read call queues, calculated from the tot
hbase.regionserver.global.memstore.lowerLimit						Maximum size of all memstores in a region server before flush
hbase.regionserver.global.memstore.size						Maximum size of all memstores in a region server before new
hbase.regionserver.maxlogs						If more than this many logs, force flush of oldest region to old
hbase.regionserver.thread.compaction.large						Large compaction threads
hbase.regionserver.thread.compaction.small	1	1	INT	No	[1,10]	Small compaction threads
hbase.rpc.timeout	60000	60000	INT	Yes	[30000,360000]	Server side RPC timeout

Configure the preceding parameters based on your business needs. The parameters are not specified by default. This indicates that read and write operations share threads.

1.5 Pre-split

First-time users of ApsaraDB for HBase may be unfamiliar with the service. These users may fail to specify the number of regions when creating tables, as well as use inappropriate rowkey design, which cause hotspotting.

The following statement is the most commonly used statement for creating a table:

```
create 't3',{NAME => 'f1',COMPRESSION => 'snappy' }, { NUMREGIONS => 50,
SPLITALGO => 'HexStringSplit' }
```

- The NUMREGIONS parameter specifies the number of regions. Typically, each region should be between 6 and 8 GB. If you have a large cluster, we recommend that you increase the number of regions based on the recommended region size.
- The SPLITALGO parameter indicates the split algorithm used to split rowkeys. ApsaraDB for HBase supports three pre-split algorithms: HexStringSplit, DecimalStringSplit, and UniformSplit.

Use scenarios for where each split algorithm is used:

- HexStringSplit: The rowkey is prefixed with a hexadecimal string.
- DecimalStringSplit: The rowkey is prefixed with a decimal string.
- UniformSplit: The prefix of the rowkey has no particular pattern.

For more information about the rowkey, see [Rowkey design](#).

- For more information about the compression algorithms, see [Data compression and encoding](#).

2 HBase table design

2.1 Rowkey design

Rowkey design is an important factor that affects the performance of ApsaraDB for HBase. The following text provides a collection of common problems and their solutions:

Rows are sorted in lexicographical order by rowkey. This design optimizes scanning and allows you to store related rows or adjacent rows that will be read together. However, poor rowkey design is a common cause of hotspotting.

Hotspotting occurs when a large amount of traffic is concentrated on one or a small number of nodes in a cluster. Traffic refers to operations such as reads and writes. If the traffic overloads the server that hosts a region, the performance of the region will degrade and may even cause the region to become unavailable. This may also have adverse effects on other regions in the same region server because the server cannot provide services for the requested load. Therefore, it is important to design data access patterns that can distribute load evenly across clusters.

To prevent hotspotting during write operations, the rowkey must be designed so that data can be written to as many regions as possible at the same time. Try to avoid writing data to only one region, unless it is necessary for the data to be in one region.

The following sections describe common methods to prevent hotspotting. Each method has its own pros and cons.

Salting

Salting in HBase refers to placing a random number at the beginning of a rowkey. This operation randomly assigns a prefix to each rowkey to cause it to sort differently than usual. The number of the possible prefixes corresponds to the number of regions to which you want to distribute data. If you notice rowkey patterns that appear repeatedly in other more evenly distributed rows, we recommend that you use salting. In the following example, salting distributes the load across multiple region servers. The example also illustrates the negative impact salting has on read operations.

The following table shows a few rowkeys, which are divided into regions based on their prefixes. For example, rowkeys that have the prefix 'a' are distributed to Region A, and rowkeys that have the prefix 'b' are distributed to Region B. The rowkeys of the following table all start with 'f'. Therefore, these rows are distributed to a single region.

```
foo0001  
foo0002  
foo0003  
foo0004
```

To distribute the rows evenly across different regions, you need four salts: a, b, c, and d. Each letter prefix corresponds to a different region, which distributes the rowkeys across four different regions. The following rowkeys are prefixed with a different letter each, and are written to four different regions at the same time. The throughput is four times that of writing all the data to one region.

```
a-foo0003  
b-foo0001  
c-foo0004  
d-foo0002
```

When you insert a new row, a random prefix from one of the four possible salt values is assigned to the row.

```
a-foo0003  
b-foo0001  
c-foo0003  
c-foo0004  
d-foo0002
```

When salting is performed, prefixes are assigned at random, which improves the throughput of write operations. However, the original order of the rows are affected, which increases the workload of read operations.

Hashing

Compared with salting, hashing is the use of a one-way hash to generate a consistent prefix instead of a randomly generated prefix. This allows you to specify the same prefix for specific rows in a way that distributes the load across region servers, but allows for predictability during read operations. You can use deterministic hash to refactor a rowkey on the client and retrieve the row by using the GET operation.

Take the example described in the salting section. You can use a one-way hash to obtain the rowkey `foo0003` and predict the prefix `a`. You can then combine the rowkey and prefix to obtain the row. This method can be further optimized. For example, always make the specific rowkey pairs in the same region.

Reversing the Key

Another common method used to prevent hotspotting is to reverse a fixed-length or numeric rowkey so that the most frequently changed part (the least significant digit) is in the front. This randomizes the rowkey at the cost of row order.

Monotonically increasing rowkeys or time series data

When data is written to an ApsaraDB for HBase cluster, the process is locked. During this time, all clients will wait for a region (a single node) to become unlocked. After the write operation completes, the cycle starts again. This problem frequently occurs when monotonically increasing or time series data are used as rowkeys. This also applies to sequential rowkeys, which orders non-sequential data in a sequential order, causing hotspotting. Therefore, try to avoid using timestamps or sequences (for example, 1, 2, 3) as rowkeys.

If you need to import files ordered by time (such as logs) to ApsaraDB for HBase, we recommend that you reference OpenTSDB documentation. The documentation includes a page that describes the pattern for ApsaraDB for HBase. The format of rowkey in OpenTSDB is `[metric_type] [event_timestamp]`. At first glance, this seems to contradict the idea of not using timestamps as rowkeys. However, OpenTSDB puts `metric_type` before `event_timestamp`. There are hundreds of `metric_type` values which are enough to distribute the load across the regions. Therefore, the PUT operations can still be distributed across various regions of the table, despite a continuous data input stream.

Minimize row and column sizes

In ApsaraDB for HBase, values are stored as cells in the system. To find a cell, you need to know the row, column name, and timestamp. Typically, if the size of the rows or column names is too large or even larger than the size of the value, some interesting scenarios may occur. There is an index in the storefiles of ApsaraDB for HBase that is used to facilitate random access to values. However, if the coordinates required to access a cell is too large, the index may consume a large amount of memory and is ultimately exhausted. To solve this problem, you can set a larger

region size, or use smaller rows and column names. You can also use compression to solve this problem to a greater degree.

In most cases, minor inefficiencies do not have great impacts on performance. However, in big data scenarios, it cannot be ignored. because column families, properties, and rowkeys may be repeated hundreds of millions of times in the data.

Column family

Make sure that the name of the column family is as small as possible. We recommend that you use only one character. (For example, use f)

Properties

Although the detailed property names (such as myVeryImportantAttribute) are easy to understand, we recommend that you use short property names (for example, via) in ApsaraDB for HBase.

Rowkey length

Make the rowkey short enough to be readable, which is helpful for obtaining data. (for example, Get vs. Scan). The short rowkey is useless for data access, but it is not better than a longer rowkey on improving the retrieve capabilities of get/scan. You need tradeoffs when you design rowkeys.

Byte pattern

The long type has 8 bytes. You can save unsigned integers up to 18,446,744,073,709, 551,615 within 8 bytes. If you store the preceding number as a string, assuming that each character takes up on byte, the number of bytes required to store the number is nearly three times the original.

You can use the following sample code to test this:

```
// long
//
long l = 1234567890L;
byte[] lb = Bytes.toBytes(l);
System.out.println("long bytes length: " + lb.length);    // Returns 8

String s = String.valueOf(l);
byte[] sb = Bytes.toBytes(s);
System.out.println("long as string length: " + sb.length);    //
Returns 10

// hash
//
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] digest = md.digest(Bytes.toBytes(s));
```

```

System.out.println("md5 digest bytes length: " + digest.length);
// Returns 16

String sDigest = new String(digest);
byte[] sbDigest = Bytes.toBytes(sDigest);
System.out.println("md5 digest as string length: " + sbDigest.length);
// Returns 26

```

However, binary representation makes the data hard to read outside the code. In the following example, a shell is displayed when you want to add a value:

```

hbase(main):001:0> incr 't', 'r', 'f:q', 1
COUNTER VALUE = 1

hbase(main):002:0> get 't', 'r'
COLUMN                                CELL
f:q                                    timestamp=1369163040570
, value=\x00\x00\x00\x00\x00\x00\x00\x01
1 row(s) in 0.0310 seconds

```

The shell tries to print a string, but in this case, it can only print a hexadecimal. The same thing happens when the rowkey is in the region. That will be fine if you know what is stored, but it may be hard to understand the result if any data can be put in the same cell. This is the most important consideration.

Reverse timestamp

A common problem in databases is finding the latest version of a row. In this case, a reversed timestamp can be used as part of the rowkey to facilitate sorting. This technology includes appending (Long.MAX_VALUE-timestamp) to the end of a key. For example, [key] [reverse_timestamp].

You can use [key] to scan the value of the latest [key] in the table and obtain the first record. The rowkeys of ApsaraDB for HBase are sorted in a sequential order, so the key is the first and before any other older rowkeys.

This technique can be used instead of requesting the version numbers in order to permanently save all versions (or for an extended period of time). In addition, you can quickly obtain other versions by using the same scan technique.

Rowkey and column family

The rowkey is within a column family. Therefore, the same rowkey can exist in each column family of the same table without any conflicts.

Rowkeys cannot be modified

Rowkeys cannot be modified. The only way to modify a rowkey is to delete it and then insert a new one. We recommend that you use a well-designed rowkey from the beginning (or before you insert a large amount of data).

Relationship between the rowkey and the region split

If the table has been pre-split, the next step is to understand how rowkeys are distributed across the region boundaries. Consider using displayable 16-bit characters as the key part of the rowkey to explain the importance. For example, 0000000000000000 to ffffffffffffffff. You can obtain 10 regions by using Bytes.split to specify the key ranges. This is a splitting method when you create a region with Admin.createTable(byte[] startKey, byte[] endKey, numRegions).

```
48 48 48 48 48 48 48 48 48 48 48 48 48 48 48
// 0
54 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10
// 6
61 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -68
// =
68 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -
124 -126 // D
75 75 75 75 75 75 75 75 75 75 75 75 75 75 72
// K
82 18 18 18 18 18 18 18 18 18 18 18 18 18 14
// R
88 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -44
// X
95 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -102
// -
102 102 102 102 102 102 102 102 102 102 102 102 102 102 102
// f
```

The problem is that the data will be stored in the first two regions and the last region, which causes heavy workloads on these regions due to uneven data distribution. You can refer to the ASCII Table to understand the reason. Based on the ASCII table, 0 is the 48th and f is the 102nd. Only the values [0-9] and [a-f] are meaningful, so the values of the range from 58th to 96th will not appear in the keyspace and the regions in the middle within this range will never be used. To pre-split the keyspace in this example, you need to customize the split.

Tutorial 1: Pre-splitting tables is a good practice. However, when you pre-split tables, make sure that all regions have the corresponding keyspaces. Although the preceding example is about the keyspace of the 16-bit key, the solution is also applicable to other keyspaces.

Tutorial 2: Although the 16-bit key is not preferred (usually used for the data that can be displayed), it can still be used with the pre-splitting tables if all regions have the corresponding keyspaces.

The following case shows how to pre-split a 16-bit rowkey.

```
public static boolean createTable(Admin admin, HTableDescriptor table
, byte[][] splits)
throws IOException {
    try {
        admin.createTable( table, splits );
        return true;
    } catch (TableExistsException e) {
        logger.info("table " + table.getNameAsString() + " already exists
");
        // the table already exists...
        return false;
    }
}

public static byte[][] getHexSplits(String startKey, String endKey,
int numRegions) {
    byte[][] splits = new byte[numRegions-1][];
    BigInteger lowestKey = new BigInteger(startKey, 16);
    BigInteger highestKey = new BigInteger(endKey, 16);
    BigInteger range = highestKey.subtract(lowestKey);
    BigInteger regionIncrement = range.divide(BigInteger.valueOf(
numRegions));
    lowestKey = lowestKey.add(regionIncrement);
    for(int i=0; i < numRegions-1;i++) {
        BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger
.valueOf(i)));
        byte[] b = String.format("%016x", key).getBytes();
        splits[i] = b;
    }
    return splits;
}
```

2.2 Schema design

You can create and update schemas of ApsaraDB for HBase by using HBase Shell or Admin in the Java API.

Before you modify column families, disable the following table:

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(config);
String table = "Test";

admin.disableTable(table);           // Disable the table

HColumnDescriptor f1 = ...;
admin.addColumn(table, f1);          // Add a column family
HColumnDescriptor f2 = ...;
admin.modifyColumn(table, f2);       // Modify a column family
HColumnDescriptor f3 = ...;
admin.modifyColumn(table, f3);       // Modify a column family
```

```
admin.enableTable(table);
```

Update schema

After you change a table or a column family (including the encoding algorithm, compaction pressure format, and block size), the changes will take effect the next time when a major compaction is performed or the StoreFile is rewritten.

Rules for table schema design

- The size of a region is between 10 GB and 50 GB.
- The size of a cell must be no larger than 10 MB. If the size of a cell exceeds 10 MB, you need to use Medium-sized Objects (MOBs). (currently not supported by ApsaraDB for HBase, which will be supported in version 2.0). If the size of a cell is even larger and the MOBs is not applicable, you can store it directly in Object Storage Service (OSS).
- Typically, a table contains one to three column families. Do not design an ApsaraDB for HBase table in the same way as an RDBMS table.
- A table can be divided into about 50 to 100 regions based on the rowkeys. We recommend that you define one or two column families for a table. Note: Each column family is continuous and different column families are separated.
- Make your column family name as short as possible because each value in the storage contains a column family name (ignoring prefix encoding).
- If you store data and logs on different devices based on time, define rowkeys consisting of device IDs and times. You can then create a table where no additional data is written to old regions except during specific time periods. In this case, you minimize the number of active regions, but maintain a large number of old regions that do not have new writes. Having a large number of regions is acceptable because only active regions consume resources.

Number of column families

Currently, ApsaraDB for HBase is not optimized for more than one column family. We recommend that you make the number of column families as small as possible.

The flushing and compaction operations are performed on one region. If a flush is triggered on a column family that has a large amount of data, the adjacent column families will also be flushed even though the amount of data they carry is small.

The compaction operation is now triggered based on the number of all files in a column family, rather than the file size.

When flushing and compaction involve multiple column families, many redundant I/O operations are performed. To solve this problem, you need to make flushing and compaction operations working on only one column family.

Try to operate on only one column family in the schema. Group columns with similar usage rates into one column family so that you can access only one column family each time to improve efficiency.

Column family cardinality

If there are multiple column families in a table, make sure that the cardinalities (such as the number of rows) among column families do not differ too much. For example, column family A contains one million rows and column family B contains one billion rows. The data of column family A may be distributed by rowkeys to many regions (and region servers). This will make scanning column family A very inefficient.

Number of versions

The number of row versions is configured per column family by the HColumnDescriptor parameter. The default value is 3. This parameter is very important, because ApsaraDB for HBase does not overwrite a value and it only appends data later. The early versions distinguished by the timestamp will be deleted when you run a major compaction. The usage of the HColumnDescriptor parameter is described in the data model section. The value of this version can be increased or decreased based on the specific application.

We recommend that you do not set the maximum number of versions to a high level (for example, hundreds or more) unless old data is very important to you. This causes the storage of files to become extremely large.

Minimum number of versions

Similar to the maximum number of row versions, the minimum number of versions is also configured per column family by HColumnDescriptor parameter. The default value is 0, which means that the feature is disabled. The minimum number of versions is used together with the Time To Live (TTL) parameter. You can configure the parameters such as: save valuable data for the last T seconds, up to

N versions, but at least M versions (M is the minimum version number, and $M < N$). This parameter is enabled for a column family only during the time to live and must be less than the number of row versions.

Supported data types

ApsaraDB for HBase supports the bytes-in/bytes-out interface through Put and Result, so anything that can be converted into byte arrays can be saved as values. The input can be strings, numbers, complex objects, or even images as long as they can be converted into bytes.

There is an actual length limit for a value. (For example, saving 10-50 MB objects to ApsaraDB for HBase may negatively affect the query performance.) All rows in ApsaraDB for HBase follow the HBase data model including versioning. When you design the schema, take these into account as well as the block size of the column families.

TTL

You can set TTL seconds for the column family. ApsaraDB for HBase will automatically delete data after it times out. The time zone of TTL in ApsaraDB for HBase is UTC .

The stored files that contain expired rows can be deleted by minor compaction. To disable this feature, you can set the `hbase.store.delete.expired.storefile` parameter to false, or set the minimum number of versions to a non-zero value.

The latest version of ApsaraDB for HBase will support storing the specified time in each cell. Cell TTLs are submitted as a property of the update request (such as Appends, Increments, and Puts) by using `Mutation#setTTL`. If the TTL property is set, it will be applied to all cells updated by this operation. There are two obvious differences between cell TTL handling and ColumnFamily TTLs:

- 1. The Cell TTLs are measured in the unit of milliseconds instead of seconds.**
- 2. The TTL of a cell cannot exceed the valid time set by ColumnFamily TTLs.**