AI DataLake

Development Guide

Issue 01

Date 2026-11-05



HUAWEI TECHNOLOGIES CO., LTD.

Copyright © Huawei Technologies Co., Ltd. 2026. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://www.huawei.com

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Iceberg Development Guide 1
1.1 Introduction to Iceberg 1
1.1.1 Iceberg Functions 1
1.1.2 Iceberg Table Properties 2
1.1.3 Iceberg Catalog 12
1.1.4 Schema Evolution of Iceberg Tables 13
1.1.5 Data Types Supported by Iceberg Tables 14
1.1.6 Iceberg Hidden Partitioning and Partition Evolution 17
1.1.7 Iceberg Liquid Clustering 19
1.2 Iceberg on Aura 22
1.2.1 Preparations Before Using Iceberg 22
1.2.2 Iceberg Configuration and Types 23
1.2.3 Iceberg on Aura DDL Syntax Description 24
1.2.4 Iceberg Table Query and Write 27
1.2.5 Expression Partitioning 28
1.2.6 Iceberg Table Services 37
1.3 Iceberg on Spark 38
1.3.1 Interconnecting Spark with Iceberg Catalog 38
1.3.2 Iceberg Configurations and Types 40
1.3.3 Iceberg on Spark DDL Syntax Description 63
1.3.4 Querying Data in the Iceberg Table 71
1.3.5 Writing Data to an Iceberg Table 76
1.3.6 Stored Procedures for Iceberg Tables 85
1.3.7 Using Spark to Read and Write Iceberg in Streaming Mode 113
1.3.8 Liquid Clustering Usage 115
1.4 Iceberg on Flink 116
1.4.1 Preparations Before Using Iceberg 116
1.4.2 Managing Iceberg Catalogs 116
1.4.3 Iceberg Configurations and Types 118
1.4.4 Iceberg on Flink DDL Syntax 122
1.4.5 Querying and Writing to Iceberg 123
1.5 Iceberg Development FAQs 124
1.5.1 How Do I Handle Concurrent Data Commit Conflicts? 124

 Copyright © Huawei Technologies Co., Ltd.

1.5.2 How Do I Handle Optimistic Lock Concurrent Commit Conflicts? 125
1.5.3 How Do I Use LakeFormation to Maintain Tables? 127
1.5.4 How Do I Use an Engine to Maintain Tables? 129
2 Lance Development Guide 134
2.1 Introduction to Lance 134
2.1.1 Lance Functions 134
2.1.2 Lance Table Directory Structures 136
2.1.3 Lance Indexes 138
2.1.4 Lance Table Schema Evolution 141
2.1.5 Data Types Supported by Lance Tables 141
2.1.6 Lance Namespace and LakeFormation Catalog 143
2.2 Lance on Aura 143
2.2.1 Preparations Before Using Lance 143
2.2.2 Lance Configurations and Types 144
2.2.3 Lance on Aura DDL Syntax Description 144
2.2.4 Querying and Writing Data to a Lance Table 153
2.2.5 Updating a Lance Table 153
2.2.6 Using Indexes for Lance Tables 154
2.2.7 Lance Late Read 163
2.3 Lance Development FAQs 166
2.3.1 How Do I Handle Lance Table Concurrency Conflicts? 166
3 Paimon Development Guide 170
3.1 Introduction to Paimon 170
3.1.1 Paimon Functions 170
3.1.2 Paimon Table Properties 172
3.1.3 Paimon Catalog 175
3.1.4 Paimon Table Schema Evolution 176
3.1.5 Data Types Supported by Paimon Tables 177
3.1.6 Paimon Table Partitions 178
3.2 Paimon on Spark 179
3.2.1 Interconnecting Spark with Paimon Catalog 179
3.2.2 Paimon Data Types 182
3.2.3 DDL Syntax 183
3.2.4 Writing Data to a Paimon Table 186
3.2.5 Querying Data in a Paimon Table 187
3.2.6 System Tables 188
3.2.7 Paimon Table Procedure Management 190
3.3 Paimon on Flink 196
3.3.1 Preparations for Using Paimon 196
3.3.2 Managing Paimon Catalogs 196
3.3.3 Paimon Configuration and Data Types 199
3.3.4 Paimon on Flink DDL Syntax 208

 Copyright © Huawei Technologies Co., Ltd.

3.3.5 Querying and Writing Data to a Paimon Table....210
3.4 Paimon Development FAQs....211
3.4.1 How Do I Handle Concurrent Data Commit Conflicts?....211
3.4.2 How Do I Handle Excessive Small Files?....212
3.4.3 How Do I Achieve Data Compatibility After Schema Evolution?....213
3.4.4 How Do I Maintain Tables?....214
4 Spark UDF Development Guide....216
4.1 Calling UDFs in Spark SQL Jobs....216
4.2 Calling UDAFs in Spark SQL Jobs....223
4.3 Calling UDTFs in Spark SQL Jobs....231
4.4 Calling Remote UDFs in Spark SQL Jobs....239
4.4.1 Overview....239
4.4.2 Creating a Maven Project and Configuring the Project Structure....244
4.4.3 Writing the UDF Code and Exporting the JAR File....252
4.4.4 Creating a Function with FunctionGraph....255
4.4.5 Using Remote UDFs....261
4.4.5.1 Creating a Function Using SQL....261
4.4.5.2 Calling a Function Using SQL....261
4.4.5.3 Deleting a Function Using SQL....262
4.4.5.4 Displaying All Functions....262
4.4.5.5 Displaying Function Details....263
4.5 Calling Remote UDAFs in Spark SQL Jobs....263
4.5.1 Overview....263
4.5.2 Creating a Maven Project and Configuring the Project Structure....266
4.5.3 Writing the UDAF Code and Exporting the JAR File....273
4.5.4 Creating a Function with FunctionGraph....279
4.5.5 Using Remote UDAFs....283
4.5.5.1 Creating a Function Using SQL....283
4.5.5.2 Calling a Function Using SQL....283
4.5.5.3 Deleting a Function Using SQL....284
4.5.5.4 Displaying All Functions....284
4.5.5.5 Displaying Function Details....285
4.6 Calling Remote UDTFs in Spark SQL Jobs....285
4.6.1 Overview....285
4.6.2 Creating a Maven Project and Configuring the Project Structure....287
4.6.3 Writing the UDTF Code and Exporting the JAR File....296
4.6.4 Creating a Function with FunctionGraph....299
4.6.5 Using Remote UDTFs....303
4.6.5.1 Creating a Function Using SQL....303
4.6.5.2 Calling a Function Using SQL....303
4.6.5.3 Deleting a Function Using SQL....304
4.6.5.4 Displaying All Functions....304

 Copyright © Huawei Technologies Co., Ltd.

4.6.5.5 Displaying Function Details..... 305

1 Iceberg Development Guide

1.1 Introduction to Iceberg

1.1.1 Iceberg Functions

Apache Iceberg is an open table format for huge analytic datasets. It adds tables to compute engines using a high-performance table format that works just like an SQL table.

Iceberg was built for huge tables. Iceberg is used in production where a single table can contain tens of petabytes of data and even these huge tables can be read without a distributed SQL engine.

Iceberg provides the following functions:

- Fast scan planning enables table reads and file searches without relying on a distributed SQL engine.
- Advanced filtering uses table metadata to prune data files based on partition and column-level statistics.
- Iceberg is designed to solve correctness problems in eventually-consistent cloud object stores. It works with any cloud store and reduces NameNode congestion when in HDFS, by avoiding listing and renames.
- Serializable isolation enables table changes to be atomic, and readers never see partial or uncommitted changes.
- Multiple concurrent writers use optimistic concurrency and will retry to ensure that compatible updates succeed, even when writes conflict.

Key Features

- **Scalability**: Easily scales to accommodate extensive data tables and large datasets.
- **Query performance**: Enhances data retrieval efficiency through its metadata management and query optimization capabilities.
- **Data version management**: Offers robust data version control, helping users track and revert data states.

 Copyright © Huawei Technologies Co., Ltd.

- **Ease of use**: Provides straightforward APIs and CLI tools, facilitating effortless creation, management, and query of data tables.
- **Flexible partitioning strategies**: Supports adaptable partitioning schemes tailored to diverse dataset characteristics.
- **Multi-version data support**: Enables effective version handling and historical tracing of data entries.

Concepts

- Table: Fundamental entity representing a structured collection of data, including metadata details, storage locations, and partitioning strategies.
- Partition: Division of data in a table into multiple subsets based on predefined criteria such as time, geographical locations, or product categories.
- Metadata: Descriptive elements outlining table structures, partitioning strategies, and data evolution histories, which are stored in persistent storage media like HDFS and S3.
- Snapshot: A static view of table data captured at a specific moment, encapsulating both data and associated metadata.
- Manifest: An inventory detailing constituent data files within a table, recording metadata like file paths, sizes, and partition specifics.

File Organizational Structure

Iceberg organizes data into two layers: the metadata management layer and the data storage layer.

- Metadata management layer:
  - Metadata files are in JSON format. They store the metadata information of the current version and all snapshots.
  - Manifest list files, also known as snapshot files, are in Avro format. A snapshot file is generated for each commit, with each line storing the path of a manifest file, the partition range of its stored data files, and information such as the number of data files added or deleted. This provides filtering information during queries to enhance speed.
  - Manifest files are in Avro format. They contain a list of information about multiple data files, where each row offers a detailed description of a data file, including its status, path, partition information, column-level statistics (like max/min values, count of nulls), file sizes, and the number of data rows within the files. Column-level statistics help in filtering out unnecessary files while scanning table data.
- Data storage layer: The files are in Parquet format by default.

1.1.2 Iceberg Table Properties

When creating or modifying an Iceberg table, you can use **TBLPROPERTIES** to set related properties.

 Copyright © Huawei Technologies Co., Ltd.

Read Properties

Table 1-1 Read properties

Property	Default Value	Description
read.split.target-size	134217728 (128 MB)	Target size for combining data input splits
read.split.metadata-target-size	33554432 (32 MB)	Target size for combining metadata input splits
read.split.planning-lookback	10	Number of bins to consider for combining input splits
read.split.open-file-cost	4194304 (4 MB)	Estimated cost to open a file, used as a minimum weight for combining splits
read.parquet.vectorization.enabled	false	Whether Parquet vectorized reads are used
read.parquet.vectorization.batch-size	5000	Batch size for parquet vectorized reads
read.orc.vectorization.enabled	false	Whether ORC vectorized reads are used
read.orc.vectorization.batch-size	5000	Batch size for ORC vectorized reads

Write Properties

Table 1-2 Write properties

Property	Default Value	Description
write.format.default	parquet	Default file format for the table. Possible options can be **parquet**, **avro**, or **orc**.
write.delete.format.default	parquet (The value is the same as that of **write.format.default** by default.)	Default delete file format for the table. Possible options can be **parquet**, **avro**, or **orc**.
write.parquet.row-group-size-bytes	134217728 (128 MB)	Parquet row group size
write.parquet.page-size-bytes	1048576 (1 MB)	Parquet page size

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
write.parquet.page-row-limit	20000	Parquet page row limit
write.parquet.dict-size-bytes	2097152 (2 MB)	Parquet dictionary page size
write.parquet.compression-codec	zstd	Parquet compression codec. Possible options can be **zstd**, **lz4**, **gzip**, **snappy**, or **uncompressed**.
write.parquet.compression-level	null	Parquet compression level
write.parquet.bloom-filter-enabled.column.col1	-	Hint to parquet to write a Bloom filter for the column **col1**
write.parquet.bloom-filter-max-bytes	1048576 (1 MB)	Maximum number of bytes for a Bloom filter bitset
write.parquet.bloom-filter-fpp.column.col1	0.01	False positive probability for a Bloom filter applied to **col1**. The value ranges from 0.0 to 1.0.
write.avro.compression-codec	gzip	Avro compression codec. Possible options can be **gzip** (deflate with 9 level), **zstd**, **snappy**, or **uncompressed**.
write.avro.compression-level	null	Avro compression level
write.orc.stripe-size-bytes	67108864 (64 MB)	Default ORC stripe size, in bytes
write.orc.block-size-bytes	268435456 (256 MB)	Default file system block size for ORC files
write.orc.compression-codec	zlib	ORC compression codec. Possible options can be **zstd**, **lz4**, **lzo**, **zlib**, **snappy**, or **none**.
write.orc.compression-strategy	speed	ORC compression strategy. Possible options can be **speed** and **compression**.

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
write.orc.bloom.filter.columns	-	Comma separated list of column names for which a Bloom filter must be created.
write.orc.bloom.filter.fpp	0.05	False positive probability for Bloom filter. The value ranges from 0.0 to 1.0.
write.location-provider.impl	null	Optional custom implementation for LocationProvider
write.metadata.compression-codec	none	Metadata compression codec. Possible options can be **none** or **gzip**.
write.metadata.metrics.max-inferred-column-defaults	100	Maximum number of columns for which metrics are collected. For tables with nested fields, the number of stored metrics may exceed the configured limit.
write.metadata.metrics.default	truncate(16)	Default metrics mode for all columns in the table. Possible options can be **none**, **counts**, **truncate**(*length*), or **full**.
write.metadata.metrics.column.col1	-	Metrics mode for the column **col1** to allow per-column tuning. Possible options can be **none**, **counts**, **truncate**(*length*), or **full**.
write.target-file-size-bytes	536870912 (512 MB)	Target size of generated files, in bytes
write.delete.target-file-size-bytes	67108864 (64 MB)	Target size of delete files generated, in bytes

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
write.distribution-mode	none (The default value varies depending on the engine.)	Distribution of write data. Possible options are as follows: • **none**: Do not shuffle rows. • **hash**: Hash distribute by partition key. • **range**: Range distribute by partition key or sort key if table has an SortOrder.
write.delete.distribution-mode	hash	Distribution of write delete data
write.update.distribution-mode	hash	Distribution of write update data
write.merge.distribution-mode	none	Distribution of write merge data
write.wap.enabled	false	Whether to enable write-audit-publish writes
write.summary.partition-limit	0	Number of partitions that are changed, below which partition-level summary stats are included in the snapshot summaries.
write.metadata.delete-after-commit.enabled	false	Whether to delete the oldest tracked version metadata files after each table commit
write.metadata.previous-versions-max	100	Maximum number of previous version metadata files to track. Excess files will be deleted.
write.object-storage.enabled	false	Whether to enable the object storage location provider that adds a hash component to file paths
write.object-storage.partitioned-paths	true	Whether to include the partition values in the file path during data writes to object storage

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
write.data.path	*Table location* + /data	Base location for data files
write.metadata.path	*Table location* + /metadata	Base location for metadata files
write.delete.mode	copy-on-write	Mode used by **delete** commands. Possible options can be **copy-on-write** or **merge-on-read** (supported by v2 and v3).
write.delete.isolation-level	serializable	Isolation level for delete commands. Possible options can be **serializable** or **snapshot**.
write.update.mode	copy-on-write	Mode used by **update** commands. Possible options can be **copy-on-write** or **merge-on-read** (supported by v2 and v3).
write.update.isolation-level	serializable	Isolation level for update commands. Possible options can be **serializable** or **snapshot**.
write.merge.mode	copy-on-write	Mode used by **merge** commands. Possible options can be **copy-on-write** or **merge-on-read** (supported by v2 and v3).
write.merge.isolation-level	serializable	Isolation level for merge commands. Possible options can be **serializable** or **snapshot**.

 Copyright © Huawei Technologies Co., Ltd.

Table Behavior Properties

Table 1-3 Table behavior properties

Property	Default Value	Description
commit.retry.num-retries	4	Number of times to retry a commit before failing
commit.retry.min-wait-ms	100	Minimum time in milliseconds to wait before retrying a commit
commit.retry.max-wait-ms	60000 (1 min)	Maximum time in milliseconds to wait before retrying a commit
commit.retry.total-timeout-ms	1800000 (30 min)	Total retry timeout period in milliseconds for a commit
commit.status-check.num-retries	3	Number of times to check whether a commit succeeded after a connection is lost before failing due to an unknown commit state
commit.status-check.min-wait-ms	1000 (1s)	Minimum time in milliseconds to wait before retrying a status-check
commit.status-check.max-wait-ms	60000 (1 min)	Maximum time in milliseconds to wait before retrying a status-check
commit.status-check.total-timeout-ms	1800000 (30 min)	Total timeout period in which the commit status-check must succeed, in milliseconds
commit.manifest.target-size-bytes	8388608 (8 MB)	Target size for merging manifest files
commit.manifest.min-count-to-merge	100	Minimum number of manifests to accumulate before merging
commit.manifest-merge.enabled	true	Whether to automatically merge manifests on writes

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
history.expire.max-snapshot-age-ms	432000000 (5 days)	Default maximum age of snapshots to keep on the table and all of its branches while expiring snapshots
history.expire.min-snapshots-to-keep	1	Default minimum number of snapshots to keep on the table and all of its branches while expiring snapshots
history.expire.max-ref-age-ms	Long.MAX_VALUE (forever)	Default maximum age of snapshot references to keep while expiring snapshots, except for the main branch, which never expires

Others

Property	Default Value	Description
format-version	2	Metadata format version, which determines the functions and write capabilities supported by the table • **v1**: Only write in append mode is supported. Row-level delete and edit are not supported. • **v2** (default): Row-level delete, update, and merge operations are supported. Incremental update, delete, and edit are supported. • **v3**: Currently, only the Spark engine supports **variant** data type, row-level lineage tracing, and binary deletion.

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
compatibility.snapshot-id-inheritance.enabled	false	Whether to allow a snapshot to be committed when the snapshot ID is not specified. If the value of **format-version** is greater than 1, this configuration takes effect forcibly.
write.spark.fanout.enabled	false	Whether to enable the fanout writer in Spark. Enabling the fanout writer does not require data clustering but requires more memory. This is a Spark-specific parameter.
write.spark.accept-any-schema	false	Whether to allow the data schema to be different from the table schema when Spark writes data. If this parameter is set to **true**, the system automatically supports field addition and sequence adjustment. This is a Spark-specific parameter.
spark.sql.native-read	false	Whether to enable Iceberg Native read. This is a Spark-specific parameter.
spark.sql.native-write	false	Whether to enable Iceberg Native write. This is a Spark-specific parameter.

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
checkencoding	low	The UTF-8 encoding verification behavior during string reading. The options are as follows: The value **no** indicates that the encoding verification is not performed. The value **low** indicates that the encoding verification is performed and invalid characters are replaced. The value **high** indicates that the encoding verification is performed and an error is reported when there are invalid characters. This is an Aura-specific parameter.

Catalog Properties

Property	Default Value	Description
catalog-impl	org.apache.iceberg.lakeformation.LakeFormationCatalog	Custom catalog implementation class used by the compute engine
io-impl	null	Custom file I/O implementation class used by the catalog
warehouse	null	Root path of the data warehouse
uri	null	Uniform resource identifier, for example, the address of the Hive metadata service
clients	2	Size of the client connection pool
cache-enabled	true	Whether to enable catalog entry caching

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
cache.expiration-interval-ms	30000	Time to live (TTL) of the locally cached catalog entries, in milliseconds. The value **0** indicates that the cache is disabled, and a negative value indicates permanent cache.

1.1.3 Iceberg Catalog

Iceberg Catalog serves as the top-tier component within Iceberg, tasked with overseeing the metadata and its associated operations across all Iceberg tables. This Catalog governs both the structure and metadata of these tables, providing interfaces essential for their creation, query, and modification—acting as the gateway through which you engage with Iceberg tables. Through it, users can pinpoint the exact location of the current metadata file for each table, making it an indispensable element for both reading from and writing to Iceberg tables.

The current AI DataLake version supports using LakeFormation Catalog as the Catalog component for Iceberg tables.

LakeFormation Catalog

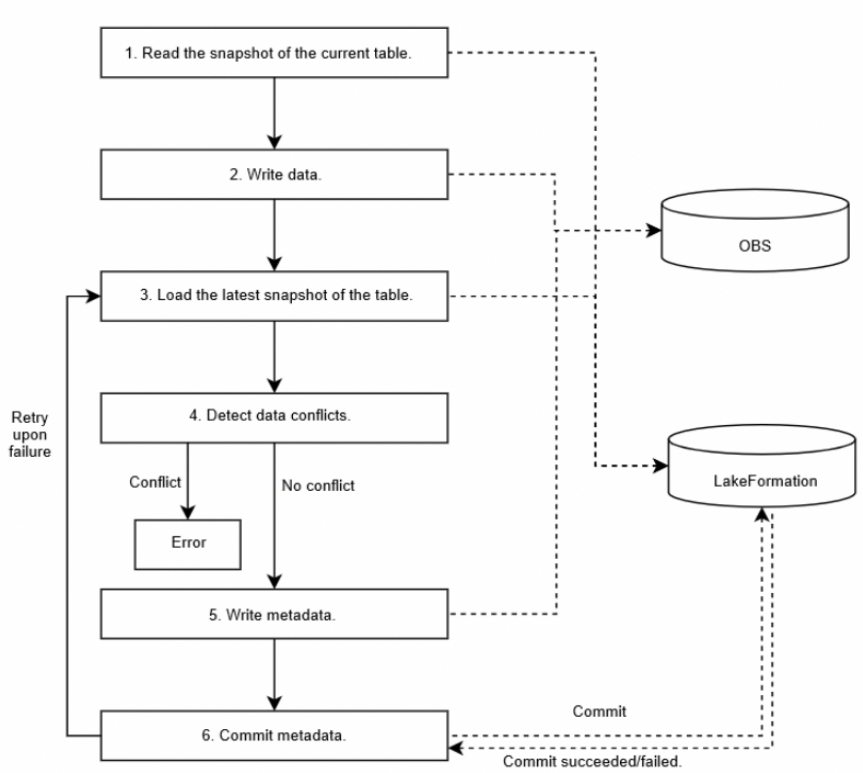
LakeFormation Catalog relies on the LakeFormation metadata service to oversee the most recent snapshots.

It uses an optimistic concurrency control (OCC) mechanism to maintain data integrity during concurrent writes across multiple tenants.

All Iceberg tables currently created on AI DataLake are LakeFormation Catalog tables.

The following figure outlines the concurrency handling process for LakeFormation Catalog commits.

 Copyright © Huawei Technologies Co., Ltd.



1. Read the current table snapshot information from LakeFormation, including the latest metadata file path and snapshot ID.
2. Write new data to the table based on the current snapshot.
3. Load the latest snapshot of the table.
4. Detect data conflicts. If a conflict occurs, the statement execution fails. Otherwise, attempt to commit the transaction.
5. Write metadata, including manifest files, manifest list, and metadata file.
6. Submit the latest metadata file path and snapshot ID to LakeFormation. If the submission fails, retry from step 3.

1.1.4 Schema Evolution of Iceberg Tables

Iceberg supports in-place table evolution. You can evolve a table schema simply by modifying the metadata, without the need to create a table or rewrite data.

Schema Evolution

Iceberg supports the following schema evolution changes:

- **Add**: Add a new column to the table or to a nested structure.
- **Drop**: Remove an existing column from the table or a nested structure.
- **Rename**: Rename an existing column or field in a nested structure.
- **Update**: Widen the type of a column, structure field, map key, map value, or list element. Note that map keys do not support adding or dropping structure fields that would change equality.
- **Reorder**: Change the order of columns or fields in a nested structure.

 Copyright © Huawei Technologies Co., Ltd.

Iceberg schema updates are metadata changes, so no data files need to be rewritten to perform the update.

Evolution Scope Supported by Data Types

- Table 1-4 lists the evolution scope supported by primitive data types.

  Table 1-4 Evolution scope supported by primitive data types

Primitive Type	v1 and v2 Valid Type Promotion	Requirements
unknown	-	-
int	long	-
date	-	Promotion to **timestamptz** or **timestamptz_ns** is not allowed. Values outside the promoted type's range will result in a runtime failure.
float	double	-
decimal(P, S)	decimal(P', S) (P' > P)	Enhance precision only.

- The evolution scope supported by the struct type is as follows:
  - Delete a field from a struct.
  - Add a field to a struct.
  - Rename a field in a struct.
  - Reorder fields in a struct.
  - Evolve the fields in a struct (for details, see **Table 1-4**).

1.1.5 Data Types Supported by Iceberg Tables

Primitive Types

Supported primitive types are defined in **Table 1-5**.

 Copyright © Huawei Technologies Co., Ltd.

CAUTION

- Timestamp values without time zone represent a date and time of day regardless of zone. The time value is independent of zone adjustments (for example, 2017-11-16 17:10:34 is always retrieved as 2017-11-16 17:10:34).
- Timestamp values with a time zone (Timestamptz) represent a point in time. They are stored as UTC and do not retain a source time zone (for example, 2017-11-16 17:10:34 PST is stored/retrieved as 2017-11-17 01:10:34 UTC and these values are considered identical).
- Character strings must be stored as UTF-8 encoded byte arrays.

Table 1-5 Primitive data types

Primitive Type	Description
boolean	True or false
int	32-bit signed integers. They can promote to the long type.
long	64-bit signed integers
float	32-bit IEEE 754 floating points. They can promote to the double type.
double	64-bit IEEE 754 floating points
decimal(P,S)	Fixed-point decimals. **P** specifies the precision (the total number of digits), and **S** specifies the scale (the number of digits in the fractional part). Scale is fixed, and precision must be 38 or less.
date	Calendar dates without the time zone or time
timestamp	Timestamps with microsecond precision, excluding the time zone
timestamp_tz	Timestamps with microsecond precision and the time zone
string	Arbitrary-length character sequences encoded with UTF-8
binary	Arbitrary-length byte arrays
geometry	Plane geometric types, including two-dimensional spatial data for points, lines, and planes

 Copyright © Huawei Technologies Co., Ltd.

Primitive Type	Description
geography	Geographic coordinate types, including spherical geographic location data based on longitude and latitude
timestamp_ns	Timestamp with nanosecond precision, excluding the time zone
timestamp_tz_ns	Timestamp with nanosecond precision and the time zone
unknown	Unknown type (used only as a placeholder when the schema is not determined). It can only be NULL and is not written to data files.

Nested Types

- A struct is a tuple of typed values. Each field in the tuple is named and has an integer ID that is unique in the table schema. Each field can be either optional or required, meaning that values can (or cannot) be null. Fields can be of any type and may have an optional comment or doc string. They can have default values.
- A list is a collection of values with some element type. The element field has an integer ID that is unique in the table schema. Elements can be either optional or required. They can be of any type.
- A map is a collection of key-value pairs with a key type and a value type. Both the key field and value field have an integer ID that is unique in the table schema. Map keys are required and map values can be either optional or required. Both map keys and map values may be any type, including nested types.

Semi-structured Types

A variant is a value that stores semi-structured data. The structure and data types in a variant are not necessarily consistent across rows in a table or data file. The variant type and binary encoding are defined in the Parquet project, with support currently available for v1. Support for variants is added in Iceberg v3.

- Variants are similar to JSON with a wider set of primitive values including dates, timestamps, timestamptz, binary, and decimals.
- Variant values may contain nested types:
  - An array is an ordered collection of variant values.
  - An object is a collection of fields, each of which contains a string key and a variant value.
- As a semi-structured type, variants differ from other Iceberg data types in the following ways:
  - Variant arrays are similar to lists, but may contain any variant value rather than a fixed element type.

 Copyright © Huawei Technologies Co., Ltd.

  - Variant objects are similar to structs, but may contain variable fields identified by name and field values may be any variant value rather than a fixed field type.

1.1.6 Iceberg Hidden Partitioning and Partition Evolution

Hidden partitioning is an automatic partitioning mechanism built into Iceberg. A table automatically calculates partitions based on original fields (such as timestamps) using built-in functions. You do not need to manually maintain partition columns. In this way, query logic and physical storage are decoupled, and you can directly modify partitioning rules without migrating data.

Typical application scenarios:

- Log, traces, and time series data is partitioned by day, hour, or month.
- Multi-dimensional partitioning (time and level, type, or business line) is required.
- The partition policy of a table needs to be modified, but data does not need to be rewritten.
- Multi-person collaboration is required to avoid incorrect manual partitioning.

The core of Iceberg hidden partitioning is to automatically calculate partitions based on original fields. You do not need to maintain physical partition columns. This approach significantly improves query efficiency through partition pruning, decouples storage from logic, and supports flexible evolution of partitioning schemes without requiring data migration.

Hidden Partition Transformations

Table 1-6 lists the hidden partition transformations.

- All transforms must return null for a null input value.
- The **void** transform may be used to replace the transform in an existing partition field so that the field is effectively dropped in v1 tables. For details, see **Partition Evolution**.
- **bucket[N]**: Applies a hash function (typically Iceberg-specific) to the source value, then computes the modulus with N to assign a bucket number from 0 to N–1.

  Different data types (for example, int and string) produce different hash values, even for logically equivalent values like 34 and "34". As a result, type promotion is not supported.
- truncate[W]
  - For numeric types (int, long, and decimal), values are truncated to the greatest number that is both no greater than the original values and a multiple of *W*. For example, **truncate[10]** converts 17 to 10.
  - For string and binary types, values are truncated to the first *W* characters or bytes. For example, **truncate[3]** converts **"iceberg"** to **"ice"**.

 Copyright © Huawei Technologies Co., Ltd.

Table 1-6 Hidden partition transformations

Transform Name	Description	Source Type	Result Type
identity	Use the source value, which is unmodified.	Any except for geometry, geography, and variant	Source type
bucket[N]	Hash the value and take the result modulo *N*.	int, long, decimal, date, time, timestamp, timestamptz, timestamp_ns, timestamptz_ns, string, uuid, fixed, binary	int
truncate[W]	Truncate the value to width *W*.	int, long, decimal, string, binary	Source type
year	Extract the year from the date or timestamp.	date, timestamp, timestamptz, timestamp_ns, timestamptz_ns	int
month	Extract the month from the date or timestamp.	date, timestamp, timestamptz, timestamp_ns, timestamptz_ns	int
day	Extract the day from the date or timestamp.	date, timestamp, timestamptz, timestamp_ns, timestamptz_ns	int
hour	Extract the hour from the timestamp.	timestamp, timestamptz, timestamp_ns, timestamptz_ns	int
void	Always produce null.	Any data type	Source type or int

Partition Evolution

Table partitioning can be evolved by adding, removing, renaming, or reordering partition spec fields.

Changing a partition spec produces a new spec, which is identified by a unique spec ID. The new spec is added to the partition spec list of the table and can be set as the default spec for the table.

Evolution rules:

 Copyright © Huawei Technologies Co., Ltd.

- When evolving a spec, changes should not cause partition field IDs to change because the partition field IDs are used as the partition tuple field IDs in manifest files.
- In v2, partition field IDs must be explicitly tracked for each partition field. New IDs are assigned based on the last assigned partition ID in table metadata.
- In v1, partition field IDs were not tracked, but were assigned sequentially starting at 1000 in the reference implementation. This assignment caused problems when reading metadata tables based on manifest files from multiple specs because partition fields with the same ID may contain different data types. For compatibility with old versions, the following rules are recommended for partition evolution in v1 tables:
  - Do not reorder partition fields.
  - Do not drop partition fields. Instead replace the field's transform with the void transform.
  - Only add partition fields at the end of the previous partition spec.

1.1.7 Iceberg Liquid Clustering

Liquid Clustering is a data organization strategy for Iceberg tables. After a clustering column is specified, Liquid Clustering physically groups data files with similar clustering column values to improve data read efficiency during query. Iceberg organizes these grouped data files using cubes as the basic management unit.

Liquid Clustering serves as an alternative to partitioned tables. Instead of relying on fixed partition directories, Liquid Clustering automatically clusters data based on clustering columns during writes and incrementally maintains the data layout. It adapts dynamically as data distribution changes. When data in a table changes, it can locally optimize the data layout without requiring a full-table repartition.

Table 1-7 Comparison between partition tables and Liquid Clustering

Dimension	Partitioned Tables	Liquid Clustering
Clustering dimension	Partition keys are limited by cardinality, primarily one-dimensional.	A clustering key can contain 2 to 4 columns, allowing for a mix of high and low cardinality. There is no directory splitting or metadata expansion.
Layout evolution	Modifying a partition key requires rewriting historical data.	Modifying a clustering key affects only newly written data. Old cubes with mismatched clustering columns are retained by default and can be rewritten using rewrite-all only when necessary, eliminating the need for full table migration.

 Copyright © Huawei Technologies Co., Ltd.

Dimension	Partitioned Tables	Liquid Clustering
Multidimensional filtering and pruning	Pruning occurs only along partition fields.	File-level data skipping allows multiple clustering keys to be skipped at the same time, greatly reducing the scanning workload.
File size maintenance	Compaction is fully manually scheduled.	Incremental maintenance: determined based on cube health (partial cubes smaller than **min-cube-size** are merged, and those exceeding the **target** size are split). Only unhealthy cubes are rewritten, and rest are skipped.
Metadata and catalog	Partition directories expand as cardinality grows.	There are no physical partition directories. Cube mappings are converged into a single sidecar JSON file, and the **liquid_cubes** metadata table is provided to intuitively monitor the cube health status.
Sorting and O&M	Partition pruning and Z-order mechanisms are combined.	The built-in Hilbert curve sorting ensures data locality and is executed through the **rewrite_data_files** entry. It is mutually exclusive with the **sort**/**zorder** policy to avoid mixing multiple O&M logics.

Application Scenarios

- Tables with high-cardinality columns that are frequently used in query filters.
- Tables with severe data skew.
- Tables with rapidly increasing data volumes that require continuous maintenance and optimization.

Notes and Constraints

- The format version of the Iceberg table must be **2** or later.
- Two to four clustering columns are required.
- The Iceberg table must be non-partitioned because Liquid Clustering is an alternative to partitioned tables.
- Duplicate clustering columns are not allowed.
- Clustering columns cannot be deleted from the schema.
- The **binpack**/**sort**/**zorder** policy and **partial-progress** mode are incompatible with Iceberg tables that have Liquid Clustering enabled.

 Copyright © Huawei Technologies Co., Ltd.

Cube

A Cube is the basic management unit for organizing data in Liquid Clustering. It refers to a logical group of data files generated under the same set of clustering rules. Each Cube contains the following:

- **Cube ID**: A universally unique identifier (UUID).
- **Data file list**: All data files belonging to the Cube.
- **Clustering column ID**: The clustering column used to generate the Cube, stored as a field ID.
- **Total data volume**: Total size of all data files within the Cube, in bytes.
- **Associated snapshot**: The Iceberg snapshot of the Cube.

Cube Health

As data is continuously written, files within a Cube may become scattered or no longer comply with the current clustering specification. To address this, Iceberg compares each Cube with the clustering specification that takes effect in the table and outputs the health status of the Cube. The health status reflects the clustering effectiveness of the data within the Cube. Background tasks use this status to determine whether to perform a re-clustering rewrite operation on the Cube.

Table 1-8 Cube health status

Health Status	Description	Whether to Be Rewritten
Healthy	The total size of the Cube is greater than or equal to the value of the **min-cube-size-bytes** configuration item, and the clustering columns of the Cube exactly match the current clustering specification of the table.	Rewriting is performed only when **rewrite-all** is set to **true**.
Partial	The total size of the Cube is smaller than the value of **min-cube-size-bytes**. The Cube is too small and needs to be merged and rewritten to improve read efficiency.	Rewriting is always performed.
column_mismatch	The clustering columns used for creating the Cube do not match the active clustering specification of the table.	Rewriting is performed only when **rewrite-all** is set to **true**.

Configuration Items

Liquid Clustering is configured through Iceberg table properties.

 Copyright © Huawei Technologies Co., Ltd.

Table 1-9 Configuration items

Table Property	Description	Default Value
write.liquid-clustering.columns	List of clustering columns, which are separated by commas (,).	None
write.liquid-clustering.target-cube-size-bytes	Target size of each Cube, in bytes. Files are grouped into Cubes. The size of a single Cube is not recommended to exceed this value.	10 GB
write.liquid-clustering.min-cube-size-bytes	Minimum threshold of a Cube, in bytes. Cubes whose size is less than this value will be marked as **Partial** and trigger rewriting.	5 GB

Clustering Column Selection

When specifying clustering columns, prioritize the columns most frequently used in query filter conditions. Multiple clustering columns can be defined in any order. If two columns are highly correlated, only one of them can be designated as a clustering column.

The following data types are supported for clustering columns:

- INTEGER
- LONG
- FLOAT
- DOUBLE
- DECIMAL
- DATE
- TIMESTAMP
- STRING

1.2 Iceberg on Aura

1.2.1 Preparations Before Using Iceberg

Setting GUC

The optional GUC parameter **enable_meta_scan** is used to optimize the query performance.

This parameter is enabled by default. However, if the table contains a small amount of data, disabling this parameter may improve the query performance. Enable or disable this GUC parameter based on your actual needs.

Sample code:

 Copyright © Huawei Technologies Co., Ltd.

```
SHOW enable_meta_scan;
SET enable_meta_scan=off;
SET enable_meta_scan=on;
or
SET enable_meta_scan=true;
SET enable_meta_scan=false;
```

1.2.2 Iceberg Configuration and Types

Mapping between Aura types and Iceberg types

Table 1-10 Iceberg types corresponding to Aura types

Aura	Iceberg
smallint	integer
tinyint	integer
integer	integer
bigint	long
numeric	decimal
decimal	decimal
real	float
float4	float
double precision	double
float8	double
boolean	boolean
char	string
varchar	string
text	string
date	date
timestamp without time zone	timestamp without time zone
bytea	binary
array	list
struct	struct

 Copyright © Huawei Technologies Co., Ltd.

Table 1-11 Aura types corresponding to Iceberg types

Iceberg	Aura
boolean	boolean
integer	integer
long	long
float	float
double	double
date	date
time	Not supported
timestamp with time zone	Not supported
timestamp without time zone	timestamp without time zone
string	string
uuid	Not supported
fixed	Not supported
binary	bytea
decimal	decimal
struct	struct
list	array
map	Not supported
nanosecond timestamp	Not supported
nanosecond timestamp with time zone	Not supported
unknown	Not supported
variant	Not supported
geometry	Not supported
geography	Not supported

1.2.3 Iceberg on Aura DDL Syntax Description

Creating an Iceberg Table

You can create an Iceberg table using the **CREATE TABLE** syntax. Unlike tables in other formats, there are no special parameters required for creating an Iceberg

 Copyright © Huawei Technologies Co., Ltd.

table—simply specify **STORE AS ICEBERG**. For details about the syntax, see **CREATE TABLE**.

Example:

```
CREATE TABLE iceberg_ext(
col1 int,
col2 varchar(20),
...
)PARTITION BY (col3 bigint)
TABLEPROPERTIES (
'write.metadata.delete-after-commit.enabled'='true',
'write.metadata.previous-versions-max' = '10'
) STORE AS ICEBERG;
```

The parameter **write.metadata.delete-after-commit.enabled** controls whether to delete the metadata files of older versions after committing a transaction, which defaults to **false**. When this parameter is set to **true**, you can use **write.metadata.previous-versions-max** to specify the number of metadata files to retain.

Exercise caution when using these parameters. While deleting metadata files can conserve storage space, ensure that data consistency and availability remain unaffected.

Clearing an Iceberg Table

You can use the **TRUNCATE TABLE** syntax to clear data from a table. For details, see **TRUNCATE**.

Example:

```
TRUNCATE TABLE iceberg_ext;
```

Deleting an Iceberg Table

Executing the **DROP TABLE** syntax will remove both the metadata and data associated with an Iceberg table (note that for managed tables, their metadata and data can be restored through the LakeFormation console post-deletion). For details about the syntax, see **DROP TABLE**.

Example:

```
DROP TABLE iceberg_ext;
```

Modifying Table Columns

Using the **ALTER TABLE** command with clauses like **ADD COLUMN**, **RENAME COLUMN TO**, **ALTER COLUMN**, and **DROP COLUMNS**, you can perform schema evolution tasks such as adding, renaming, updating, reordering, or removing columns in Iceberg tables.

NOTE

Iceberg assigns each column a unique **fieldId**. When writing data, this **fieldId** is embedded in the file's schema. During reads, the metadata maps columns using **fieldId**, enabling reading post-schema evolution. For non-Iceberg files lacking **fieldId**, mapping relies on column names. If column names are reused during schema evolution, it may cause type mismatches, read failures, or retrieval of outdated data.

 Copyright © Huawei Technologies Co., Ltd.

Example:

Create an Iceberg partitioned table.

```
CREATE TABLE t_iceberg(a int, b array<struct<x:int, y:int>>) partition by (c int) store as iceberg;
```

- Adding columns:

  Iceberg tables allow adding one or more columns to a table or nested fields with complex types. You need to specify the column name, type, optional comments, and position. Comments cannot be added when inserting into nested fields with complex types.

  ```
  -- Add to a table.
  alter table t_iceberg add column d int, e float comment 'this is e', f array<int> after a, g
  struct<x:int,y:int> comment 'this is g' before b;
  -- Add to a nested field with complex types. Use element to specify the sub-type of the array.
  alter table t_iceberg add column b.element.z int after x;
  ```

- Renaming columns:

  Iceberg tables allow renaming table columns or nested fields with complex types.

  ```
  alter table t_iceberg rename column a to newa;
  alter table t_iceberg rename column b.element.x to newx;
  ```

- Updating columns:

  Iceberg tables allow updating types, comments, and positions for table columns, as well as types and positions for nested fields.

  ```
  alter table t_iceberg alter column a type bigint comment 'this is a' after c, c type bigint comment 'this is c' after a, b.element.y type bigint after first;
  ```

  Currently, only upgrades from lower-precision to higher-precision types are allowed:

  – Integer types: From SMALLINT to INT or BIGINT, or from INT to BIGINT.
  – Floating-point types: From FLOAT4 to FLOAT8.
  – NUMERIC type: From a lower precision to a higher precision. The scale cannot be updated.
  – String types: Length increases of VARCHAR or CHAR.

  > **CAUTION**
  >
  > Floating-point storage is imprecise. When upgrading from FLOAT4 to FLOAT8, decimal precision may change, affecting results.

- Dropping columns:

  Iceberg allows dropping table columns or nested fields within complex types. However, you cannot delete partition columns or remove all columns from a table or all fields from a complex type.

  ```
  alter table t_iceberg drop columns (a, b.x);
  ```

Modifying Table Properties

You can modify the properties of a table using the **SET/UNSET TABLEPROPERTIES** capability of the **ALTER TABLE** syntax.

Example:

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER TABLE iceberg_ext SET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled' = 'false');
ALTER TABLE iceberg_ext UNSET TABLEPROPERTIES ('write.metadata.delete-after-commit.enabled');
```

1.2.4 Iceberg Table Query and Write

Querying an Iceberg Table

You can query data in an Iceberg table. For details about the syntax, see **SELECT**.

NOTE

Iceberg v3 tables support the query of system columns **_rowid** and **_last_updated_sequence_number**.

Note:

Only the latest committed data can be queried.

Example:

```
SELECT * FROM iceberg_ext order by col1;
```

Writing to an Iceberg Table

You can insert one or more rows of data to an Iceberg table. For details about the syntax, see **INSERT**.

Note:

- Frequent insertion of small data may lead to the issue of small files. You are advised to periodically rewrite data/manifest files to maintain storage and query efficiency.
- When a DML transaction is committed, the engine may automatically schedule a background manifest rewrite (compaction) to reduce the number of small manifest files and help prevent query performance degradation over time.
- Currently, data files can only be written in Parquet format.

Example:

```
INSERT INTO iceberg_ext VALUES (1, 'a', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT OVERWRITE INTO iceberg_ext VALUES (1, 'b', 1.1234567, 100.11, '2025-03-03', '2025-03-03
16:00:00');
INSERT INTO iceberg_ext (col1, col2, col3, col4, col5, col6) SELECT col1, col2, col3, col4, col5, col6 FROM
iceberg_table WHERE col1 > 18;
```

Deleting Data from an Iceberg Table

You can delete one or more rows from an Iceberg table. For details about the syntax, see **DELETE**.

Note:

- Currently, only the merge-on-read approach is supported for delete operations.
- When performing deletions concurrently with updates or other deletions, specify partition conditions to minimize conflicts.

 Copyright © Huawei Technologies Co., Ltd.

- For deleting entire partitions, the operation may be optimized by marking metadata states. To check if optimization applies, use **EXPLAIN DELETE** and look for the keyword "Do speed up for delete". If enabled, the returned row count might exceed the actual number of deleted rows.

Example:

```
DELETE FROM iceberg_ext where a = 1;
```

Updating Data in an Iceberg Table

You can update one or more rows in an Iceberg table. For details about the syntax, see **UPDATE**.

Note:

- Currently, only the merge-on-read approach is supported for update operations.
- When updating data concurrently with deletions or other updates, specify partition conditions to minimize conflicts.

Example:

```
UPDATE iceberg_ext set b = 1 where a = 1;
```

Merging Iceberg Tables

You can merge one Iceberg table with another. For details about the syntax, see **MERGE INTO**.

Note:

Currently, only the merge-on-read approach is supported for merge operations.

Example:

```
MERGE into t_iceberg p1 using t_iceberg p2 on (p1.a = p2.c) when matched then delete;
```

1.2.5 Expression Partitioning

When working with large datasets, data partitioning is a key technique for improving query performance. However, traditional partitioning methods require you to manually define partition logic, which not only increases user effort but also limits flexibility. Expression partitioning supports Iceberg's hidden partitioning feature by abstracting the "partition logic" from the user perspective and delegating it to Iceberg metadata management. How can we ensure both efficiency and flexibility in data partitioning? With expression partitioning, you no longer need to manage complex partition logic—Iceberg automatically optimizes partitioning strategies based on data characteristics and query patterns, enhancing query performance and data management efficiency.

Constraints

- A maximum of four partition columns are supported.
- Void expressions are not supported.
- Complex data types are unsupported for all partition expressions.

 Copyright © Huawei Technologies Co., Ltd.

- Truncate and bucket expressions do not support binary data.
- The **INSERT INTO PARTITION** function only works with identity expressions; other expressions are unsupported.
- Iceberg constraint: A table column cannot appear in multiple time expressions such as **year**, **month**, **day**, or **hour**. However, it can coexist in time and bucket/truncate expressions.
- Iceberg constraint: Partition keys follow default naming rules and cannot conflict with existing column names in the table.
- Partition evolution: Deleting all partition keys from a partitioned table is not supported.
- Partition evolution: Only updating partition keys on a partitioned table is allowed. Non-partitioned tables do not support this operation.
- Partition evolution: Renaming or assigning names to partition keys is unsupported.

Supported Partition Expressions

Table 1-12 lists the partition expressions supported by Aura and their corresponding data types.

Table 1-12 Partition expressions supported by Aura and their corresponding data types

Expression	Supported Data Type	Result Type	Description
identity	smallint, int, bigint, long, numeric, date, timestamp, text, varchar, bpchar	Same as input type	Does not support float, double, boolean, or bytea.
bucket[N]	smallint, int, bigint, long, numeric, date, timestamp, text, bpchar, varchar	int	• Computes the hash value of the output and applies modulo operation. Results fall within the range [0, N). • Uses MurmurHash3 algorithm.
truncate[W]	smallint, int, bigint, long, numeric, text, bpchar, varchar	Same as input type	• Does not support bytea. • For details, see **Partition Expression: truncate**.
year	date, timestamp	int	Example: year('2025-10-10') returns 2025.
month	date, timestamp	int	Example: month('2025-10-10') returns 2025-10.
day	date, timestamp	int	Example: day('2025-10-10') returns 2025-10-10.
hour	timestamp	int	Example: hour('2025-10-10 10:10:10.000') returns 2025-10-10 10:00:00.

 Copyright © Huawei Technologies Co., Ltd.

Expression	Supported Data Type	Result Type	Description
void	Not supported	Not supported	Maps all values to null.

All expressions return null when processing null inputs.

The **identity** expression outputs the input value without modification.

The **year** expression calculates the number of years since 1970. The **month** expression calculates the number of months since January 1, 1970. The **day** expression calculates the number of days since January 1, 1970. The **hour** expression calculates the number of hours since January 1, 1970, at 00:00:00. When recording partition values, Iceberg converts integer results into human-readable time strings. For example, instead of storing the integer 55, Iceberg records the result of year('2025-10-10') as the string '2025'.

Aura offers SQL functions **fq_iceberg_year()**, **fq_iceberg_month()**, **fq_iceberg_day()**, **fq_iceberg_hour()**, **fq_iceberg_truncate()**, and **fq_iceberg_bucket()** that correspond to the Iceberg partition expressions **year**, **month**, **day**, **hour**, **truncate**, and **bucket**.

Partition Expression: truncate

The **truncate** expression behaves differently depending on the data type involved. To compute its value, you can use the function **fq_iceberg_truncate()**.

For integer types, **truncate(W, integer_value)** returns the largest number that does not exceed **integer_value** and is a multiple of **W**. Here is an example:

```
select fq_iceberg_truncate (10, -2) ;
fq_iceberg_truncate
---------------------
                -10
(1 row)

select fq_iceberg_truncate (10, 0) ;
fq_iceberg_truncate
---------------------
                  0
(1 row)

select fq_iceberg_truncate (10, 9) ;
fq_iceberg_truncate
---------------------
                  0
(1 row)

select fq_iceberg_truncate (10, 10) ;
fq_iceberg_truncate
---------------------
                 10
(1 row)

select fq_iceberg_truncate (10, 19) ;
fq_iceberg_truncate
---------------------
                 10
(1 row)
```

 Copyright © Huawei Technologies Co., Ltd.

For string types, **truncate(W, string_value)** returns the prefix substring of **string_value** with a length of **W** characters. Note that the length is measured in characters. Here is an example:

```
select *, length(t)  from fq_iceberg_truncate (2, '') as t;
iceberg_truncate | length
------------------+--------
                 |      0
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'a') as t;
iceberg_truncate | length
-----------------+--------
a                |      1
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'tt') as t;
iceberg_truncate | length
-----------------+--------
tt               |      2
(1 row)

select *, length(t)  from fq_iceberg_truncate (2, 'cow') as t;
iceberg_truncate | length
-----------------+--------
co               |      2
(1 row)

 select *, length(t)  from fq_iceberg_truncate (2, 'Hello, China!') as t;
 iceberg_truncate | length
------------------+--------
 Hello            |      2
(1 row)
```

For numeric types, **truncate(W, num_value)** returns the largest number that does not exceed **num_value**, where shifting the decimal point to the right by scale positions results in a multiple of **W**. Here is an example:

```
select fq_iceberg_truncate (50, 10.64::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.600
(1 row)

select fq_iceberg_truncate (50, 10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
              10.650
(1 row)

select fq_iceberg_truncate (50, -10.66::numeric(15,3));
 fq_iceberg_truncate
---------------------
             -10.700
(1 row)
```

Creating an Expression-Partitioned Table

For detailed CREATE TABLE syntax, see **CREATE TABLE**.

Below is an example SQL statement for creating an expression-partitioned Iceberg-managed table:

```
set current_schema to test;
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
CREATE TABLE
```

 Copyright © Huawei Technologies Co., Ltd.

```
select * from pg_get_partition_keys('test', 'iceberg_table');
     table     |  partition_key
---------------+-----------------
 iceberg_table | bucket(16,val)
 iceberg_table | truncate(10,id)
 iceberg_table | c3
 iceberg_table | year(ts)
(4 rows)
```

In this example, c3 serves as an identity partition key. Additionally, three more partition keys are created using **bucket**, **truncate**, and **year** expressions. Iceberg automatically assigns internal names to these partition keys based on the rules outlined in the table below.

Table 1-13 Default internal naming rules for expression partition keys in Iceberg

Expression	Default Naming Rule	Example
year(source_column)	<source_column>_year	year(ts) -> 'ts_year'
month(source_column)	<source_column>_month	month(ts) -> 'ts_month'
day(source_column)	<source_column>_day	day(ts) -> 'ts_day'
hour(source_column)	<source_column>_hour	hour(ts) -> 'ts_hour'
bucket(N, source_column)	<source_column>_bucket	bucket(ts) -> 'ts_bucket'
truncate(W, source_column)	<source_column>_trunc	truncate(ts) -> 'ts_trunc'
identity(source_column)	<source_column>	identity(ts) -> 'ts'

Column names must not conflict with partition key names. Otherwise, the table creation will fail with an error.

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), ts_year bigint ) store as iceberg;
ERROR:  column "ts_year" specified more than once
```

Additionally, Iceberg does not allow a time-type column to be used simultaneously in multiple expressions such as **year**, **month**, **day**, and **hour**. However, it can still be used in other expressions like **truncate** or **bucket**.

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), month(ts) ) store as iceberg;
ERROR:  java.lang.IllegalArgumentException: Cannot add redundant partition: 1002: ts_year: year(4)
conflicts with 1003: ts_month: month(4).

CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val),  c3 int comment 'this is c3 field', year(ts), bucket(7, ts) ) store as iceberg;
CREATE TABLE
```

Aura supports up to four partition keys per table. Exceeding this limit will cause table creation to fail with an error.

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 int comment 'this is c3 field', year(ts) ) store as iceberg;
ERROR:  partitioned table should have no more than 4 partition keys
```

 Copyright © Huawei Technologies Co., Ltd.

Aura does not support complex types as partition keys. Using them will cause table creation to fail with an error.

```
CREATE TABLE iceberg_table(id bigint, data text, category text, ts timestamp, val numeric) partition by
(bucket(16, val), truncate(10, id), c3 struct<c: text, d: int>, year(ts) ) store as iceberg;
ERROR:  Complex type cannot be served as a value-partitioning column.

CREATE TABLE iceberg_table(a int , b struct<c: text, d: int>) PARTITION BY(bucket(5, b.c), truncate(3, b.d))
store as ICEBERG;
ERROR:  syntax error at or near "."
LINE 1: ... struct<c: text, d: int>) PARTITION BY(bucket(5, b.c), trunc...
                                                             ^
```

Aura also supports creating expression-partitioned foreign Iceberg tables, but additional conditions must be met:

- The number of partition keys must match those of the Iceberg table stored in OBS.
- The definition of each partition key—including its order, expression, and associated column—must exactly align with the corresponding Iceberg table in OBS.

The following sections will use the **iceberg_test_part_expr** table as an example, defined as follows:

```
set current_schema to test;
CREATE TABLE iceberg_test_part_expr(id bigint, data text, category text, ts timestamp) partition by (c3 int,
truncate(4, data), year(ts)) store as iceberg;
CREATE TABLE
```

Importing Data

An Iceberg table with expression partitioning supports both single-row insertion and batch data import.

```
INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'hellohello',   'shanxi', '2025-08-29', 1);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'he',           'shanxi', '2024-09-29', 20);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'helloworld',   'shanxi', '2023-06-29', 6);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'hella',        'shanxi', '2022-05-29', 60);
INSERT 1
INSERT INTO iceberg_test_part_expr values(9, 'zhangmingmin', 'shanxi', '2025-04-29', 100);
INSERT 1
```

For the **INSERT INTO PARTITION** feature, Aura only works with tables where all partition keys are of the **identity** type. Tables using other types of expressions for partitioning are not supported.

```
INSERT INTO iceberg_test_part_expr PARTITION(c3=1, data='hell', ts='2025') values (9, 'hellohello',
'shanxi', '2025-08-29 01:05:59.550463') ;
ERROR:  INSERT INTO PARTITION clause only support identity partition.
```

To view all partitioned data in a table, you can use the function **pg_get_partitions()**, as shown below:

```
select partition_name from pg_get_partitions('iceberg_test_part_expr');
     partition_name
---------------------------
 c3=1/data=hell/ts=2025
 c3=20/data=he/ts=2024
 c3=100/data=zhan/ts=2025
```

 Copyright © Huawei Technologies Co., Ltd.

```
 c3=6/data=hell/ts=2023
 c3=60/data=hell/ts=2022
(5 rows)
```

Query Optimization Based on Column Expressions

With support for expression partitioning, Aura enables column expression pushdown to optimize queries through static partition pruning. Here is an example:

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where c3 = 1;
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                           operation                                | A-rows
----+--------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                    |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr   |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      2

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (c3 = 1)
        Pushdown Predicate Filter: (c3 = 1)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Filter: (c3 = 1)
        Pushdown Predicate Filter: (c3 = 1)
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 2)
        DN read from: direct
(19 rows)
```

After pushing down the column expression c3 = 1, the number of scanned partitions is reduced from 6 to 2.

Query Optimization Based on SQL Functions

Following the implementation of expression partitioning, Aura supports pushdown optimization for whitelisted SQL functions. The currently supported SQL functions are listed below:

Table 1-14 Supported SQL functions for pushdown optimization

Partition Expression	SQL Function	Pushdown Condition
year	year(), date_trunc()	For **date_trunc()**, the time unit must be **year**.
month	date_trunc()	For **date_trunc()**, the time unit must be **month**.
day	date_trunc()	For **date_trunc()**, the time unit must be **day**.
hour	date_trunc()	For **date_trunc()**, the time unit must be **hour**.

 Copyright © Huawei Technologies Co., Ltd.

Partition Expression	SQL Function	Pushdown Condition
truncate	left(), substring()	For **substring()**, it must start from the beginning of the string.

Examples of static partition pruning using string functions **left()** and **substring()**:

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
substring(data, 1, 3) > 'hello';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
  1 | ->  Row Adapter                                               |      1
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      1
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      1

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: ("substring"(data, 1, 3) > 'hello'::text)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 1)
        DN read from: direct
(16 rows)

explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
left(data, 5) < 'hello';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
  1 | ->  Row Adapter                                               |      2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |      2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |      5

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: ("left"(data, 5) < 'hello'::text)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 5)
        DN read from: direct
(16 rows)
```

When **substring()** is pushed down, the number of partitions decreases from 6 to 1. With **left()**, the count reduces from 6 to 5.

Partition pruning works only for timestamp types, not for date types.

Similarly, examples using time-based functions like **date_trunc()**:

```
explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where
date_trunc('year', ts) > '2023-12-30 00:00:00';
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 id |                           operation                           | A-rows
----+---------------------------------------------------------------+--------
```

 Copyright © Huawei Technologies Co., Ltd.

```
  1 | ->  Row Adapter                                                |     4
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |     4
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |     4

                      Predicate Information (identified by plan id)
 ------------------------------------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (date_trunc('year'::text, ts) > '2023-12-30 00:00:00'::timestamp without time zone)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 4)
        DN read from: direct
(16 rows)

explain(analyze, predicate_only on, costs off, timing off) select * from iceberg_test_part_expr where year(ts)
< 2024;
                                QUERY PLAN
-----------------------------------------------------------------------------------------
 id |                          operation                          | A-rows
 ----+-------------------------------------------------------------------------------+--------
  1 | ->  Row Adapter                                                |     2
  2 |    ->  Partitioned Vector Foreign Scan on iceberg_test_part_expr        |     2
  3 |       ->  Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr |     2

          Predicate Information (identified by plan id)
 ------------------------------------------------------------------
  2 --Partitioned Vector Foreign Scan on iceberg_test_part_expr
        Filter: (year(ts) < 2024)
        Server Type: lf
        DN read from: direct
  3 --Partitioned Vector Foreign Meta Scan on iceberg_test_part_expr
        Server Type: lf
        Pruning results: (Manifests total: 6, Manifests left: 2)
        DN read from: direct
(16 rows)
```

When **date_trunc()** is pushed down, the number of partitions decreases from 6 to 4. With **year()**, the count reduces from 6 to 2.

Partition Evolution

Partition evolution enables in-place table evolution without rewriting existing data. It only affects newly inserted data moving forward.

Below are examples of deleting, adding, or directly overwriting partition keys:

```
ALTER TABLE iceberg_test_part_expr DROP PARTITION COLUMN c3;
ALTER TABLE iceberg_test_part_expr ADD PARTITION COLUMN c3;
ALTER TABLE iceberg_test_part_expr SET PARTITION COLUMN c3, year(ts);
select partition_key from pg_get_partition_keys('test', 'iceberg_test_part_expr');
  partition_key
-----------------
  c3
  year(ts)
(2 rows)
```

After enabling partition evolution, Iceberg's **INSERT OVERWRITE** behavior aligns with Spark's dynamic overwrite functionality. Specifically, expression partitions corresponding to columns in the **SELECT** query results will be overwritten. For more details, refer to **Spark Dynamic Overwrite**.

The example below demonstrates how this feature impacts results:

```
create table test_overwrite_tb ( a int, b int, c text) partition by (bucket(12, b )) store as iceberg ;
insert into test_overwrite_tb values(1,1,'1111'), (2,2, '222222');
```

 Copyright © Huawei Technologies Co., Ltd.

```
select partition, spec_id from iceberg_partitions('test_overwrite_tb');
  partition   | spec_id
--------------+---------
 {b_bucket=8} |       0
 {b_bucket=0} |       0
(2 rows)
select * from test_overwrite_tb ;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
(2 rows)

alter table test_overwrite_tb add partition column bucket(12, a );
alter table test_overwrite_tb drop partition column bucket(12, b );
insert overwrite into test_overwrite_tb select 11,1, 3333;
```

In the final **INSERT OVERWRITE**, the partition key is set as **bucket(12, a)**. Since this differs from the table's initial partition key, the operation does not overwrite existing partition data but instead writes to a new partition. The verification result is shown below:

```
select * from test_overwrite_tb;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
 11 | 1 | 3333
(3 rows)

select partition, spec_id from iceberg_partitions('test_overwrite_tb');
    partition    | spec_id
-----------------+---------
 {b_bucket=8}    |       0
 {a_bucket_12=3} |       2
 {b_bucket=0}    |       0
(3 rows)
```

If a new record with a = 11 is inserted afterward, all data within the current partition will be overwritten.

```
insert overwrite into test_overwrite_tb select 11,4, 4444 ;
select * from test_overwrite_tb ;
 a  | b |   c
----+---+--------
 11 | 1 | 1111
  2 | 2 | 222222
 11 | 4 | 4444
(3 rows)
```

1.2.6 Iceberg Table Services

Aura offers a range of table services that you can use as needed, for example, cleaning up old snapshots, organizing metadata and data files, to enhance storage and query efficiency. For details about the syntax, see **Iceberg Table Service Functions**.

Examples:

- Clean up old snapshots.

```
select * from iceberg_expire_snapshots('iceberg_test','2025-05-15 14:12:00+08');
deleted_data_files_count | deleted_position_delete_files_count | deleted_equality_delete_files_count |
deleted_manifest_files_count | deleted_manifest_lists_count | deleted_statistics_files_count
--------------------------+-------------------------------------+-------------------------------------
+------------------------------+------------------------------+---------------------------------
0                                | 0                                                                |
```

 Copyright © Huawei Technologies Co., Ltd.

```
0                                       | 0                                 |
3                                  | 0
(1 row)
```

- Rewrite metadata files.

```
SELECT * from iceberg_rewrite_manifests('test1');
rewritten_manifests_count | added_manifests_count
--------------------------+----------------------
2                         | 1
(1 row)
```

- Remove orphaned files.

```
SELECT * from iceberg_remove_orphanfiles('iceberg_test');
orphan_file_location
--------------------------------------------------------------
obs://xxxx/test.txt
obs://xxxx/test2.txt
(2 rows)
```

- Rewrite data files.

```
OPTIMIZE reason_t2 REWRITE DATA WITH OPTIONS ('rewrite-all' = 'true') WHERE TABLE_SC =
'Hainan';
INSERT 2
```

1.3 Iceberg on Spark

1.3.1 Interconnecting Spark with Iceberg Catalog

AI DataLake Spark allows you to use LakeFormation to manage metadata and authentication (row and column permissions) for Paimon and Iceberg tables. This simplifies data lake governance, enabling unified metadata management, fine-grained permission control, and transactional guarantees. Spark reads and writes to Paimon and Iceberg tables through data catalogs. You can select a data catalog based on your business scenarios.

Notes and Constraints

- **A LakeFormation instance has been created with metadata planned.**

  LakeFormation Catalog must be located in the same region as the AI DataLake workspace. Otherwise, they cannot be associated in subsequent processes.

- After a Paimon or Iceberg data catalog is configured, the job-level configurations take precedence over endpoint-level configurations. In addition, endpoint-level configurations take effect only after the endpoint is restarted.

Configuring a Data Catalog

Using the Default Data Catalog

Iceberg is used as the default data catalog for AI Data Lake Spark jobs. The system is pre-configured with **spark.aidatalake.lakehouse.default.format=iceberg**, so no manual configuration is required.

 Copyright © Huawei Technologies Co., Ltd.

Using a Custom Data Catalog

CAUTION

- Iceberg does not support user-defined functions (UDFs).
- If a non-customized catalog is selected, the system uses the configured default data catalog (specified by **spark.aidatalake.lakehouse.default.format**).

To use a custom data catalog, configure **spark.sql.catalog.{iceberg_catalog_name}=org.apache.iceberg.spark.SparkCatalog**, where **iceberg_catalog_name** should be replaced with your target data catalog name.

Configuration methods:

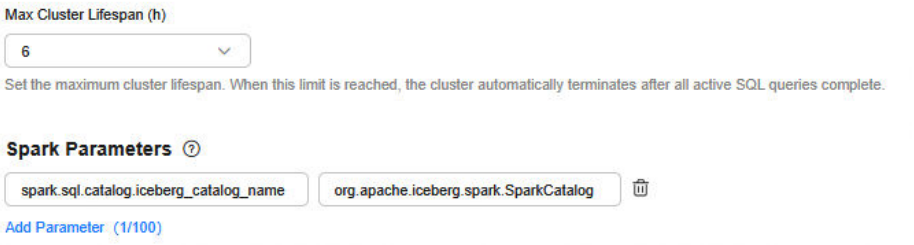
- Configuration via parameters in **spark_config** of the API

  For details about the API, see **RunSparkJob** or **RunSparkSql**. Example API request:

```
"endpoint_name": "spark-sql-test",
  "catalog_context": {
    "catalog_name": "iceberg_catalog_name",//The catalog name must be the same as that in the configuration. This catalog can be used only for Iceberg.
    "database_name": "default"
  },
  "statement": "SELECT * FROM iceberg_catalog_name.db.your_table LIMIT 10",//Three-part naming configuration is supported.
  "spark_config": {
    "spark.sql.catalog.iceberg_catalog_name": "org.apache.iceberg.spark.SparkCatalog"
  }
}
```

- Configuration via the endpoint creation or modification page
  - Configuration via the endpoint creation page

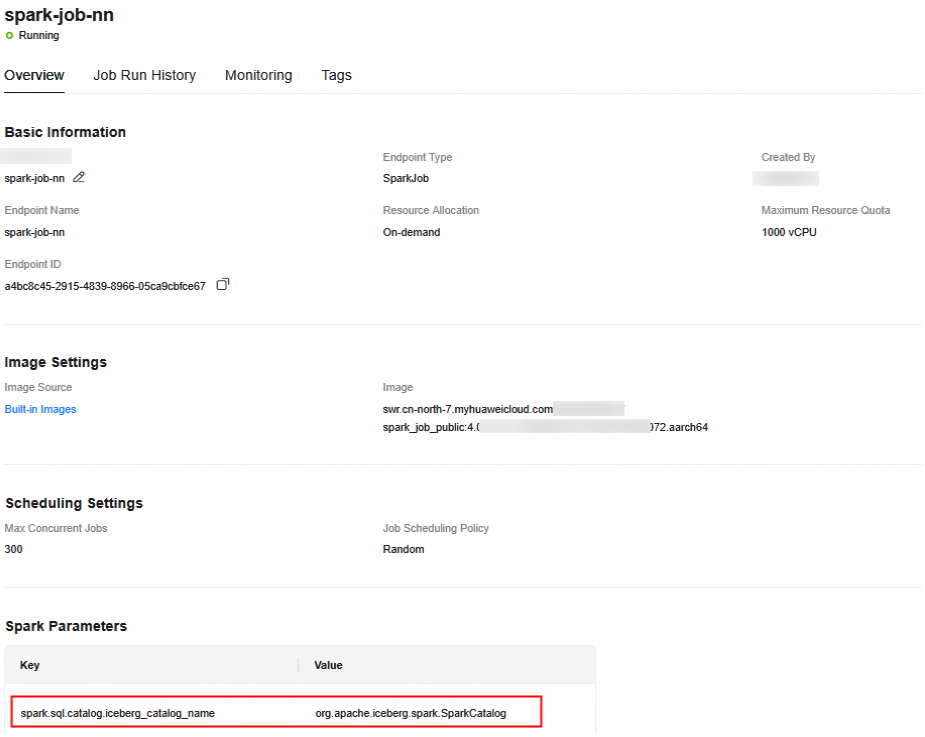
    Figure 1-1 Configuring on the endpoint creation page

  - Configuration via the endpoint modification page

 Copyright © Huawei Technologies Co., Ltd.

Figure 1-2 Configuring on the endpoint modification page



1.3.2 Iceberg Configurations and Types

Mapping between Spark types and Iceberg types

Spark and Iceberg support different type sets. Iceberg automatically performs type conversions, but not all mapping types are supported. Therefore, you need to understand the type conversions in Iceberg before designing the column types of a table.

Iceberg types corresponding to Spark types

The following table describes how Spark types are converted to Iceberg types. The conversion applies to both the creation of Iceberg tables and the writing of data to Iceberg tables through Spark.

Table 1-15 Iceberg types corresponding to Spark types

Spark	Iceberg
boolean	boolean
short	integer
byte	integer
integer	integer
long	long
float	float

 Copyright © Huawei Technologies Co., Ltd.

Spark	Iceberg
double	double
date	date
timestamp	timestamp with timezone
timestamp_ntz	timestamp without timezone
char	string
varchar	string
string	string
binary	binary
decimal	decimal
struct	struct
array	list
map	map

NOTE

This table mainly shows the type conversions at the time of table creation. In practice, a wider range of types are supported for conversions during write operations. Here are some key points during write operations:

- Iceberg numeric types (**integer**, **long**, **float**, **double**, **decimal**) support type promotion during write operations. For example, you can write Spark data of the **short**, **byte**, **integer**, and **long** types to Iceberg data of the **long** type.
- You can write Spark data of the **binary** type to Iceberg data of the **fixed** type. Note that length assertions will be performed during this process.

Spark types corresponding to Iceberg types

The following table describes how Iceberg types are converted to Spark types. The conversion applies to the reading of Iceberg tables through Spark.

Table 1-16 Spark types corresponding to Iceberg types

Iceberg	Spark
boolean	boolean
integer	integer
long	long
float	float
double	double
date	date

 Copyright © Huawei Technologies Co., Ltd.

Iceberg	Spark
time	Not supported
timestamp with timezone	timestamp
timestamp without timezone	timestamp_ntz
string	string
uuid	string
fixed	binary
binary	binary
decimal	decimal
struct	struct
list	array
map	map
nanosecond timestamp	Not supported
nanosecond timestamp with timezone	Not supported
unknown	Not supported
variant	variant **NOTE** Spark 4.0+ is supported.
geometry	Not supported
geography	Not supported

Runtime Configuration

Configuration Setting Priority

Iceberg allows configuration to be specified at different levels. The effective configuration for a read or write operation is determined by the following priority order:

1. DataSourceAPI read/write options: Use **.option(...)** in read/write operations to explicitly transfer the options.
2. Spark session configurations: Use **spark.conf.set(...)**, **spark-defaults.conf**, or **--conf** in spark-submit for global settings.
3. Table properties: Set the properties in the Iceberg table by running **ALTER TABLE SET TBLPROPERTIES**.
4. Default value:

 Copyright © Huawei Technologies Co., Ltd.

Use the next-level configurations as a backup if no higher-level configurations are defined. This allows global default values to be enabled when needed, maintaining flexibility.

Spark SQL options

Iceberg allows you to set various global behaviors using Spark SQL configuration options. These can be set using Spark, SparkSession configurations, or spark-submit parameters. For example:

```
//Disable vectorization.
val spark = SparkSession.builder()
.appName("IcebergExample")
.master("local[*]")
.config("spark.sql.catalog.my_catalog","org.apache.iceberg.spark.SparkCatalog")
.config("spark.sql.extensions","org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")
.config("spark.sql.iceberg.vectorization.enabled","false")
.getOrCreate()
```

Spark Option	Default Value	Description
spark.sql.iceberg.vectorization.enabled	Inherits from table properties. (Parquet: the default value of **read.parquet.vectorization.enabled** is **true**. ORC: The default value of **read.orc.vectorization.enabled** is **false**.)	Controls whether to enable vectorized read of data files. • If this parameter is set to **true**, Iceberg uses batch-based columnar vectorization to accelerate queries, processing 5000 (specified by **read.parquet.vectorization.batch-size** or **read.orc.vectorization.batch-size**) rows at a time. • If this parameter is set to **false**, a non-vectorized row-by-row reader is used, which offers lower performance but better compatibility.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.parquet.reader-type	ICEBERG	Sets the Parquet reader implementation. The options are as follows: • **ICEBERG** (default value): Uses the built-in Parquet reader of Iceberg. • **COMET**: Uses the Apache DataFusion CometParquet reader (which performs I/O and decompression in the JVM, but decodes at the native layer to improve performance; also converts Spark physical plans to native execution).
spark.sql.iceberg.check-nullability	true	Checks whether the nullability of the write schema matches the nullability of the table. • If this parameter is set to **true**, an exception is thrown if a field declared as NOT NULL in the written data contains a null value. • If this parameter is set to **false**, the check is skipped and null values can be written into the NOT NULL fields (which may cause problems in subsequent queries).

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.check-ordering	true	Checks whether the column order of the write schema matches the column order of the table schema. ● If this parameter is set to **true**, an exception is thrown if the column order of the written data is inconsistent with that defined in the table. ● If this parameter is set to **false**, the check is skipped. Columns can be matched by name instead of position.
spark.sql.iceberg.planning.preserve-data-grouping	false	● If this parameter is set to **true**, scanning tasks of the same partition are placed in the same read shard for storage partitioned joins, which optimizes the locality of data in the partition. ● If this parameter is set to **false**, scanning tasks are allocated based on the default policy. Data in the same partition may not be in the same partition.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.aggregate-push-down.enabled	true	Controls whether to push down aggregate functions (**MAX**, **MIN**, and **COUNT**). • If this parameter is set to **true**, Iceberg attempts to push aggregation calculations into the data file read layer and uses file-level statistics (such as min/max values) to skip unnecessary data reads. • If this parameter is set to **false**, all aggregation calculations are completed at the Spark layer.
spark.sql.iceberg.distribution-mode	Determined based on your environment: • If the table has a sorting order, the value is **range**. • If the table has partitions, the value is **hash**. • In other cases, the value is **none**.	Controls the data distribution policy during write operations. The options are as follows: • **none**: Data is not redistributed. • **hash**: Hash distribution is performed based on the partition key. This is applicable to scenarios where data is evenly distributed in partitions. • **range**: Range distribution is performed based on the partition key or sorting key. This is applicable to data skew scenarios. For details, see **1.3.5 Writing Data to an Iceberg Table**.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.wap.id	null	Specifies the write-audit-release (WAP) snapshot staging ID. After this parameter is set, a new snapshot is created in the staged state and associated with this ID. You need to manually publish the snapshot. If this parameter is set to **null**, the snapshot is directly committed. Before setting this parameter, set the table property **write.wap.enabled** to **true**.
spark.wap.branch	null	Specifies the WAP branch name. After this parameter is set, the snapshot is committed to the specified branch instead of the main branch to isolate the development or test environment. If this parameter is set to **null**, the snapshot is committed to the current branch. Before setting this parameter, set the table property **write.wap.enabled** to **true**.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.compression-codec	Inherits from table properties. (Parquet: The default value of **write.parquet.compression-codec** is **zstd**. Avro: The default value of **write.avro.compression-codec** is gzip. ORC: The default value of **write.orc.compression-codec** is **zlib**.)	Specifies the compression codec used for writing data. Different file formats support different codecs: • Parquet files support the **uncompressed**, **snappy**, **gzip**, **lz4**, **zstd**, **brotli**, **lzop**, **bz2**, and **zstd-no-seek** codecs. • Avro files support the **null** (no compression), **deflate**, **snappy**, **bzip2**, **xz**, **lz4**, and **zstd** codecs. • ORC files support the **none**, **snappy**, **zlib**, **lz4**, **lzo** and **zstd** codecs
spark.sql.iceberg.compression-level	Inherits from table properties (Parquet: the default value of **write.parquet.compression-level** is **null**. Avro: the default value of **write.avro.compression-level** is **null**.)	Specifies the compression level of Parquet/Avro files. Generally, the value is an integer ranging from 1 to 9. A larger value indicates a higher compression ratio but a slower speed. The default value **null** indicates that the default level of the codec is used. This parameter is valid only for codecs that support level configurations, such as **gzip**, **zstd**, and **deflate**.
spark.sql.iceberg.compression-strategy	Inherits from the table property **write.orc.compression-strategy** (default value: **speed**).	Specifies the ORC compression policy. The options are as follows: • **speed** (default value): The compression speed is prioritized to reduce CPU usage. • **compression**: The compression ratio is prioritized to reduce storage usage.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.data-planning-mode	Inherits from the table property **read.data-planning-mode** (default value: **AUTO**).	Specifies the scan planning mode of data files. The options are as follows: • **AUTO** (default value): The system automatically selects **LOCAL** or **DISTRIBUTED** based on the number of manifest files. • **LOCAL**: All files are planned to be executed in the Driver process. • **DISTRIBUTED**: Files are planned and distributed to Spark Executors for execution. (This requires the value of **spark.driver.maxResultSize** be no less than 256 MB. Otherwise, the **LOCAL** mode is used.)
spark.sql.iceberg.delete-planning-mode	Inherits from the table property **read.delete-planning-mode** (default value: **AUTO**).	Specifies the scan planning mode of the deletion files. The options are as follows: • **AUTO** (default value): The system automatically selects a value. • **LOCAL**: This planning is performed on the Driver. • **DISTRIBUTED**: The distributed planning is used. (This requires the value of **spark.driver.maxResultSize** be no less than 256 MB. Otherwise, the **LOCAL** mode is used.)

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.advisory-partition-size	Inherits from the table property **write.target-file-size-bytes** (default value: **536870912**, that is, 512 MB).	Specifies the recommended partition size (in bytes) for writing data to a table when Spark Adaptive Query Execution (AQE) is enabled. It is used to adjusts the size of the output files, which affects the amount of data written by each task.
spark.sql.iceberg.locality.enabled	false	Controls whether to report the localization information of Spark tasks on the executor. ● If this parameter is set to **true**, Iceberg reports the localized location of data blocks to Spark so that Spark can schedule compute tasks to the nodes where the data is stored, thereby reducing network transmission. This parameter is valid for storage systems that support data locality, such as HDFS, but is invalid for object storage systems like OBS/S3. ● If this parameter is set to **false**, Spark randomly allocates tasks to executors.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.executor-cache.enabled	true	Controls whether to enable the cache on the executor. • If this parameter is set to **true**, Iceberg caches DeleteFiles on the executor to avoid repeated reads during row-level operations. • If this parameter is set to **false**, the file is read from the storage for each operation.
spark.sql.iceberg.executor-cache.timeout	10 (10 minutes)	Specifies the timeout interval for a cache entry, in minutes. Cache entries that are not accessed within this period will be cleared.
spark.sql.iceberg.executor-cache.max-entry-size	67108864 (64 MB)	Specifies the maximum size of each cache entry, in bytes. Files larger than this size will not be cached.
spark.sql.iceberg.executor-cache.max-total-size	134217728 (128 MB)	Specifies the maximum total size of the executor cache, in bytes. When the total cache size exceeds this value, old entries are evicted based on the least recently used (LRU) policy.
spark.sql.iceberg.executor-cache.locality.enabled	false	• Controls whether to enable the locality-aware executor cache usage. If this parameter is set to **true**, cache hits are prioritized on the executor where data is localized, reducing cross-node cache lookups. • If this parameter is set to **false**, data locality is not considered during cache lookup.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
spark.sql.iceberg.merge-schema	false	Controls whether to modify the table schema to match the schema of the written data. ● If this parameter is set to **true**, Iceberg automatically adds new columns that do not exist in the table to the table schema. Only missing columns are added. Existing columns are not deleted or modified. ● If this parameter is set to **false**, an exception is thrown if the written data contains a column that does not exist in the table schema.
spark.sql.iceberg.report-column-stats	true	Controls whether to report the Puffin table statistics to the cost-based optimizer (CBO) of Spark. ● If this parameter is set to **true**, Iceberg transfers column statistics (such as NDV and null value count) to Spark CBO to optimize the query plan. ● If this parameter is set to **false**, Iceberg does not report column statistics to Spark CBO. This parameter takes effect only when Spark CBO is enabled (**spark.sql.cbo.enabled** is set to **true**).

Read options

Spark read options are transferred when DataFrame Reader is configured, as shown below:

```
//Time travel
spark.read
```

 Copyright © Huawei Technologies Co., Ltd.

```
.option("snapshot-id",10963874102873L)
.table("catalog.db.table")
```

Spark Option	Default Value	Description
snapshot-id	null	Specifies the ID (long type) of the table snapshot to be read. It is used for the TimeTravel function to read data from a specified historical version. This parameter is mutually exclusive with **as-of-timestamp**. If this parameter is set to **null**, the latest snapshot is read.
as-of-timestamp	null	Specifies a timestamp in milliseconds (long type). If this parameter is set, the latest snapshot generated before this timestamp is used. It is used for time-based historical data query. This parameter is mutually exclusive with **snapshot-id**. If both parameters are set, **snapshot-id** takes precedence. If this parameter is set to **null**, the latest snapshot is read.
split-size	Inherits from the table properties. (The default value of **read.split.target-size** is **134217728**, that is, 128 MB. The default value of **read.split.metadata-target-size** is **33554432**, that is, 32 MB.)	Overrides the table's **read.split.target-size** (target size of data file splitting) and **read.split.metadata-target-size** (target size of metadata file splitting). It is used to control the amount of data processed by each Spark task. A smaller value indicates more tasks to be generated.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
lookback	Inherits from the table property **read.split.planning-lookback** (default value: **10**).	Overrides the table's **read.split.planning-lookback**. It specifies the number of files that are backtracked when splits are planned, which is used to merge small files into the same split. A larger value indicates more files to be merged but a longer planning time.
file-open-cost	Inherits from the table property **read.split.open-file-cost**. The default value is **4194304**, that is, 4 MB.	Overrides the table's **read.split.open-file-cost**. It indicates the estimated cost (in bytes) of opening a file, which is used for cost optimization during split planning.
vectorization-enabled	Inherits from table properties. (Parquet: the default value of **read.parquet.vectorization.enabled** is **true**. ORC: The default value of **read.orc.vectorization.enabled** is **false**.)	Overrides the vectorized read switch of this table. If this parameter is set to **true**, Parquet/ORC vectorized read is enabled, and columnar batch processing is used to improve query performance. If this parameter is set to **false**, a non-vectorized row-by-row reader is used, which offers lower performance but better compatibility.
batch-size	Inherits from table properties. (Parquet: The default value of **read.parquet.vectorization.batch-size** is **5000**. ORC: The default value of **read.orc.vectorization.batch-size** is **5000**.)	Overrides the vectorized batch size of this table, that is, the number of rows processed in each batch. Larger batches (for example, 10,000) increase throughput but also increase memory consumption. Smaller batches (for example, 1,000) reduce memory usage but decrease throughput.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
stream-from-timestamp	null	Specifies a timestamp (long type) in milliseconds as the start point for streaming. If the timestamp is earlier than the oldest known ancestor snapshot, the oldest snapshot is used. If this parameter is set to **null**, streaming reads start from the latest snapshot.
streaming-max-files-per-micro-batch	Integer.MAX_VALUE (2147483647)	Specifies the maximum number of files per micro-batch. It is used to control the number of files processed in each batch of the structured streaming. Setting a smaller value can prevent delays caused by large batches.
streaming-max-rows-per-micro-batch	Integer.MAX_VALUE (2147483647)	Specifies the maximum number of rows per micro-batch (soft limit).

NOTE

When the **streaming-max-rows-per-micro-batch** option defines a soft maximum value, a batch will always contain all rows in the next unprocessed data file. However, if including additional files would exceed the soft maximum limit, those files will not be included.

Write options

Spark write options are transferred when DataFrameWriterV2 is configured, as shown below:

```
//Use Avro instead of Parquet for writing.
df.writeTo("catalog.db.table")
.option("write-format","avro")
.option("snapshot-property.key","value")
.append()
```

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
write-format	Inherits from the table property **write.format.default** (default value: **parquet**).	Specifies the file format used for the write operation. The options are as follows: • **parquet** (default) • **avro** • **orc**
target-file-size-bytes	Inherits from the table property **write.target-file-size-bytes** (default value: **536870912**, that is, 512 MB).	Overrides the table's **write.target-file-size-bytes**. It is used to control the target size of each output data file. A smaller value results in more small files (which may increase the scanning overhead during queries), while a larger value reduces the number of files but increases the size of a single file (which may increase memory consumption).
check-nullability	true	Controls whether to perform the nullability check on a field. If this parameter is set to **true**, an exception is thrown if a field declared as NOT NULL in the written data contains a null value. If this parameter is set to **false**, this check is skipped and null values can be written to the **NOT NULL** field (which may cause subsequent query exceptions).

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
snapshot-property.*custom-key*	null	Adds an entry with a custom key and corresponding value to the snapshot summary. For example, **snapshot-property.user=alice** will add **{"user":"alice"}** to the snapshot metadata. This parameter is used for audit tracking or change marking. (The DSv2 API requires the prefix **snapshot-property.**)
fanout-enabled	false	Overrides the table's **write.spark.fanout.enabled**. ● If this parameter is set to **true**, data is written by partition to prevent a single task from writing to too many partitions (one write task per partition). This is suitable for scenarios where data needs to be written to a large number of partitions, but it increases the shuffle overhead and memory consumption. ● If this parameter is set to **false**, a single task is written to multiple partitions, which may cause memory pressure when data is written to a large number of partitions.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
check-ordering	true	Checks whether the input schema is the same as the table schema. • If this parameter is set to **true**, an exception is thrown if the column order of the written data is inconsistent with that defined in the table. • If this parameter is set to **false**, the check is skipped. Columns can be matched by name instead of position.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
isolation-level	null	Specifies the isolation level required for DataFrame overwrite operations. The options are as follows: • **null** (default): No check is performed. This is suitable for idempotent writes. • **serializable**: Checks concurrent insert or delete operations in the target partition to ensure full isolation. • **snapshot**: Checks only concurrent deletions in the target partition and allows concurrent insertions. Default values for table properties: • The default value of **write.delete.isolation-level** is **serializable**. • The default value of **write.update.isolation-level** is **serializable**. • The default value of **write.merge.isolation-level** is **serializable**.
validate-from-snapshot-id	null	If the isolation level is set, this parameter specifies the baseline snapshot ID (of the long type) for checking concurrent write conflicts in the table. It should be the ID of any snapshot before the table is read. You can obtain the value from the Table API or Snapshots table. If this parameter is set to **null**, the oldest known snapshot of the table is used.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
compression-codec	Inherits from table properties. (Parquet: The default value of **write.parquet.compression-codec** is **zstd**. Avro: The default value of **write.avro.compression-codec** is **gzip**. ORC: The default value of **write.orc.compression-codec** is **zlib**.)	Overrides the compression codec used for writing data to this table. • For Parquet, the value can be **uncompressed**, **snappy**, **gzip**, **lz4**, **zstd**, **brotli**, **lzop**, **bz2** or **zstd-no-seek**. • For Avro, the value can be **null** (no compression), **deflate**, **snappy**, **bzip2**, **xz**, **lz4**, or **zstd**. • For ORC, the value can be **none**, **snappy**, **zlib**, **lz4**, **lzo** or **zstd**.
compression-level	Inherits from table properties (Parquet: the default value of **write.parquet.compression-level** is **null**. Avro: the default value of **write.avro.compression-level** is **null**.)	Overrides the compression level used for writing data in Parquet and Avro formats to this table. The value is an integer ranging from 1 to 9. A larger value indicates a higher compression ratio but a slower compression speed. The default value **null** indicates that the default level of the codec is used. This parameter is valid only for codecs that support level configurations, such as **gzip**, **zstd**, and **deflate**.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
compression-strategy	Inherits from the table property **write.orc.compression-strategy** (default value: **speed**).	Overrides the compression strategy used for writing data in ORC format to this table. The options are as follows: • **speed** (default value): The compression speed is prioritized to reduce CPU usage. • **compression**: The compression ratio is prioritized to reduce storage usage.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
distribution-mode	Determined based on your environment. If the table has a sorting order, the value is **range**. If the table has partitions, the value is **hash**. In other cases, the value is none.	Overrides the distribution mode used for writing data to this table. The options are as follows: • **none**: Data is not redistributed. • **hash**: Data is distributed by partition key in hash mode. This is applicable to scenarios where data is evenly distributed in partitions. • **range**: Range distribution is performed based on the partition key or sorting key. This is applicable to data skew scenarios. Default values for table properties: The default values of **write.delete.distribution-mode**, **write.update.distribution-mode**, and **write.merge.distribution-mode** are hash. For details, see **1.3.5 Writing Data to an Iceberg Table**.

 Copyright © Huawei Technologies Co., Ltd.

Spark Option	Default Value	Description
delete-granularity	file	Overrides the write and delete granularity used for this table. The options are as follows: • **file** (default value): Data is deleted by file. A **position delete** file is generated to point to a specific data file line. • **partition**: Data is deleted by partition. A **partition delete** file is generated to mark the entire partition as deleted. (The default value of the table property **write.delete.granularity** is **partition**.) This parameter affects the efficiency of the DELETE operations. The **file** level is more accurate but generates more small files. The **partition** level is more efficient but deletes the entire partition.

1.3.3 Iceberg on Spark DDL Syntax Description

CREATE TABLE

Spark can create tables in any Iceberg catalog. Iceberg will convert the column type in Spark to corresponding Iceberg type. The command is as follows:

```
CREATE DATABASE IF NOT EXISTS spark_catalog.db location 'obs://yourbucket/path/to/db';
CREATE TABLE spark_catalog.db.sample (id bigint NOT NULL COMMENT 'unique id', data string) USING iceberg;
```

Table creation commands support the full range of Spark **create** clauses, including:

- **PARTITIONED BY**: Configures partitioning.
- **LOCATION '*(fully-qualified-uri)*'**: Sets the table location.
- **COMMENT '*table documentation*'**: Sets the table description.
- **TBLPROPERTIES ('key'='value', ...)**: Sets table configuration. For details about common table properties, see **1.1.2 Iceberg Table Properties**.

 Copyright © Huawei Technologies Co., Ltd.

PARTITIONED BY

To create a partitioned table, use the following command:

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string) USING iceberg
PARTITIONED BY (category);
```

The **PARTITIONED BY** clause supports transform expressions to create hidden partitions.

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp) USING iceberg
PARTITIONED BY (bucket(16, id), days(ts), category);
```

Supported transformations are:

- **year(ts)**: partition by year
- **month(ts)**: partition by month
- **day(ts) or date(ts)**: equivalent to dateint partitioning
- **hour(ts) or date_hour(ts)**: equivalent to dateint and hour partitioning
- **bucket(*N*, *col*)**: partition by hashed value mod *N* buckets
- **truncate(*L*, *col*)**: partition by value truncated to *L*
  - Strings are truncated to the given length (*L*).
  - Integers and longs truncate to bins: truncate(10, i) produces partitions 0, 10, 20, 30, ...

CTAS Syntax

Iceberg supports **CTAS** (CREATE TABLE ... AS SELECT) as an atomic operation when using a SparkCatalog. **CTAS** is supported, but is not atomic when using SparkSessionCatalog.

```
CREATE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
```

New tables generated by **CTAS** do not inherit partition specifications or table properties from the source table specified in the **SELECT** statement. To configure partition specifications or table properties for the new table, you can explicitly declare partition specifications using the **PARTITIONED BY** clause and define table properties using the **TBLPROPERTIES** clause in the CTAS statement. The command is as follows:

```
CREATE TABLE spark_catalog.db.sample USING iceberg PARTITIONED BY (part) TBLPROPERTIES
('key'='value') AS SELECT ...;
```

RTAS Syntax

Iceberg supports **RTAS** (REPLACE TABLE ... AS SELECT) as an atomic operation when using a SparkCatalog. **RTAS** is supported, but is not atomic when using SparkSessionCatalog.

Atomic table replacement creates a new snapshot with the results of the SELECT query, but keeps table history. Methods are as follows:

- Method 1:
  ```
  REPLACE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
  ```
- Method 2:
  ```
  REPLACE TABLE spark_catalog.db.sample USING iceberg PARTITIONED BY (part) TBLPROPERTIES
  ('key'='value') AS SELECT ...;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- Method 3:
  ```
  CREATE OR REPLACE TABLE spark_catalog.db.sample USING iceberg AS SELECT ...;
  ```

DROP TABLE

The **DROP TABLE** behavior in Iceberg changed in 0.14. Detailed changes are as follows:

- Prior to 0.14, running **DROP TABLE** would remove the table metadata from the catalog and delete the table contents as well.
- From 0.14 onwards, **DROP TABLE** would only remove the table metadata from the catalog. In order to delete the table contents, **DROP TABLE PURGE** should be used.
  - **DROP TABLE** (removes metadata only)
    ```
    DROP TABLE spark_catalog.db.sample;
    ```
  - **DROP TABLE PURGE** (removes metadata and deletes data)
    ```
    DROP TABLE spark_catalog.db.sample PURGE;
    ```

ALTER TABLE

NOTE

Renaming a table (**ALTER TABLE ... RENAME TO**) is not supported.

In Spark, Iceberg supports the following **ALTER TABLE** operations:

- Setting or removing table properties (**ALTER TABLE ... SET**/**UNSET TBLPROPERTIES**)
  - Setting table properties
    ```
    ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ('read.split.target-size'='268435456');
    ```
    SET TBLPROPERTIES can also be used to set the table comment (description).
    ```
    ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ( 'comment' = 'A table comment.');
    ```
  - Removing table properties (**UNSET**):
    ```
    ALTER TABLE spark_catalog.db.sample UNSET TBLPROPERTIES ('read.split.target-size');
    ```
- Adding, deleting, and renaming columns
  - Adding columns
    - To add a column to Iceberg, use the **ADD COLUMNS** clause with **ALTER TABLE**. Altering a map **key** column by adding columns is not allowed. Only map values can be updated. Here is an example:
      - Add struct-type **point** columns to the **spark_catalog.db.sample** table.
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN point struct<x: double, y: double>;
        ```
      - Add double-type **point.z** columns to the **spark_catalog.db.sample** table.
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN point.z double;
        ```
      - Add array-type **points** columns to the **spark_catalog.db.sample** table.
        ```
        ALTER TABLE spark_catalog.db.sample ADD COLUMN points array<struct<x: double, y: double>>;
        ```
      - Add double-type **points.element.z** columns to the **spark_catalog.db.sample** table.

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER TABLE spark_catalog.db.sample ADD COLUMN points.element.z double;
```

    - Add map-type **points** columns to the **spark_catalog.db.sample** table.
      ```
      ALTER TABLE spark_catalog.db.sample ADD COLUMN points map<struct<x: int>, struct<a: int>>;
      ```

  - You can also add columns in any position by adding **FIRST** or **AFTER** clauses. Examples are as follows:
    - Example 1:
      ```
      ALTER TABLE spark_catalog.db.sample ADD COLUMN new_column bigint AFTER {Existing field names in the table (for example, id or data)};
      ```
    - Example 2:
      ```
      ALTER TABLE spark_catalog.db.sample ADD COLUMN new_column bigint FIRST;
      ```

- Renaming columns (**ALTER TABLE ... RENAME COLUMN**)

  Iceberg allows any field to be renamed. To rename a field, use **RENAME COLUMN**.

  ```
  ALTER TABLE spark_catalog.db.sample RENAME COLUMN data TO payload;
  ```

- To drop columns, use **ALTER TABLE** ... **DROP COLUMN**.
  ```
  ALTER TABLE spark_catalog.db.sample DROP COLUMN id;
  ```

- Adding, deleting, and renaming nested fields

  To add or remove columns from a struct, use **ADD COLUMN** or **DROP COLUMN** with a nested column name.

- Reordering top-level columns and nested struct fields.

  Iceberg allows reordering top-level columns or columns in a struct using **FIRST** and **AFTER** clauses.

  ```
  ALTER TABLE spark_catalog.db.sample ALTER COLUMN id FIRST;
  ```

- Widening the type of int, float, and decimal fields (ALTER TABLE ... ALTER COLUMN)

  ALTER COLUMN is used to widen types, make a field optional, set comments, and reorder fields.

  Iceberg allows updating column types if the update is safe. Safe updates are:

  - **int** to **bigint**
  - **float** to **double**
  - **decimal(P,S)** to **decimal(P2,S)** when P2 > *P* (Precision can be increased, but scale must remain unchanged.)

  For example, run the following command to change the data type of the **measurement** column to **double**:

  ```
  ALTER TABLE spark_catalog.db.sample ALTER COLUMN measurement TYPE double;
  ```

  Column comments can also be updated using **ALTER COLUMN**:
  ```
  ALTER TABLE spark_catalog.db.sample ALTER COLUMN {Double-type field} COMMENT 'unit is kilobytes per second';
  ```

- Making required columns optional

  Nullability for a non-nullable column can be changed using **DROP NOT NULL**.

  ```
  ALTER TABLE spark_catalog.db.sample ALTER COLUMN id DROP NOT NULL;
  ```

- SQL extensions can be used to add support for partition evolution and setting a table's write order.

 Copyright © Huawei Technologies Co., Ltd.

DDL Syntax of Iceberg SQL Extensions

These commands are available in Spark when you use Iceberg SQL extensions.

```
ALTER TABLE ... ADD PARTITION FIELD;
```

- ADD PARTITION FIELD

  Iceberg supports adding new partition fields to a spec using **ADD PARTITION FIELD**.

  ```
  ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD id;
  ```

  Partition transforms are also supported.

  – Example 1:
  ```
  ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD bucket(16, id);
  ```
  – Example 2:
  ```
  ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD truncate(4, data);
  ```
  – Example 3:
  ```
  ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD year(ts);
  ```
  – Example 4:
  ```
  ALTER TABLE spark_catalog.db.sample ADD PARTITION FIELD day(ts);
  ```

  Adding a partition field is a metadata operation and does not change any of the existing table data. New data will be written with the new partitioning, but existing data will remain in the old partition layout. Old data files will have null values for the new partition fields in metadata tables.

  Dynamic partition overwrite behavior will change when the table's partitioning changes because dynamic overwrite replaces partitions implicitly. To overwrite explicitly, use the new DataFrameWriterV2 API.

- DROP PARTITION FIELD

  Partition fields can be removed using **DROP PARTITION FIELD**. Note that although the partition is removed, the column will still exist in the table schema.

  Dropping a partition field is a metadata operation and does not change any of the existing table data. New data will be written with the new partitioning, but existing data will remain in the old partition layout.

  – Create an Iceberg table.
  ```
  CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp, shard int) USING iceberg PARTITIONED BY (category);
  ```
  – Remove partition fields.

    ▪ Example 1:
    ```
    ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD category;
    ```
    ▪ Example 2:
    ```
    ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD bucket(16, id);
    ```
    ▪ Example 3:
    ```
    ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD truncate(4, data);
    ```
    ▪ Example 4:
    ```
    ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD year(ts);
    ```
    ▪ Example 5:
    ```
    ALTER TABLE spark_catalog.db.sample DROP PARTITION FIELD shard;
    ```

- REPLACE PARTITION FIELD

 Copyright © Huawei Technologies Co., Ltd.

A partition field can be replaced by a new partition field in a single metadata update by using **REPLACE PARTITION FIELD**.

- Example 1:
  ```
  ALTER TABLE spark_catalog.db.sample REPLACE PARTITION FIELD ts_day WITH day(ts);
  ```
- Example 2:
  ```
  ALTER TABLE spark_catalog.db.sample REPLACE PARTITION FIELD ts_day WITH day(ts) AS day_of_ts;
  ```

- WRITE ORDERED BY

  Iceberg tables can be configured with a sorting order that is used to automatically sort data that is written to the table in some engines. For example, MERGE INTO in Spark will use the table ordering.

  To set the write order for a table, use **WRITE ORDERED BY**. Examples are as follows:

  - Example 1:
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category, id;
    ```
  - Example 2:
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category ASC, id DESC;
    ```
  - Example 3:
    ```
    ALTER TABLE spark_catalog.db.sample WRITE ORDERED BY category ASC NULLS LAST, id DESC NULLS FIRST;
    ```

  WRITE ORDERED BY sets a global ordering where rows are ordered across tasks, like using **ORDER BY** in an **INSERT** command.
  ```
  INSERT INTO spark_catalog.db.sample SELECT id, data, category, ts FROM {Another table} ORDER BY ts, category;
  ```

  To order within each task, not across tasks, use **LOCALLY ORDERED BY**.

  ```
  ALTER TABLE spark_catalog.db.sample WRITE LOCALLY ORDERED BY category, id;
  ```

  To cancel the sorting order of the table, use **UNORDERED**.

  ```
  ALTER TABLE spark_catalog.db.sample WRITE UNORDERED;
  ```

- WRITE DISTRIBUTED BY PARTITION

  WRITE DISTRIBUTED BY PARTITION will request that each partition is handled by one writer, the default implementation is hash distribution.

  ```
  ALTER TABLE spark_catalog.db.sample WRITE DISTRIBUTED BY PARTITION;
  ```

  DISTRIBUTED BY PARTITION and **LOCALLY ORDERED BY** may be used together, to distribute data by partition and locally order rows within each task.

  ```
  ALTER TABLE spark_catalog.db.sample WRITE DISTRIBUTED BY PARTITION LOCALLY ORDERED BY category, id;
  ```

- SET IDENTIFIER FIELDS

  Iceberg supports setting identifier fields to a spec using **SET IDENTIFIER FIELDS**. A Spark table can support Spark SQL **upsert** operations if the table has identifier fields.

  - Example 1: Ensure that all fields are set.
    ```
    ALTER TABLE spark_catalog.db.sample SET IDENTIFIER FIELDS id;
    ```
  - Example 2:
    ```
    ALTER TABLE spark_catalog.db.sample SET IDENTIFIER FIELDS id, data;
    ```

  Identifier fields must be **NOT NULL** columns when they are created or added. The later **ALTER TABLE ... SET IDENTIFIER FIELDS** statement will overwrite the previous setting.

- DROP IDENTIFIER FIELDS

 Copyright © Huawei Technologies Co., Ltd.

When removing identifier fields using **DROP IDENTIFIER FIELDS**, although the identifier is removed, the column will still exist in the table schema.

- Example 1: Ensure that all fields are set.
  ```
  ALTER TABLE spark_catalog.db.sample DROP IDENTIFIER FIELDS id;
  ```
- Example 2:
  ```
  ALTER TABLE spark_catalog.db.sample DROP IDENTIFIER FIELDS id, data;
  ```

Branching and Tagging DDL Syntax

- CREATE BRANCH

  Branches can be created via the **CREATE BRANCH** statement with the following options:

  - **IF NOT EXISTS**: Do not fail if the branch already exists.
  - **CREATE OR REPLACE**: Update the branch if it already exists.
  - Create a branch at a specific snapshot.
  - Create a branch with a specified retention period.

    - Example 1:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE BRANCH `audit-branch`;
      ```
    - Example 2:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE BRANCH IF NOT EXISTS `audit-branch`;
      ```
    - Example 3:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE OR REPLACE BRANCH `audit-branch`;
      ```

- CREATE TAG

  Tags can be created via the **CREATE TAG** statement with the following options:

  - **IF NOT EXISTS**: Do not fail if the tag already exists.
  - **CREATE OR REPLACE**: Update the tag if it already exists
  - Create a tag at a specific snapshot.
  - Create a tag with a specified retention period.

    - Example 1:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE TAG `historical-tag`;
      ```
    - Example 2:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE TAG IF NOT EXISTS `historical-tag`;
      ```
    - Example 3:
      ```
      ALTER TABLE spark_catalog.db.sample CREATE OR REPLACE TAG `historical-tag`;
      ```

- REPLACE BRANCH

  The snapshot which a branch references can be updated via the **REPLACE BRANCH** statement. Retention can also be updated in this statement.

  ```
  ALTER TABLE spark_catalog.db.sample REPLACE BRANCH `audit-branch` AS OF VERSION {current-snapshot-id} RETAIN 60 DAYS;
  ```

- REPLACE TAG

  The snapshot which a tag references can be updated via the **REPLACE TAG** statement. Retention can also be updated in this statement.

  ```
  ALTER TABLE spark_catalog.db.sample REPLACE TAG `historical-tag` AS OF {current-snapshot-id} RETAIN 60 DAYS;
  ```

 Copyright © Huawei Technologies Co., Ltd.

- DROP BRANCH

  Branches can be removed via the **DROP BRANCH** statement.

  ```
  ALTER TABLE spark_catalog.db.sample DROP BRANCH `audit-branch`;
  ```

- DROP TAG

  Tags can be removed via the **DROP TAG** statement.

  ```
  ALTER TABLE spark_catalog.db.sample DROP TAG `historical-tag`;
  ```

DDL Syntax of Iceberg Views

Iceberg views are designed to be engine-agnostic, allowing the same view definition to yield consistent behavior and results across different query engines. This enhances flexibility and compatibility in data querying.

Iceberg views are a common representation of SQL views and aim to be interpreted across multiple query engines. The following describes how to create and manage views in Spark.

- Creating a view

  Create a simple view without any comments or properties.

  ```
  CREATE VIEW <viewName> AS SELECT * FROM <tableName>;
  ```

  Using **IF NOT EXISTS** prevents the SQL statement from failing in case the view already exists.

  ```
  CREATE VIEW IF NOT EXISTS <viewName> AS SELECT * FROM <tableName>;
  ```

  Create a view with a comment, including aliased and commented columns that are different from the source table.

  ```
  CREATE VIEW <viewName> (ID COMMENT 'Unique ID', ZIP COMMENT 'Zipcode')  COMMENT 'View Comment' AS SELECT id, zip FROM <tableName>;
  ```

- Creating a view with properties

  Create a view with properties using **TBLPROPERTIES**.

  ```
  CREATE VIEW <viewName> TBLPROPERTIES ('key1' = 'val1', 'key2' = 'val2') AS SELECT * FROM <tableName>;
  ```

  Display view properties.

  ```
  SHOW TBLPROPERTIES <viewName>;
  ```

- Dropping a view
  - Drop an existing view.

    ```
    DROP VIEW <viewName>;
    ```

  - Using **IF EXISTS** prevents the SQL statement from failing if the view does not exist.

    ```
    DROP VIEW IF EXISTS <viewName>;
    ```

- Replacing a view

  Update a view's schema, its properties, or the underlying SQL statement using **CREATE OR REPLACE**:

  ```
  CREATE OR REPLACE VIEW <viewName> (updated_id COMMENT 'updated ID') TBLPROPERTIES ('key1' = 'new_val1') AS SELECT id FROM <tableName>;
  ```

- Setting and removing view properties
  - Set the properties of an existing view using **ALTER VIEW** ... **SET TBLPROPERTIES**.

    ```
    ALTER VIEW <viewName> SET TBLPROPERTIES ('key1' = 'val1', 'key2' = 'val2');
    ```

  - Remove the properties from an existing view using **ALTER VIEW** ... **UNSET TBLPROPERTIES**.

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER VIEW <viewName> UNSET TBLPROPERTIES ('key1', 'key2');
```

- Showing available views

  List all views in the currently set namespace (set using **USE** *<Namespace>*).

  ```
  SHOW VIEWS;
  ```

  List all available views in the defined catalog and/or namespace using one of the below variations.

  - List all available views in the defined catalog.
    ```
    SHOW VIEWS IN <catalog>;
    ```
  - List all available views in the defined namespace.
    ```
    SHOW VIEWS IN <namespace>;
    ```
  - List all available views in the defined catalog and namespace.
    ```
    SHOW VIEWS IN <catalog>.<namespace>;
    ```
- Showing the CREATE statement of a view
  ```
  SHOW CREATE TABLE <viewName>;
  ```
- Displaying view details using **DESCRIBE**
  ```
  DESCRIBE [EXTENDED] <viewName>;
  ```

1.3.4 Querying Data in the Iceberg Table

To use Iceberg in Spark, first configure Spark catalogs. For details about how to configure Spark catalogs, see **1.3.2 Iceberg Configurations and Types**. Iceberg uses Apache Spark's DataSourceV2 API for data source and catalog implementations

Querying Data in a Table

In Spark, tables use identifiers that include a catalog name. Querying table data includes:

Create a table.

```
CREATE TABLE spark_catalog.db.table (id bigint, data string, category string, ts timestamp, shard int) USING iceberg PARTITIONED BY (category);
```

- Query table data.
  ```
  SELECT * FROM spark_catalog.db.table;
  ```
- Metadata tables, like the history and snapshots, can use the Iceberg table name as a namespace. For example:
  ```
  SELECT * FROM spark_catalog.db.table.files;
  ```

Loading Tables as DataFrames

To load a table as a DataFrame, use **spark.table**.

```
//spark-shell
val df = spark.table("spark_catalog.db.table")
```

Using Catalogs with DataFrameReader

Paths and table names can be loaded with Spark's DataFrameReader API. How tables are loaded depends on how the identifier is specified. When **spark.read.format("iceberg").load(*table*)** or **spark.table(*table*)** is used, the table variable can take a number of forms as listed below.

- **tablename**: loads from current namespace of the current catalog, that is **currentCatalog.currentNamespace.tablename**.

 Copyright © Huawei Technologies Co., Ltd.

- **catalog.tablename**: loads tables from the specified catalog.
- **namespace.tablename**: loads tables in the specified namespace from the current catalog.
- **catalog.namespace.tablename**: loads tables in the specified namespace from the specified catalog.
- **namespace1.namespace2.tablename**: loads tables in a multi-level namespace from the current catalog.

The above list is in order of priority. For example, a matching catalog will take priority over any namespace resolution.

Time Travel

- SQL commands

  Spark supports time travel using the **TIMESTAMP AS OF** or **VERSION AS OF** clause in SQL queries. The **VERSION AS OF** clause can contain a long snapshot ID or a string branch or tag name.

  – Time travel to October 26, 1986 at 01:21:00
  ```
  SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF '1986-10-26 01:21:00';
  ```
  – Time travel to the snapshot with ID **10963874102873**
  ```
  SELECT * FROM spark_catalog.db.table VERSION AS OF 10963874102873;
  ```
  Command for querying the ID of snapshot **current-snapshot-id**
  ```
  DESCRIBE EXTENDED spark_catalog.db.table;
  ```
  – Time travel to the head snapshot of **audit-branch**.
  ```
  SELECT * FROM spark_catalog.db.table VERSION AS OF 'audit-branch';
  ```
  – Time travel to the snapshot referenced by the tag **historical-snapshot**
  ```
  SELECT * FROM spark_catalog.db.table VERSION AS OF 'historical-snapshot';
  ```

  In addition, **FOR SYSTEM_TIME AS OF** and **FOR SYSTEM_VERSION AS OF** clauses are also supported. They provide the same functions as **TIMESTAMP AS OF** and **VERSION AS OF**.

  – Example 1:
  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_TIME AS OF '1986-10-26 01:21:00';
  ```
  – Example 2:
  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 10963874102873;
  ```
  – Example 3:
  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 'audit-branch';
  ```
  – Example 4:
  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_VERSION AS OF 'historical-snapshot';
  ```

  Timestamps may also be supplied as a Unix timestamp, in seconds. For example:

  ```
  SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF 499162860;
  ```

  or

  ```
  SELECT * FROM spark_catalog.db.table FOR SYSTEM_TIME AS OF 499162860;
  ```

  The branch or tag may also be specified using a similar syntax to metadata tables, with **branch**_*<Branch name>* or **tag**_*<Tag name>*.

  ```
  SELECT * FROM spark_catalog.db.table.`branch_audit-branch`;
  ```

  or

  ```
  SELECT * FROM spark_catalog.db.table.`tag_historical-snapshot`;
  ```

  Identifiers with **-** are not valid, and so must be escaped using back quotes (`). Note that the identifier with branch or tag (specified by **branch**_*<Branch*

 Copyright © Huawei Technologies Co., Ltd.

name> or **tag_***<Tag name>*) may not be used in combination with **VERSION AS OF**.

- Schema selection in time travel queries

  Different time travel queries can use either the snapshot's schema or the table's schema based on their purposes. The rules are as follows:

  – Use the snapshot's schema.

    - Example 1:
      ```
      SELECT * FROM spark_catalog.db.table TIMESTAMP AS OF '1986-10-26 01:21:00';
      ```
    - Example 2:
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 10963874102873;
      ```

  – Use the table's schema.

    - Example 1:
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 'audit-branch';
      ```
    - Example 2:
      ```
      SELECT * FROM spark_catalog.db.table.`branch_audit-branch`;
      ```

  – Use the snapshot's schema.

    - Example 1:
      ```
      SELECT * FROM spark_catalog.db.table VERSION AS OF 'historical-snapshot';
      ```
    - Example 2:
      ```
      SELECT * FROM spark_catalog.db.table.`tag_historical-snapshot`;
      ```

- DataFrame

  To select a specific table snapshot or the snapshot at a specified time point in the DataFrame API, Iceberg supports four Spark read options:

  – **snapshot-id**: Select a specific table snapshot. You need to specify the snapshot ID.

  – **as-of-timestamp**: Select the current snapshot at a Unix timestamp, in milliseconds.

  – **branch**: Select the head snapshot of the specified branch.

  – **tag**: Select the snapshot associated with the specified tag.

  Examples:

  – Time travel to October 26, 1986 at 01:21:00
    ```
    spark.read.option("as-of-timestamp", "499162860000").format("iceberg").load("path/to/table")
    ```

  – Time travel to the snapshot with ID **10963874102873**
    ```
    spark.read.option("snapshot-id", 10963874102873L).format("iceberg").load("path/to/table")
    ```

  – Time travel to the snapshot referenced by the tag **historical-snapshot**
    ```
    spark.read.option(SparkReadOptions.TAG, "historical-snapshot").format("iceberg").load("path/to/table")
    ```

  – Time travel to the head snapshot of **audit-branch**.
    ```
    spark.read.option(SparkReadOptions.BRANCH, "audit-branch").format("iceberg").load("path/to/table")
    ```

- Incremental read

  To read appended data incrementally, use the following options:

  – **start-snapshot-id**: Start snapshot ID used in incremental scans (exclusive).

  – **end-snapshot-id**: End snapshot ID used in incremental scans (inclusive). This is optional. Omitting it will default to the current snapshot.

 Copyright © Huawei Technologies Co., Ltd.

```
spark.read.format("iceberg").option("start-snapshot-id", "10963874102873").option("end-snapshot-id", "63874143573109").load("path/to/table")
```

Inspecting Tables

Iceberg allows you to inspect a table's history, snapshots, and other metadata using metadata tables.

Metadata tables are identified by adding the metadata table name after the original table name. For example, the history for table *db.table* is read using *db.table*.**history**. Note that a level-3 or level-2 namespace, that is, *spark_catalog.db.table*.**history** or *db.table*.**history**, must be used. A level-1 namespace, that is, *table*.**history**, is not supported.

- History

  Use the following statement to show table history:

  ```
  SELECT * FROM spark_catalog.db.table.history;
  ```

- Metadata log entries

  Use the following statement to show table metadata log entries:

  ```
  SELECT * from spark_catalog.db.table.metadata_log_entries;
  ```

- Snapshots

  Use the following statement to show all valid snapshots for a table:

  ```
  SELECT * FROM spark_catalog.db.table.snapshots;
  ```

- Entries

  Use the following statement to show all the table's current manifest entries for both data and delete files.

  ```
  SELECT * FROM spark_catalog.db.table.entries;
  ```

- Files

  Use the following statement to show a table's current files:

  ```
  SELECT * FROM spark_catalog.db.table.files;
  ```

- Manifests

  Use the following statement to show a table's current file manifests:

  ```
  SELECT * FROM spark_catalog.db.table.manifests;
  ```

  Fields within the **partition_summaries** column of the manifests table correspond to the **field_summary** structs within the manifest list, with the following order:

  – contains_null
  – contains_nan
  – lower_bound
  – upper_bound

  contains_nan could return null, which indicates that this information is not available from the file's metadata. This usually occurs when reading from the V1 table, where **contains_nan** is not populated.

- Partitions

  Use the following statement to show a table's current partitions:

  ```
  SELECT * FROM spark_catalog.db.table.partitions;
  ```

  For unpartitioned tables, the partitions table will not contain the **partition** and **spec_id** fields.

 Copyright © Huawei Technologies Co., Ltd.

The partitions metadata table shows partitions with data files or delete files in the current snapshot. However, delete files are not applied, and so in some cases, partitions may be shown even though all their data rows are marked deleted by delete files.

- Positional delete files

  Use the following statement to show all positional delete files from the current snapshot of table:

  ```
  SELECT * from spark_catalog.db.table.position_deletes;
  ```

- All data files

  Use the following statement to show all of the table's data files and each file's metadata:

  ```
  SELECT * FROM spark_catalog.db.table.all_data_files;
  ```

- All delete files

  Use the following statement to show the table's delete files and each file's metadata from all the snapshots:

  ```
  SELECT * FROM spark_catalog.db.table.all_delete_files;
  ```

- All entries

  Use the following statement to show the table's manifest entries from all the snapshots for both data and delete files:

  ```
  SELECT * FROM spark_catalog.db.table.all_entries;
  ```

- All manifests

  Use the following statement to show all of the table's manifest files:

  ```
  SELECT * FROM spark_catalog.db.table.all_manifests;
  ```

  Fields within the **partition_summaries** column of the manifests table correspond to the **field_summary** structs within the manifest list, with the following order:

  – contains_null
  – contains_nan
  – lower_bound
  – upper_bound

  contains_nan could return null, which indicates that this information is not available from the file's metadata. This usually occurs when reading from the V1 table, where **contains_nan** is not populated.

- References

  Use the following statement to show a table's known snapshot references:

  ```
  SELECT * FROM spark_catalog.db.table.refs;
  ```

- Inspecting with DataFrames

  Metadata tables can be loaded using the DataFrameReader API:

  – Example 1:
    ```
    spark.read.format("iceberg").load("db.table.files");
    ```
  – Example 2:
    ```
    spark.read.format("iceberg").load("obs://bucket/path/to/table#files");
    ```

- Time travel with metadata tables

  Inspect tables' metadata with the time travel feature.

  – Example 1:
    ```
    SELECT * FROM spark_catalog.db.table.manifests TIMESTAMP AS OF '2021-09-20 08:00:00';
    ```

 Copyright © Huawei Technologies Co., Ltd.

- Example 2:
  ```
  SELECT * FROM spark_catalog.db.table.partitions VERSION AS OF 10963874102873;
  ```

Metadata tables can also be inspected with time travel using the DataFrameReader API.

```
spark.read.format("iceberg").option("snapshot-id", 10963874102873L).load("db.table.files")
```

1.3.5 Writing Data to an Iceberg Table

Iceberg uses Apache Spark's DataSourceV2 API for data source and catalog implementations. Some functions are only available when using Iceberg SQL extensions in Spark.

Table 1-17 DataSourceV2 API functions supported by Spark

Function	Description	Usage Requirements
INSERT INTO	SQL statement used to append new data to a table.	Requires the setting of **spark.sql.storeAssignmentPolicy=ANSI**.
MERGE INTO	SQL statement used for row-level updates.	Requires Iceberg Spark extensions.
INSERT OVERWRITE	SQL statement used to replace data in the table with the result of a query.	Requires the setting of **spark.sql.storeAssignmentPolicy=ANSI**.
DELETE FROM	SQL statement used to remove data that meets criteria from tables.	Row-level delete requires Iceberg SQL extensions.
UPDATE	SQL statement used by update queries to accept a filter to match rows to update and update fields in the rows.	Requires Iceberg SQL extensions.
DataFrame appending	You can use **append** to append a DataFrame to an Iceberg table.	None
DataFrame overwriting	You can use **overwritePartitions()** to overwrite partitions dynamically.	None
DataFrame CTAS and RTAS	You can use the **create**, **replace**, or **createOrReplace** to run a CTAS or RTAS to perform creation and replacement operations.	Requires the DataSourceV2 API.
DataFrame merge into	You can use **mergeInto** to merge data into a table.	Requires the DataSourceV2 API.

Writing Data with SQL

Spark supports SQL **INSERT INTO**, **MERGE INTO**, and **INSERT OVERWRITE**, as well as the new DataFrameWriterV2 API.

 Copyright © Huawei Technologies Co., Ltd.

- INSERT INTO

  To append new data to a table, use **INSERT INTO**.

  Create a table.

  ```
  CREATE TABLE spark_catalog.db.table (id bigint, data string) USING iceberg;
  ```

  Write data into the table.

  - Example 1:
    ```
    INSERT INTO spark_catalog.db.table VALUES (1, 'a'), (2, 'b');
    ```
  - Example 2:
    ```
    INSERT INTO spark_catalog.db.table SELECT ...;
    ```

- MERGE INTO

  Spark added support for **MERGE INTO** queries that can express row-level updates.

  Iceberg supports **MERGE INTO** by rewriting data files that contain rows that need to be updated in an overwrite commit. **MERGE INTO** is recommended instead of **INSERT OVERWRITE** because Iceberg can replace only the affected data files, and the data overwritten by a dynamic overwrite may change if the table's partitioning changes.

  MERGE INTO updates the target table using a set of updates from another query. The update for a row in the target table is found using the **ON** clause.

  ```
  MERGE INTO spark_catalog.db.target t
  USING (SELECT ...) s
  ON t.id = s.id
  WHEN ...;
  ```

  Updates to rows in the target table are listed using **WHEN MATCHED ... THEN .....** Multiple **MATCHED** clauses can be added with conditions that determine when each match should be applied. The first matching expression is used.

  - Example 1:
    ```
    WHEN MATCHED AND s.op = 'delete' THEN DELETE
    ```
  - Example 2:
    ```
    WHEN MATCHED AND t.count IS NULL AND s.op = 'increment' THEN UPDATE SET t.count = 0
    ```
  - Example 3:
    ```
    WHEN MATCHED AND s.op = 'increment' THEN UPDATE SET t.count = t.count + 1
    ```

  Source rows (updates) that do not match can be inserted into the target table using **WHEN NOT MATCHED**.

  ```
  WHEN NOT MATCHED THEN INSERT *
  ```

  Inserts also support additional conditions.

  ```
  WHEN NOT MATCHED AND s.event_time > still_valid_threshold THEN INSERT (id, count) VALUES (s.id, 1)
  ```

  Only one record in the source data can update any given row of the target table, or else an error will be thrown.

  Source rows (updates) that do not match can be updated or deleted from the target table using **WHEN NOT MATCHED**.

  ```
  WHEN NOT MATCHED BY SOURCE THEN UPDATE SET status = 'invalid'
  ```

- INSERT OVERWRITE

  INSERT OVERWRITE can replace data in the table with the result of a query. Overwrites are atomic operations for Iceberg tables.

  The partitions that will be replaced by **INSERT OVERWRITE** depend on Spark's partition overwrite mode and the partitioning of a table. **MERGE**

 Copyright © Huawei Technologies Co., Ltd.

INTO can rewrite only affected data files and has more easily understood behavior, so it is recommended instead of **INSERT OVERWRITE**.

– Overwrite behavior

The Spark's default overwrite mode is static, but the dynamic overwrite mode is recommended when you write data to Iceberg tables. The static overwrite mode determines which partitions to overwrite in a table by converting the **PARTITION** clause to a filter, but the **PARTITION** clause can only reference table columns.

The dynamic overwrite mode is configured by setting **spark.sql.sources.partitionOverwriteMode=dynamic**.

The following example demonstrates the behavior of dynamic and static overwrites:

Create a **logs** table defined by the following DDL:

```
CREATE TABLE spark_catalog.my_app.logs (uuid string NOT NULL,  level string NOT NULL, ts timestamp NOT NULL, message string) USING iceberg PARTITIONED BY (level, hours(ts));
```

▪ Dynamic overwrite

When the Spark's overwrite mode is dynamic, partitions that have rows in the query result will be replaced.

For example, this query removes duplicate log events from the **logs** table.

```
INSERT OVERWRITE spark_catalog.my_app.logs SELECT uuid, first(level), first(ts), first(message) FROM spark_catalog.my_app.logs WHERE cast(ts as date) = '2020-07-01' GROUP BY uuid;
```

In dynamic mode, this operation will replace any partition with rows in the query result. Because the date of all rows is restricted to July 1, only hours of that day will be replaced.

▪ Static overwrite

When the Spark's overwrite mode is static, the **PARTITION** clause is converted to a filter that is used to delete data from the table. If the **PARTITION** clause is omitted, all partitions will be replaced.

Because there is no **PARTITION** clause in the query above, it will drop all existing rows in the table when in static mode, but will only write the logs from July 1.

To overwrite just the partitions that were loaded, add a **PARTITION** clause that aligns with the **SELECT** query filter.

```
INSERT OVERWRITE spark_catalog.my_app.logs PARTITION (level = 'INFO') SELECT uuid, first(level), first(ts), first(message) FROM spark_catalog.my_app.logs WHERE level = 'INFO' GROUP BY uuid;
```

Note that this mode cannot replace hourly partitions like the dynamic example query because the **PARTITION** clause can only reference table columns, not hidden partitions.

- DELETE FROM

Spark added support for **DELETE FROM** queries to remove data from tables.

Delete queries accept a filter (**WHERE** clause) to match rows to delete. Only the rows that match the filter are deleted, and the operation is atomic for Iceberg tables.

Example 1:

 Copyright © Huawei Technologies Co., Ltd.

```
DELETE FROM spark_catalog.db.table WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01 00:00:00';
```

Example 2:

```
DELETE FROM spark_catalog.db.all_events WHERE session_time < (SELECT min(session_time) FROM spark_catalog.db.good_events);
```

Example 3:

```
DELETE FROM spark_catalog.db.orders AS t1 WHERE EXISTS (SELECT oid FROM spark_catalog.db.returned_orders WHERE t1.oid = oid);
```

If the delete filter matches entire partitions of the table, Iceberg will perform a metadata-only delete without rewriting data files, providing high performance.

If the filter matches individual rows of a table, then Iceberg will rewrite only the affected data files. It reads the original data file, filters out the rows to be deleted, writes to a new file, and updates the metadata to reference the new file. This approach minimizes the amount of data processed.

- UPDATE

  UPDATE queries accept a filter to match rows to update and update fields of the rows.

  Example 1:

  ```
  UPDATE spark_catalog.db.table SET c1 = 'update_c1', c2 = 'update_c2' WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01 00:00:00';
  ```

  Example 2:

  ```
  UPDATE spark_catalog.db.all_events SET session_time = 0, ignored = true WHERE session_time < (SELECT min(session_time) FROM spark_catalog.db.good_events);
  ```

  Example 3:

  ```
  UPDATE spark_catalog.db.orders AS t1 SET order_status = 'returned' WHERE EXISTS (SELECT oid FROM spark_catalog.db.returned_orders WHERE t1.oid = oid);
  ```

  For more complex row-level updates based on incoming data, see the section on **MERGE INTO**.

Writing to Branches

The branch must exist before performing the write. The operation does not create the branch if it does not exist. For more information on branches, refer to **Branching and Tagging DDL Syntax**.

- Writing to a branch using SQL statements

  Branch writes can be performed via SQL by providing a branch identifier, **branch_yourBranch**, in the operation. Branch writes can also be performed as part of a write-audit-publish (WAP) workflow by specifying the **spark.wap.branch** configuration. Note that the WAP branch and branch identifier cannot both be specified.

  MERGE INTO branch data:

  ```
  MERGE INTO spark_catalog.db.table.branch_audit t USING (SELECT ...) s  ON t.id = s.id  WHEN ...;
  ```

  UPDATE branch data:

  ```
  UPDATE spark_catalog.db.table.branch_audit AS t1 SET val = 'c';
  ```

  Delete branch data.

  ```
  DELETE FROM spark_catalog.db.table.branch_audit WHERE id = 2;
  ```

  WAP branch write:

 Copyright © Huawei Technologies Co., Ltd.

a. Configure **spark.wap.branch**.
```
SET spark.wap.branch = audit-branch;
```
b. Insert data.
```
INSERT INTO spark_catalog.db.table VALUES (3, 'c');
```

- Writing data to a branch using DataFrames

  When writing data to a branch using DataFrames, you also need to specify the branch identifier **branch_yourBranch** in the operation.

  Example 1:

```
val data: DataFrame = ...
data.writeTo("spark_catalog.db.table.branch_audit").append();
```

  Example 2:

```
val data: DataFrame = ...
data.writeTo("spark_catalog.db.table.branch_audit").overwritePartitions();
```

Writing with DataFrames

Spark introduced the new DataFrameWriterV2 API for writing to tables using DataFrames. The v2 API is recommended for several reasons:

- CTAS, RTAS, and overwrite by filter are supported.
- All operations consistently write columns to a table by name.
- Hidden partition expressions are supported in **partitionedBy**.
- Overwrite behavior is explicit, either dynamic or by a user-supplied filter.
- The behavior of each operation corresponds to the following SQL statements:
  - **df.writeTo(t).create()** is equivalent to **CREATE TABLE AS SELECT**.
  - **df.writeTo(t).replace()** is equivalent to **REPLACE TABLE AS SELECT**.
  - **df.writeTo(t).append()** is equivalent to **INSERT INTO**.
  - **df.writeTo(t).overwritePartitions()** is equivalent to **dynamic INSERT OVERWRITE**.

The v1 DataFrame write API is still supported, but is not recommended.

NOTE

When using the v1 DataFrame API for writes in Spark 3, ensure that you use saveAsTable or insertInto to load tables through Catalog. Using **format("iceberg")** directly will load an isolated table reference, which means that the table used for queries cannot be automatically refreshed.

- Appending data

  To append a DataFrame to an Iceberg table, use **append**.

```
val data: DataFrame = ...
data.writeTo("spark_catalog.db.table").append()
```

- Overwriting data

  To overwrite partitions dynamically, use **overwritePartitions()**.

```
val data: DataFrame = ...
data.writeTo("spark_catalog.db.table").overwritePartitions()
```

  To explicitly overwrite partitions, use **overwrite** to supply a filter.

```
data.writeTo("spark_catalog.db.table").overwrite($"level" === "INFO")
```

- Creating a table

  To run a CTAS or RTAS, use **create**, **replace**, or **createOrReplace** operations.

 Copyright © Huawei Technologies Co., Ltd.

```
val data: DataFrame = ...
data.writeTo("spark_catalog.db.table").create()
```

If you have replaced the default Spark catalog (**spark_catalog**) with Iceberg's **SparkSessionCatalog**, run the following command:

```
val data: DataFrame = ...
data.writeTo("db.table").using("iceberg").create()
```

Create and replace operations support table configuration methods, like **partitionedBy** and **tableProperty**.

```
data.writeTo("spark_catalog.db.table").tableProperty("write.format.default",
"orc").partitionedBy($"level", days($"ts")).createOrReplace()
```

The Iceberg table location can also be specified by the **location** table property.

```
data.writeTo("spark_catalog.db.table").tableProperty("location", "/path/to/
location").createOrReplace()
```

- Merging data

  Spark 4.0 adds support for executing MERGE INTO queries using the DataFrameWriterV2 API.

  A MERGE INTO query updates a target table using a set of update operations from the source data, which in this case is a DataFrame.

  ```
  val source: DataFrame = ...                                  // For example, read a table as the source data.
  source.mergeInto("target", $"source.id" === $"target.id") // The second parameter is the ON condition.
      .whenMatched($"target.id" === 1)                         // Additional condition
      .updateAll()                                          // UPDATE SET *
      .whenMatched($"target.id" === 2)
      .delete()
      .whenNotMatched()
      .insertAll()                                          // INSERT *
      .whenNotMatchedBySource($"target.id" === 3)
      .update(Map("status" -> lit("invalid")))              // Update columns.
      .merge()
  ```

- Schema merging

  While inserting or updating, Iceberg is capable of resolving schema mismatch at runtime. If properly configured, Iceberg will perform an automatic schema evolution as follows:

  – A new column is present in the source but not in the target table: The new column is added to the target table. Column values are set to NULL in all the rows already present in the table.

  – A column is present in the target but not in the source: The target column value is set to NULL during inserting or left unchanged during row updating.

  The target table must be configured to accept any schema change by setting the property **write.spark.accept-any-schema** to **true**.

  ```
  ALTER TABLE spark_catalog.db.sample SET TBLPROPERTIES ('write.spark.accept-any-schema'='true');
  ```

  The writer must enable the **mergeSchema** option.

  ```
  data.writeTo("spark_catalog.db.sample").option("mergeSchema","true").append()
  ```

Writing Distribution Modes

Iceberg's default Spark writers require that the data in each spark task is clustered by partition values. This distribution is required to minimize the number of file handles that are held open while writing. By default, starting in Iceberg 1.2.0, Iceberg also requests that Spark pre-sort data to be written to fit this distribution.

 Copyright © Huawei Technologies Co., Ltd.

The request to Spark is done through the table property **write.distribution-mode** with the value **hash**.

The following examples demonstrate the writing distribution modes:

Create a table.

```
CREATE TABLE spark_catalog.db.sample (id bigint, data string, category string, ts timestamp) USING iceberg
PARTITIONED BY (days(ts), category);
```

To write data to the table, data needs to be sorted by **days(ts)** and **category** but this is done automatically by the default hash distribution. For example:

```
INSERT INTO spark_catalog.db.sample
SELECT id, data, category, ts FROM another_table;
```

The **write.distribution-mode** parameter supports three options, each affecting how Spark handles data partitions and sorts data.

- **none**: This is the default value for Iceberg prior to 1.2.0.
  - Core logic: This mode does not request any shuffles or sort to be performed automatically by Spark.
  - Data requirements: The data must be manually sorted by partition value (either within each spark task, or globally within the entire dataset). A global sort will minimize the number of output files.
  - Special processing: A sort can be avoided by using the Spark **write fanout** property but this will cause all file handles to remain open until each write task has completed, increasing resource usage.
  - Applicable scenarios: Data is already sorted by partition values (for example, the upstream task output is in required order), and no additional shuffle overhead is required.
- **hash**: This is the default value for Iceberg 1.2.0 and later.
  - Core logic: This mode requests that Spark uses a hash-based exchange to shuffle the incoming write data before writing.
  - Procedure: Each row is hashed based on the row's partition value and then placed in a corresponding Spark task based on that value. Further spiting and coalescing of tasks may take place because of Spark's Adaptive Query planning.
  - Advantages: Data is automatically clustered by partition values, eliminating manual sorting and balancing performance with resource efficiency.
  - Applicable scenarios: Most common write scenarios, particularly when data is not pre-sorted and requires automatic adaptation to partition distribution.
- **range**
  - Core logic: This mode requests that Spark perform a range-based exchange to shuffle the data and globally sort data before writing.
  - Procedure (two stages)
    i. Sampling stage: The first stage samples the data to be written based on the partition and sort columns to determine range distribution information about the data.
    ii. Shuffle stage: The second stage uses the obtained range information to shuffle the input data into Spark tasks. Each task gets an exclusive

 Copyright © Huawei Technologies Co., Ltd.

range of the input data which clusters the data by partition and also globally sorts.

- Characteristics:
  - It is more expensive than the **hash** distribution, requiring additional sampling and global ordering.
  - Global ordering can be beneficial for read performance if sorted columns are used during queries. Pre-sorted data reduces computational overhead during reads.
- Triggered by default: This mode is used by default if a table is created with a sort-order. Parameter **write.distribution-mode** is automatically set to **range**.
- Applicable scenarios: Tables that require sorting (for example, by time) and workloads where read performance is prioritized over write overhead.

Controlling File Sizes

When you write data to Iceberg with Spark, it is important to note that:

- Spark cannot write a file larger than a Spark task and a file cannot span an Iceberg partition boundary. This means although Iceberg will always roll over a file when it grows to the **write.target-file-size-bytes** threshold, this rollover may not occur unless the Spark task produces a file large enough to exceed that size.
- The size of the file created on disk will also be much smaller than the Spark task since the disk data will be both compressed and in columnar format as opposed to Spark's uncompressed row representation. This means a 100 MB Spark task will create a file much smaller than 100 MB even if that task is writing to a single Iceberg partition. If the task writes to multiple partitions, the files will be even smaller than that.

To control what data ends up in each Spark task, use a write distribution mode or manually repartition the data.

To adjust Spark's task size, it is important to become familiar with Spark's various Adaptive Query Execution (AQE) parameters. When **write.distribution-mode** is not **none**, AQE will control the coalescing and splitting of Spark tasks during the exchange to try to create tasks of **spark.sql.adaptive.advisoryPartitionSizeInBytes** size. These settings will also affect any user-performed re-partitions or sorts. It is important again to note that this is the in-memory Spark row size and not the on-disk, columnar-compressed size, so a larger value than the target file size will need to be specified. The ratio of in-memory size to on-disk size is data dependent.

Iceberg Idempotent Write

When the current task is interrupted and restarted when writing data to the Iceberg table, some data may have been written before the task is terminated. After the task restart, data may be repeatedly written into the table. Iceberg's idempotent write can be enabled to avoid this problem.

Enabling Iceberg idempotent write

 Copyright © Huawei Technologies Co., Ltd.

In the spark-sql command, set **spark.sql.iceberg.write.idempotent.enabled** to **true** to enable idempotent write. Its default value is **false**.

Other configuration items

Table 1-18 Other configuration items

Item	Parameter	Description
Name of an idempotent write task	spark.sql.iceberg.write.txnAppId	**txnAppId**: the exclusive ID of a task, which is a string, for example, **dailyETL**.
Idempotent write version number	spark.sql.iceberg.write.txnVersion	**txnVersion**: The value can be customized and must be monotonically increasing. It is used as the transaction version number. The value must be unique for each batch of data written to the same Iceberg table. The value is an integer, for example, **23423**.
Whether to automatically reset the version number	spark.sql.iceberg.write.txnVersion.autoReset.enabled	If this parameter is set to **true**, the version number is automatically reset after each write operation, and you do not need to set **txnVersion** again.

Each batch of new data ingested must have a unique combination of **txnAppId** and **txnVersion**, and the value of **txnVersion** must be greater than that of the previous batch. For example:

1. Suppose that the parameter combination of the previous batch of data that was successfully written is **dailyETL**:**23423** (**txnAppId**:**txnVersion**).

2. For the next batch, **txnAppId** must be set to **dailyETL**, and **txnVersion** must be set to at least **23424** (increased by 1 based on the previous version).

3. If **dailyETL** is used together with a version number of 23422 or earlier, Iceberg will ignore the write request (the version number is earlier than the latest version recorded in the table).

4. If **anotherETL:23424** is used for writing, the write operation can be performed normally because the task ID is different from that of the historical data.

Priority description

If both the write parameter and session configurations are set, the write parameter takes precedence.

Examples:

```
spark-sql
--conf spark.sql.iceberg.write.idempotent.enabled=true
```

 Copyright © Huawei Technologies Co., Ltd.

```
--conf spark.sql.iceberg.write.txnAppId=dailyETL

spark-sql >
set spark.sql.iceberg.write.txnVersion=1;
insert into Test_Iceberg select * from Test_Source;   //txnVersion is 1 and the write operation is successful.
insert into Test_Iceberg select * from Test_Source;   //autoReset is false and txnVersion is still 1, the write operation fails with an exception thrown. The same txnVersion cannot be used for different INSERT operations.

-- AUTO RESET SAMPLE USAGE
set spark.sql.iceberg.write.txnVersion.autoReset.enabled=true;
set spark.sql.iceberg.write.txnVersion=2;

insert into Test_Iceberg select * from Test_Source;   //txnVersion is 2 and no INSERT operation is performed for this txnVersion. The write operation is successful.
insert into Test_Iceberg select * from Test_Source;   //txnVersion is automatically updated to 3 and the write operation is successful.
```

1.3.6 Stored Procedures for Iceberg Tables

To use Iceberg in Spark, first configure Spark catalogs. For details about how to configure **spark_catalog** and **prod** catalogs, see **1.3.2 Iceberg Configurations and Types**. Stored procedures are only available when you use Iceberg SQL extensions in Spark 3.

How to Use

Stored procedures can be used from any configured Iceberg catalog with **CALL**. All procedures are in the namespace system.

CALL supports passing arguments by name (recommended) or by position. Mixing position and named arguments is not supported.

- Named arguments

  All procedure arguments are named. When passing arguments by name, arguments can be in any order and any optional argument can be omitted.

  ```
  CALL spark_catalog.system.{Storage name} (arg_name_2 => 'arg_2_value', arg_name_1 => 'arg_1_value');
  ```

- Positional arguments

  When you pass arguments by position, only the ending arguments may be omitted if they are optional.

  ```
  CALL spark_catalog.system.{Storage name} (arg_1, arg_2, ... arg_n);
  ```

Snapshot Management

- **rollback_to_snapshot**

  You can use **rollback_to_snapshot** to roll back a table to a specific snapshot ID. The following command rolls back table **db.sample** to the snapshot whose ID is **1**:

  ```
  CALL catalog_name.system.rollback_to_snapshot('db.sample', 1);
  ```

  The table name field is of the string type, and the snapshot ID field is of the long type.

  Table 1-19 describes the output results.

 Copyright © Huawei Technologies Co., Ltd.

Table 1-19 Output result

Parameter	Type	Description
previous_snapshot_id	long	The current snapshot ID before the rollback
current_snapshot_id	long	The new current snapshot ID after the rollback

- **rollback_to_timestamp**

  You can use **rollback_to_timestamp** to roll back a table to the snapshot that was current at some time. The following command rolls back table **db.sample** to a specific day and time.

  ```
  CALL spark_catalog.system.rollback_to_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000');
  ```

  The table name field is of the string type, and the timestamp field is of the timestamp type, which indicates the target timestamp to roll back to.

  Table 1-19 describes the output results.

- **set_current_snapshot**

  You can use **set_current_snapshot** to set the current snapshot ID of a table. Unlike rollback, the snapshot is not required to be an ancestor of the current table state. For example:

  Set the current snapshot for table **db.sample** to the snapshot whose ID is **1**.

  ```
  CALL spark_catalog.system.set_current_snapshot('db.sample', 1);
  ```

  Set the current snapshot for table **db.sample** to the snapshot whose tag is **s1**.

  ```
  CALL spark_catalog.system.set_current_snapshot(table => 'db.sample', ref => 's1');
  ```

  The table name field is of the string type, the snapshot ID field is of the long type, and **ref** is the snapshot reference (branch or tag) of the current snapshot, which is of the string type. Either **snapshot_id** or **ref** must be provided but not both.

  Table 1-20 describes the output results.

Table 1-20 Snapshot setting output

Parameter	Type	Description
previous_snapshot_id	long	The table's current snapshot ID before the setting
current_snapshot_id	long	The table's new snapshot ID after the setting

- **cherrypick_snapshot**

  You can use **cherrypick_snapshot** to pick changes from a specified snapshot and apply them to the current table state. Cherry-picking creates a new snapshot from an existing snapshot without altering or removing the original.

 Copyright © Huawei Technologies Co., Ltd.

Only append and dynamic overwrite snapshots can be cherry-picked. For example:

Cherry-pick changes from snapshot 1.

```
CALL spark_catalog.system.cherrypick_snapshot('my_table', 1);
```

Cherry-pick change from snapshot 1 with named arguments.

```
CALL spark_catalog.system.cherrypick_snapshot(snapshot_id => 1, table => 'my_table' );
```

Parameter **table** indicates the name of the table to update. The value is of the string type. Parameter **snapshot_id** indicates the ID of the snapshot to cherry-pick. The value is of the long type.

Table 1-21 describes the output results.

Table 1-21 Snapshot change output

Parameter	Type	Description
source_snapshot_id	long	The table's current snapshot ID before the cherry-pick
current_snapshot_id	long	The snapshot ID created by applying the cherry-pick

- **publish_changes**

  You can use **publish_changes** to publish changes from a staged WAP ID into the current table state. **publish_changes** creates a new snapshot from an existing snapshot without altering or removing the original. Only append and dynamic overwrite snapshots can be successfully published. For example:

  Publish changes with WAP ID **wap_id_1**.

  ```
  CALL spark_catalog.system.publish_changes('my_table', 'wap_id_1');
  ```

  Publish changes with named arguments.

  ```
  CALL spark_catalog.system.publish_changes(wap_id => 'wap_id_2', table => 'my_table');
  ```

  Parameter **table** indicates the name of the table to update. The value is of the string type. Parameter **wap_id** indicates the WAP ID to be published from the staging environment. The value is of the long type.

  Table 1-22 describes the output results.

  Table 1-22 Snapshot change output

Parameter	Type	Description
source_snapshot_id	long	The table's current snapshot ID before publishing the change
current_snapshot_id	long	The snapshot ID created by applying the change

 Copyright © Huawei Technologies Co., Ltd.

- **fast_forward**

  You can use **fast_forward** to fast-forward the current snapshot of one branch to the latest snapshot of another. For example:

  Fast-forward the current snapshot of the **main** branch to the latest snapshot of the **audit-branch** branch.

  ```
  CALL spark_catalog.system.fast_forward('my_table', 'main', 'audit-branch');
  ```

  Both the table field and branch field are of the string type.

  Table 1-23 describes the output results.

  Table 1-23 Snapshot update output

Parameter	Type	Description
branch_updated	string	The name of the branch that has been fast-forwarded
previous_ref	long	The snapshot ID before applying fast-forward
updated_ref	long	The current snapshot ID after applying fast-forward

Metadata Management

expire_snapshots

Each write, update, delete, upsert, compaction in Iceberg produces a new snapshot while keeping the old data and metadata around for snapshot isolation and time travel. The **expire_snapshots** procedure can be used to remove older snapshots and their files which are no longer needed.

This procedure will remove old snapshots and data files which are uniquely required by those old snapshots. This means the **expire_snapshots** procedure will never remove files which are still required by a non-expired snapshot. For example:

- Remove snapshots older than specific day and time, but retain the last 100 snapshots.
  ```
  CALL spark_catalog.system.expire_snapshots('db.sample', TIMESTAMP '2021-06-30 00:00:00.000', 100);
  ```
- Remove snapshots with snapshot ID **123** (note that this snapshot ID should not belong to the current snapshot).
  ```
  CALL spark_catalog.system.expire_snapshots('db.sample', snapshot_ids => array(123));
  ```

For details about related parameters, see **Table 1-24**.

Table 1-24 expire_snapshots parameters

Parameter	Type	Description
table	string	The name of the table to update. This parameter is mandatory.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
older_than	timestamp	The timestamp before which snapshots will be removed (default: 5 days ago)
retain_last	int	The number of ancestor snapshots to preserve regardless of parameter **older_than** (defaults to 1)
max_concurrent_deletes	int	The size of the thread pool used for delete file actions (by default, no thread pool is used)
stream_results	boolean	When it is set to **true**, delete files will be sent to the Spark driver by RDD partition (by default, all the files will be sent to the Spark driver). It is recommended that you set this option to **true** to prevent Spark driver OOM from a large file size.
snapshot_ids	array of long	The array of snapshot IDs to expire
clean_expired_metadata	boolean	If this parameter is set to **true**, metadata that is no longer referenced by snapshots will be cleared, such as partition specs and schemas.

If **older_than** and **retain_last** are omitted, the table's expiration properties will be used. Snapshots that are still referenced by branches or tags will not be removed. By default, branches and tags never expire, but their retention policy can be changed with the table property **history.expire.max-ref-age-ms**. The main branch never expires.

Table 1-25 describes the output results.

Table 1-25 expire_snapshots output

Parameter	Type	Description
deleted_data_files_count	long	The number of data files deleted by this operation
deleted_position_delete_files_count	long	Number of **position delete files** deleted by this operation
deleted_equality_delete_files_count	long	Number of **equality delete files** deleted by this operation
deleted_manifest_files_count	long	The number of manifest files deleted by this operation
deleted_manifest_lists_count	long	The number of manifest list files deleted by this operation

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
deleted_statistics_files_count	long	Number of statistics files deleted by this operation

remove_orphan_files

remove_orphan_files can be used to remove files which are not referenced in any metadata files of an Iceberg table and can thus be considered "orphaned".

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	Name of the table to be removed
older_than	No	timestamp	Removes orphaned files created before this timestamp (default: 3 days ago).
location	No	string	Directory where files are to be searched (default: table storage location)
dry_run	No	boolean	If set to **true**, files are not actually deleted (default value: **false**).
max_concurrent_deletes	No	int	The size of the thread pool used for delete file actions (by default, no thread pool is used)
file_list_view	No	string	The dataset to look for files in (skipping the directory listing)
equal_schemes	No	map	The mapping of file system schemes to be considered equal. The key is a comma-separated list of schemes, and the value is a scheme (defaults to **map('s3a,s3n','s3')**).
equal_authorities	No	map	The mapping of file system authorities to be considered equal. The key is a comma-separated list of authorities, and the value is an authority.
prefix_mismatch_mode	No	string	The action behavior when the location prefix (scheme/authority) does not match: • **ERROR** (default): Throw an exception. • **IGNORE**: Do not perform any operation. **DELETE**: Delete the file.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
prefix_listing	No	boolean	When set to **true**, prefix-based file listing is used through the SupportsPrefixOperations API. When this flag is enabled, the FileIO implementation of the table must support the SupportsPrefixOperations API (default value: **false**).

Output result

Output Parameter	Type	Description
orphan_file_location	String	The path to each file determined to be an orphan by this command

Examples

List all the files that are candidates for removal by performing a dry run of the **remove_orphan_files** command on this table without actually removing them.

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', dry_run => true);
```

Remove any files in the **tablelocation/data** folder that are not known to the table **db.sample**.

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', location => 'tablelocation/data');
```

Remove any files in the **files_view** view that are not known to the table **db.sample**.

```
Dataset<Row> compareToFileList = spark.createDataFrame(allFiles,
FilePathLastModifiedRecord.class).withColumnRenamed("filePath",
"file_path").withColumnRenamed("lastModified", "last_modified");
String fileListViewName = "files_view";
compareToFileList.createOrReplaceTempView(fileListViewName);
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', file_list_view => 'files_view');
```

When a file matches references in metadata files except for location prefix (scheme/authority), an error is thrown by default. The error can be ignored and the file will be skipped by setting **prefix_mismatch_mode** to **IGNORE**.

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', prefix_mismatch_mode => 'IGNORE');
```

The file can still be deleted by setting **prefix_mismatch_mode** to **DELETE**.

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', prefix_mismatch_mode => 'DELETE');
```

The file can also be deleted by considering the mismatched prefixes equal.

```
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', equal_schemes => map('file', 'file1'));
CALL spark_catalog.system.remove_orphan_files(table => 'db.sample', equal_authorities => map('ns1',
'ns2'));
```

 Copyright © Huawei Technologies Co., Ltd.

Lists all candidate files that meet the deletion conditions and scans them using prefix listing.

```
CALL catalog_name.system.remove_orphan_files(table => 'db.sample', prefix_listing => true);
```

rewrite_data_files

Iceberg tracks each data file in the table. A large volume of data files increases the metadata stored in manifest files. Specifically, small data files create excessive metadata and reduce query efficiency due to increased file-opening overhead.

Iceberg can compact data files in parallel using Spark with the **rewriteDataFiles** action. This will combine small files into larger files to reduce metadata overhead and runtime file open cost.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table to be updated
strategy	No	string	The policy name. The value can be **binpack** (default) or **sort**.
sort_order	No	string	The sorting order. Available values include: • **Zorder**: A comma-separated list of columns, in the format of **zorder(*c1*,*c2*,*c3*)**. • **Other**: Comma-separated sorting orders in the format of ***ColumnName SortDirection NullOrder***, where **SortDirection** can be **ASC** or **DESC** and **NullOrder** can be **NULLSFIRST** or **NULLSLAST**. By default, the table sorting order is used.
options	No	map	The options to be used for actions
where	No	string	The predicate as a string used for filtering the files. Note that all files that may contain data matching the filter criteria will be selected for rewriting.

Configuration options

Parameter	Default Value	Description
max-concurrent-file-group-rewrites	5	The maximum number of file groups to be simultaneously rewritten

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Default Value	Description
partial-progress.enabled	false	Whether to enable committing file groups prior to the entire rewrite completing
partial-progress.max-commits	10	The maximum number of commits allowed for this rewrite operation if partial progress is enabled
partial-progress.max-failed-commits	value of **partial-progress.max-commits**	The maximum number of failed commits allowed before job failure, if partial progress is enabled.
use-starting-sequence-number	true	Whether to use the sequence number of the snapshot at compaction start time rather than that of the newly produced snapshot
rewrite-job-order	none	The order of forcibly rewriting jobs. The options are as follows: • **bytes-asc**: Rewrites the smallest job groups first. • **bytes-desc**: Rewrites the largest job groups first. • **files-asc**: Rewrites the job groups with the least files first. • **files-desc**: Rewrites the job groups with the most files first. • **none**: Rewrites job groups in the order they were planned (no specific ordering).
target-file-size-bytes	536870912 (512 MB; the default value inherits from the table property **write.target-file-size-bytes**)	The target output file size

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Default Value	Description
min-file-size-bytes	75% of target file size	Files below this threshold will be considered for rewriting, regardless of other criteria
max-file-size-bytes	180% of target file size	Files above this threshold will be considered for rewriting, regardless of other criteria.
min-input-files	5	Any file group with more files than this value will be rewritten, regardless of other criteria
rewrite-all	false	Whether to forcibly rewrite all provided files and overwrite other options
max-file-group-size-bytes	107374182400 (100 GB)	Largest amount of data that should be rewritten in a single file group. The entire rewrite operation is broken down into pieces based on partitioning and within partitions based on size into file groups. This helps with breaking down the rewriting of very large partitions which may not be rewritable otherwise due to the resource constraints of the cluster.
delete-file-threshold	2147483647	The minimum number of deletes that needs to be associated with a data file for it to be considered for rewriting
delete-ratio-threshold	0.3	The minimum deletion ratio required for a data file to be included in the rewriting scope
output-spec-id	Spec ID of the current partition	The identifier of the output partition specifications. Data will be reorganized during the rewrite to align with the output partitioning.
remove-dangling-deletes	false	Whether to remove dangling position and equality delete files after rewriting. A delete file is considered dangling if it does not apply to any live data files. Enabling this option will generate an additional commit to perform the removal.

Sorting policy options

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Default Value	Description
compression-factor	1.0	The number of shuffle partitions and consequently the number of output files created by the Spark sort is based on the size of the input data files used in this file rewriter. Due to compression, the disk file sizes may not accurately represent the size of files in the output. This parameter lets the user adjust the file size used for estimating the actual output data size. A factor greater than 1.0 would generate more files than expected based on the on-disk file size. A value less than 1.0 would create fewer files than expected based on the on-disk file size.
shuffle-partitions-per-file	1	The number of shuffle partitions to use for each output file. Iceberg will use a custom coalesce operation to stitch these sorted partitions back together into a single sorted file.

ZORDER sorting policy options

Parameter	Default Value	Description
var-length-contribution	8	The number of bytes considered from an input column of a type with variable length (String, Binary).
max-output-size	2147483647	The number of bytes interleaved in the ZORDER algorithm.

Output result

Output Parameter	Type	Description
rewritten_data_files_count	int	The number of data files which were re-written by this command
added_data_files_count	int	The number of new data files which were written by this command
rewritten_bytes_count	long	The number of bytes which were written by this command

 Copyright © Huawei Technologies Co., Ltd.

Output Parameter	Type	Description
failed_data_files_count	int	The number of data files that failed to be rewritten when **partial-progress.enabled** is **true**
removed_delete_files_count	int	The number of files which were deleted by this command

Examples

Rewrite the data files in table **db.sample** using the default rewrite algorithm of bin-packing to combine small files and also split large files according to the default write size of the table.

```
CALL spark_catalog.system.rewrite_data_files('db.sample');
```

Rewrite the data files in table **db.sample** by sorting all data by **id** and **name**. Use the same default values as bin-pack to determine the files to be rewritten.

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'id DESC NULLS LAST,name ASC NULLS FIRST');
```

Rewrite the data files in table **db.sample** by performing ZORDER sorting on columns **c1** and **c2**, and use the same default values as bin-pack to determine which files need to be rewritten:

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', strategy => 'sort', sort_order => 'zorder(c1,c2)');
```

Use the bin-pack policy to rewrite the data files in the partitions where at least two files in table **db.sample** need to be rewritten, and then remove all the dangling files.

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', options => map('min-input-files', '2', 'remove-dangling-deletes', 'true'));
```

Rewrite the data files in table **db.sample** and select the files that may contain data matching the filter criteria (**id=3** and name="**foo**") for rewriting:

```
CALL spark_catalog.system.rewrite_data_files(table => 'db.sample', where => 'id = 3 and name = "foo"');
```

rewrite_manifests

Rewrite the manifest files for a table to optimize scan planning efficiency.

Data files in manifest files are sorted by fields in the partition specifications. This process is executed in parallel using Spark jobs.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table to be updated

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
use_caching	No	boolean	Whether to use Spark caching during operations (defaults to **true**)
spec_id	No	int	The specification ID of the manifest files to rewrite (defaults to the current specification ID)

Output result

Output Parameter	Type	Description
rewritten_manifests_count	int	The number of manifests which were re-written by this command
added_manifests_count	int	The number of new manifest files which were written by this command

Examples

Rewrite the manifests in table **db.sample** and align manifest files with table partitioning.

```
CALL spark_catalog.system.rewrite_manifests('db.sample');
```

Rewrite the manifests in table **db.sample** and disable Spark caching (to avoid memory issues on the executor).

```
CALL spark_catalog.system.rewrite_manifests('db.sample', false);
```

rewrite_position_delete_files

Iceberg can rewrite positional delete files, which serves two main purposes:

- Lightweight compression: Merges mall locational delete files into larger file to reduce the amount of metadata stored in the manifest files and lower the overhead of opening small delete files.
- Removing dangling deletes: Filters out locational delete records that point to inactive data files. After **rewrite_data_files** is executed, locational delete records pointing to rewritten data files may not be marked for removal and may still be tracked by the active snapshot metadata of the table. This phenomenon is called "dangling delete."

How to use

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table to be updated
options	No	map	Configuration options used for this procedure
where	No	string	Filter criteria of the string type used to filter files

Configuration options

Parameter	Default Value	Description
max-concurrent-file-group-rewrites	5	The maximum number of file groups to be simultaneously rewritten
partial-progress.enabled	false	Whether to enable committing some file groups prior to the entire rewrite completing
partial-progress.max-commits	10	The maximum number of commits allowed for this rewrite operation if partial progress is enabled
rewrite-job-order	none	The order of forcibly rewriting jobs. The options are as follows: • **bytes-asc**: Rewrites the file groups with the least data volume first. • **bytes-desc**: Rewrites the file group with the largest data volume first. • **files-asc**: Rewrites the file groups with the least files first. • **files-desc**: Rewrites the file groups with the most files first. • **none**: Rewrites jobs in the order they were planned (no specific ordering).

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Default Value	Description
target-file-size-bytes	67108864 (64 MB; the default value inherits from the table property **write.delete.target-file-size-bytes**)	Target size of the output file
min-file-size-bytes	75% of target file size	Files below this threshold will be included in the rewrite scope, regardless of other criteria
max-file-size-bytes	180% of target file size	Files above this threshold will be included in the rewrite scope, regardless of other criteria
min-input-files	5	Any file group with more files than this value will be rewritten, regardless of other criteria
rewrite-all	false	Whether to forcibly rewrite all provided files and overwrite other criteria
max-file-group-size-bytes	107374182400 (100 GB)	Largest amount of data that can be rewritten in a single file group The entire rewrite operation is broken down into pieces based on partitioning and within partitions based on size into file groups. This prevents large partitions from failing to be rewritten due to resource constraints of the cluster.
max-files-to-rewrite	null	The maximum number of files that will be rewritten. If this option is not specified, all files that meet the criteria will be rewritten.

Output result

Output Parameter	Type	Description
rewritten_delete_files_count	int	Number of old positional delete files removed by this command

 Copyright © Huawei Technologies Co., Ltd.

Output Parameter	Type	Description
added_delete_files_count	int	Number of new positional delete files added by this command
rewritten_bytes_count	long	Total number of bytes in all old positional delete files removed by this command
added_bytes_count	long	Total number of bytes in all new positional delete files added by this command

Examples

Rewrite the positional delete files of table **db.sample**: Select the files to be rewritten based on the default filter criteria, generate new files that meet the **target-file-size-bytes target** size, and automatically remove dangling delete records during the rewriting process.

```
CALL spark_catalog.system.rewrite_position_delete_files('db.sample');
```

Rewrite all positional delete files of table **db.sample**: Overwrite the filter criteria, generate new files that meet the target size, and remove dangling delete records.

```
CALL spark_catalog.system.rewrite_position_delete_files(table => 'db.sample', options => map('rewrite-all', 'true'));
```

Rewrite positional delete files of table **db.sample** by file quantity: Rewrite only the positional delete files in partitions where the number of files to be rewritten by size is greater than or equal to 2, and remove dangling delete records.

```
CALL spark_catalog.system.rewrite_position_delete_files(table => 'db.sample', options => map('min-input-files','2'));
```

Migrating Tables

Snapshot and migration stored procedures help test existing Hive or Spark tables and migrate them to Iceberg tables.

snapshot

The snapshot operation creates a lightweight temporary copy of the source table for testing purposes, without modifying the source table itself.

The newly created snapshot table can be modified or written to without affecting the source table. However, the snapshot table references the raw data files of the source table. When an insert or overwrite operation is performed on the snapshot table, the newly generated files are stored in the specified location of the snapshot table, not in the storage location of the source table.

After the snapshot table test is complete, you can run the **DROP TABLE** statement to clear the table.

How to use

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
source_table	Yes	string	The name of the source table for which a snapshot is to be generated
table	Yes	string	The name of the new Iceberg snapshot table to be created
location	No	string	The location where the new snapshot table is stored. By default, the directory is automatically allocated. Currently, this parameter cannot be set in the AI DataLake table migration scenario.
properties	No	map	Properties (such as table-level configurations) to be added to the new snapshot table
parallelism	No	int	The number of threads used to read files. The default value is **1**.

Output result

Output Parameter	Type	Description
imported_files_count	long	The number of files added to the new snapshot table (that is, the number of data files referenced from the source table)

Examples

Create an isolated Iceberg snapshot table: Reference the source table **db.sample**, set the new table name to **db.snap**, and store the table in the default location allocated for **db.snap**.

```
CALL spark_catalog.system.snapshot('db.sample', 'db.snap');
```

migrate

Replace an existing table with an Iceberg table and load the data files of the source table.

Copy the table structure, partition configurations, table properties, and storage location of the source table to the new Iceberg table.

If there is an unsupported file format in any partition of the source table, the migration will fail. Currently, the Avro, Parquet, and ORC file formats are

 Copyright © Huawei Technologies Co., Ltd.

supported. The existing data files of the source table will be added to the metadata of the Iceberg table. These data files can be read through the name-ID mapping generated based on the source table structure.

To keep the source table intact during testing, you can use the snapshot process to create a temporary table to share the data files and structure of the source table.

By default, the source table is retained and named **table_BACKUP_**.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the source table to be migrated
properties	No	map	Properties to be added to the new Iceberg table (which will overwrite the properties with the same name in the source table)
drop_backup	No	boolean	If this parameter is set to **true**, the source table will not be retained as a backup. The default value is **false**.
backup_table_name	No	string	The name of the retained backup source table (the default value is the *Source table name*_**BACKUP**_)
parallelism	No	int	The number of threads used to read files. The default value is **1**.

Output result

Output Parameter	Type	Description
migrated_files_count	long	The number of data files migrated to the new Iceberg table

Examples

Migrate tables in the default Spark directory and add properties: Migrate **spark_catalog.db.sample** to an Iceberg table and add the **'foo'='bar'** property to the new table.

```
CALL spark_catalog.system.migrate('spark_catalog.db.sample', map('foo', 'bar'));
```

Conduct a basic migration (without adding additional properties): Migrate the **db.sample** file in the current directory to the Iceberg table without adding additional properties.

 Copyright © Huawei Technologies Co., Ltd.

```
CALL spark_catalog.system.migrate('db.sample');
```

add_files

The add_files command is used to add files in a Hive table or file-based table to a specified Iceberg table. Unlike migration or snapshot, the **add_files** command can import files from a specific partition and does not create a new Iceberg table. This command generates metadata for the files to be added, but does not move the files themselves. This process does not analyze the schema of the files to verify whether it matches the schema of the Iceberg table. After the operation is complete, the Iceberg table treats these files as part of its own managed file set. This means that when expired snapshots are executed later, these added files can be physically deleted. The migrate or snapshot method is preferentially recommended when applicable.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the target Iceberg table to which files are to be added
source_table	Yes	string	The source table of the file (which can be a table registered with Hive/Spark or a file path in the format of **file_format.path**, for example, **parquet./data/path**)
partition_filter	No	map	Mapping used to filter partitions in the source table (only files from partitions that meet the filter criteria are imported)
check_duplicate_files	No	boolean	Whether to prevent files that already exist in the target table from being added again (default: **true**)
parallelism	No	int	The number of threads used to read files. The default value is **1**.

Output result

Output Parameter	Type	Description
added_files_count	long	The number of files that were successfully added by this command
changed_partition_count	long	Number of affected partitions (if countable)

 Copyright © Huawei Technologies Co., Ltd.

Examples

Import files from a specified partition: Add the files from the partition where **part_col_1** equals A in the Hive/Spark table **db.src_tbl** registered in the session directory to the Iceberg table **db.tbl**.

```
CALL spark_catalog.system.add_files( table => 'db.tbl', source_table => 'db.src_tbl', partition_filter => map('part_col_1', 'A') );
```

Import all files from a file path: Add all files from the Parquet table located at **path/to/table** to the Iceberg table **db.tbl** (without filtering partitions).

```
CALL spark_catalog.system.add_files( table => 'db.tbl',   source_table => '`parquet`.`path/to/table`' );
```

register_table

The register_table command creates a directory entry for the existing **metadata.json** file that has not been associated with a directory identifier. By performing this operation, an independent Iceberg metadata file can be registered as a formal table in a specified directory so that it can be identified and managed by the directory.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table to be registered
metadata_file	Yes	string	The path of the **metadata.json** file to be registered

Output result

Output Parameter	Type	Description
current_snapshot_id	long	The current snapshot ID of the newly registered Iceberg table
total_records_count	long	The total number of records in the newly registered Iceberg table
total_data_files_count	long	The total number of data files in the newly registered Iceberg table

Examples

In the **spark_catalog** directory, register the **metadata.json** file located at **path/to/metadata/file.json** as the table **db.tbl**.

```
CALL spark_catalog.system.register_table( table => 'db.tbl', metadata_file => 'path/to/metadata/file.json' );
```

 Copyright © Huawei Technologies Co., Ltd.

Metadata

ancestors_of

The **ancestors_of** command queries the active snapshot IDs of the parent snapshots for a specified snapshot, returning the chain of ancestor snapshots.

How to use

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table for which to query the ancestor snapshots
snapshot_id	No	long	The ID of the snapshot whose parent snapshot active ID is to be queried

Output result

Output Parameter	Type	Description
snapshot_id	long	The ID of the ancestor snapshot
timestamp	long	The time when the ancestor snapshot was created

Examples

Query all ancestor snapshots of the current snapshot (default behavior):

```
CALL spark_catalog.system.ancestors_of('db.tbl');
```

Query all ancestor snapshots of a specified snapshot ID (using positional parameters):

```
CALL spark_catalog.system.ancestors_of('db.tbl', 1);
```

Query all ancestor snapshots of a specified snapshot ID (using named parameters, which is recommended for better readability):

```
CALL spark_catalog.system.ancestors_of(snapshot_id => 1, table => 'db.tbl');
```

Change Data Capture

create_changelog_view

The **create_changelog_view** creates a view that contains the change records of a specified table. This view can be used to track all data changes between different snapshots or timestamps of the table.

How to use

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the source table for which change records are generated
changelog_view	No	string	Name of the change view to be created. If this parameter is not specified, the default view name associated with the source table is generated.
options	No	map	A map of Spark read options, which is used to specify the time range for change records (see below for common options)
net_changes	No	boolean	Whether to output only the net changes (removing intermediate changes and retaining only the final result). The default value is **false**. When **compute_updates** is set to **true**, this parameter must be set to **false**.
compute_updates	No	boolean	Whether to calculate the pre/post update images (identifying the combination of delete and insert as an update). The default rule is as follows: If **identifier_columns** is provided, the default value is **true**. If it is not provided, the default value is **false**.
identifier_columns	No	array	A list of identifier columns, for example, **array ('id', 'name')**, used to identify update operations. If **compute_updates** is set to **true** and this parameter is not specified, the current identifier column of the table is used.

The **options** parameter can be used to specify the time range of change records using two dimensions (snapshot ID or timestamp). If this parameter is not specified, the full snapshot of the table is overwritten by default.

- **start-snapshot-id**: exclusive start snapshot ID (excluding changes in this snapshot). By default, the first snapshot (inclusive) of the table is used.
- **end-snapshot-id**: inclusive end snapshot ID (including changes to the snapshot). By default, the current snapshot of the table is used.
- **start-timestamp**: exclusive start timestamp, in milliseconds. By default, the first snapshot (inclusive) of the table is used.
- **end-timestamp**: inclusive end timestamp, in milliseconds. By default, the current snapshot of the table is used.

Output result

 Copyright © Huawei Technologies Co., Ltd.

Output Parameter	Type	Description
changelog_view	string	The name of the created change view

Examples

Create a view **tbl_changes** that contains the changes of the source table **db.tbl** between snapshot 1 (exclusive) and snapshot 2 (inclusive).

```
CALL spark_catalog.system.create_changelog_view(  table => 'db.tbl',  options => map('start-snapshot-id','1','end-snapshot-id', '2'));
```

Create a view **my_changelog_view** that contains the changes of the source table **db.tbl** between timestamps 1678335750489 (exclusive) and 1678992105265 (inclusive).

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('start-timestamp','1678335750489','end-timestamp', '1678992105265'),   changelog_view => 'my_changelog_view' );
```

Create a view that contains changes between snapshots 1 and 2, in which the combination of delete and insert is identified as an update operation based on the identifier columns **id** and **name**.

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('start-snapshot-id','1','end-snapshot-id', '2'),   identifier_columns => array('id', 'name') )
```

After the view is created, you can directly query the change records using SQL statements.

```
SELECT * FROM tbl_changes;
SELECT * FROM tbl_changes where _change_type = 'INSERT' AND id = 3 ORDER BY _change_ordinal;
```

Note that the change log view contains change data capture (CDC) metadata columns, which provide additional information about the tracked changes. These columns are as follows:

- **_change_type**: The type of the change. The value can be **INSERT**, **DELETE**, **UPDATE_BEFORE**, or **UPDATE_AFTER**.
- **_change_ordinal**: The order of the change.
- **_commit_snapshot_id**: The ID of the snapshot where the change occurs.

The following example shows that two records were inserted into the first snapshot and one record was deleted from the second snapshot.

id	name	_change_type	_change_ordinal	_commit_snapshot_id
1	Alice	INSERT	0	5390529835796506035
2	Bob	INSERT	0	5390529835796506035

 Copyright © Huawei Technologies Co., Ltd.

id	name	_change_type	_change_ordinal	_commit_snapshot_id
1	Alice	DELETE	1	8764748981452218370

Net Changes

This process removes intermediate changes between multiple snapshots and outputs only the net changes. The following is an example of creating a change log view to calculate net changes:

```
CALL spark_catalog.system.create_changelog_view(   table => 'db.tbl',   options => map('end-snapshot-id', '87647489814522183702'),   net_changes => true );
```

Since Alice was inserted in the first snapshot and deleted in the second snapshot, the preceding change log view, after net changes are enabled, contains only the following rows:

id	name	_change_type	_change_ordinal	_commit_snapshot_id
2	Bob	INSERT	0	5390529835796506035

Carry-over Rows

This procedure removes the carry-over rows by default. Carry-over rows are the results of row-level operations (MERGE, UPDATE, and DELETE) performed in copy-on-write mode. For example, there is a file containing row 1 (**id=1,name='Alice'**) and row 2 (**id=2,name='Bob'**). When row 2 is deleted in copy-on-write mode, the file needs to be deleted and row 1 needs to be retained in a new file. Although this is not an actual change to the table, the change log table still records the following two rows:

id	name	_change_type
1	Alice	DELETE
1	Alice	INSERT

You can query the SparkChangelogTable to view the carry-over rows.

```
SELECT * FROM spark_catalog.db.tbl.changes;
```

Pre/Post Update Images

If configured, this process can calculate the pre/post update images. Pre/post update images are derived from a set of delete and insert rows. Identifier columns are used to determine whether an insert record and a delete record refer to the

 Copyright © Huawei Technologies Co., Ltd.

same row. If the values of the identifier columns are the same, the two records are regarded as the same row's before and after states. You can set identifier columns in the table schema or pass them as procedure parameters.

The following example demonstrates the process of calculating pre/post update images using the identifier column **id**. Rows with the same identifier column value for delete and insert operations are considered as a single update operation. Specifically, assume there are the following two rows of records:

id	name	_change_type
3	Robert	DELETE
3	Dan	INSERT

In this case, the process marks the row before the update as an UPDATE_BEFORE image and the row after the update as an UPDATE_AFTER image, resulting in the following pre/post update images:

id	name	_change_type
3	Robert	UPDATE_BEFORE
3	Dan	UPDATE_AFTER

Table Statistics

compute_table_stats

This procedure is used to calculate the number of distinct values (NDV) statistics for a specific table. By default, statistics are calculated for all columns using the current snapshot of the table. This procedure can be optionally configured to calculate statistics for a specific snapshot and/or a subset of columns.

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table
snapshot_id	No	string	The ID of the snapshot for which to collect statistics
columns	No	array	The column for which to collect statistics

Output result

 Copyright © Huawei Technologies Co., Ltd.

Output Parameter	Type	Description
statistics_file	string	The path of the statistics file created by this command

Examples

Collect statistics for the latest snapshot of the **my_table** table.

```
CALL catalog_name.system.compute_table_stats('my_table');
```

Collect statistics for the snapshot whose ID is **snap1** in the **my_table** table.

```
CALL catalog_name.system.compute_table_stats(table => 'my_table', snapshot_id => 'snap1' );
```

Collect statistics for columns **col1** and **col2** of the snapshot whose ID is **snap1** in the **my_table** table.

```
CALL catalog_name.system.compute_table_stats(table => 'my_table', snapshot_id => 'snap1', columns => array('col1', 'col2'));
```

Partition Statistics

compute_partition_stats

This process incrementally calculates partition statistics from the last snapshot that has a PartitionStatisticsFile to a given snapshot (or the current snapshot if none is specified) and writes the results to the PartitionStatisticsFile. If the previous PartitionStatisticsFile does not exist, a full calculation is performed. It also registers the PartitionStatisticsFile in the table metadata.

Parameter	Mandatory (Yes/No)	Type	Description
table	Yes	string	The name of the table
snapshot_id	No	string	The ID of the snapshot for which to calculate partition statistics. The default value is the current snapshot ID.

Output result

 Copyright © Huawei Technologies Co., Ltd.

Output Parameter	Type	Description
partition_statistics_file	string	The path of the partition statistics file created by this command

Examples

Collect partition statistics for the latest snapshot of the **my_table** table.

```
CALL catalog_name.system.compute_partition_stats('my_table');
```

Collect partition statistics for the snapshot whose ID is **snap1** in the **my_table** table.

```
CALL catalog_name.system.compute_partition_stats(table => 'my_table', snapshot_id => 'snap1');
```

Table Replication

The **rewrite_table_path** procedure is used to prepare for replication of the Iceberg table to another location.

It creates a staged replica of the Iceberg table metadata file, where each absolute path source prefix is replaced with the specified target prefix. This can serve as the starting point for fully or incrementally replicating the Iceberg table to a new location.

This procedure only stages the rewritten metadata file and prepares the list of files to be replicated. The actual file replication is not included in this procedure.

Parameter	Mandatory (Yes/No)	Default Value	Type	Description
table	Yes	-	string	The name of the table
source_prefix	Yes	-	string	The existing prefix to be replaced
target_prefix	Yes	-	string	The target prefix for **source_prefix**
start_version	No	The first **metadata.json** in the table metadata log	string	The name or path of the first **metadata.json** file to be rewritten in chronological order
end_version	No	The latest **metadata.json** in the table metadata log	string	The name or path of the last **metadata.json** file to be rewritten in chronological order

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Type	Description
staging_location	No	The new directory in the table metadata directory	string	The output location of the newly rewritten metadata file

Operation modes

- **Full rewrite**: Rewrites all accessible metadata files (including **metadata.json**, manifest lists, manifests, and position delete files) and returns all accessible files in **file_list_location**. This is the default operation mode of this procedure.
- **Incremental rewrite** (optional): Limits the scope to incremental rewrite by providing **start_version** and **end_version**. An incremental rewrite rewrites only the metadata files added between **start_version** and **end_version**, and returns only the files added within this range in **file_list_location**.

Output result

Output Parameter	Type	Description
latest_version	string	The name of the latest metadata file rewritten by this procedure
file_list_location	string	The path of the CSV file that contains the mapping from the source path to the target path.

File list

This file contains the replication plan for all files added to the table between **start_version** and **end_version**.

For each file, it specifies:

- **Source path**: The original file path in the table, or the staging location if the file has been rewritten.
- **Target path**: The path with the replaced prefix.

The following example shows the replication plans for three files:

```
sourcepath/datafile1.parquet,targetpath/datafile1.parquet
sourcepath/datafile2.parquet,targetpath/datafile2.parquet
stagingpath/manifest.avro,targetpath/manifest.avro
```

Examples

 Copyright © Huawei Technologies Co., Ltd.

This example completely rewrites the metadata path of the table **my_table** from the source location in OBS to the target location in OBS. It generates a new set of metadata in the default staging location under the table metadata directory.

```
CALL catalog_name.system.rewrite_table_path(
    table => 'db.my_table',
    source_prefix => 'obs://bucket/path/to/source_table',
    target_prefix => 'obs://bucket/prefix/db.db/my_table' );
```

This example incrementally rewrites the metadata path of the table **my_table** between metadata versions **v2.metadata.json** and **v20.metadata.json**, and writes the new metadata file to the specified staging location.

```
CALL catalog_name.system.rewrite_table_path(
    table => 'db.my_table',
    source_prefix => 'obs://bucketOne/prefix/db.db/my_table',
    target_prefix => 'obs://bucketTwo/prefix/db.db/my_table',
    start_version => 'v2.metadata.json',
    end_version => 'v20.metadata.json',
    staging_location => 'obs://bucketStaging/my_table' );
```

After the rewriting is complete, a third-party tool (such as DistCp) can replicate the newly created metadata files and data files to the target location.

Finally, you can register the replicated table with the catalog using the registry method.

1.3.7 Using Spark to Read and Write Iceberg in Streaming Mode

Iceberg uses Apache Spark's DataSourceV2 API for data source and catalog implementations. Spark DSv2 is an evolving API, and its support varies across different Spark versions.

Streaming Read

Iceberg can read incremental data from a specified historical timestamp in Spark structured stream processing tasks.

```
val df = spark.readStream
    .format("iceberg")
    .option("stream-from-timestamp", Long.toString(streamStartTimestamp))
    .load("database.table_name")
```

Iceberg supports reading data only from append snapshots. Overwrite snapshots cannot be processed and an exception is thrown by default. In this case, you can set **streaming-skip-overwrite-snapshots** to **true** to ignore the overwrite operation. Similarly, an exception is thrown by default for delete snapshots. You can set **streaming-skip-delete-snapshots** to **true** to ignore the delete operation.

Streaming Write

To write the results of a streaming query to an Iceberg table, use DataStreamWriter:

```
data.writeStream
    .format("iceberg")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(1, TimeUnit.MINUTES))
    .option("checkpointLocation", checkpointPath)
    .toTable("database.table_name")
```

 Copyright © Huawei Technologies Co., Ltd.

Iceberg supports the following output modes:

- **append**: Appends the rows of each micro-batch to a table.
- **complete**: Replaces the content of the entire table with the content of each micro-batch.

Before starting a streaming query, ensure that a table has been created. For details about how to create an Iceberg table, see the SQL table creation documentation.

Iceberg does not support an experimental continuous processing mode because it does not provide an interface for committing output.

Partitioned Tables

Iceberg requires that the data of each task be sorted by partition before being written. In Spark, tasks are divided by Spark partition. Explicit sorting is recommended for batch queries, but this approach may increase latency because repartitioning and sorting are heavyweight operations for streaming workloads. In this case, you can enable the fanout writer to avoid unnecessary delay.

```
data.writeStream
    .format("iceberg")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(1, TimeUnit.MINUTES))
    .option("fanout-enabled", "true")
    .option("checkpointLocation", checkpointPath)
    .toTable("database.table_name")
```

The fanout writer opens a file for each partition value and does not close these files until the write task is complete. Avoid using the fanout writer in batch write operations, as explicitly sorting output rows is less costly for batch workloads.

Maintaining Streaming Tables

Streaming writes quickly create new table versions, generating a large amount of table metadata to track these versions. It is highly recommended to maintain metadata by adjusting the commit frequency, expiring old snapshots, and automatically cleaning up metadata files.

Adjusting the Commit Frequency

High-frequency commits generate data files, manifest files, and snapshots, increasing maintenance overhead. It is recommended to set at least a 1-minute interval for commits and increase it as needed.

Expiring Old Snapshots

Each batch written to the table generates a new snapshot. Iceberg tracks snapshots in the table metadata until they expire. Frequent commits may lead to snapshot stacking, so you are strongly advised to periodically maintain the tables to which streaming queries write. Snapshot expiration is the process of removing metadata and any data files that are no longer needed. By default, this process expires snapshots older than five days.

 Copyright © Huawei Technologies Co., Ltd.

Merging Small Files

Streaming writes typically involve small amounts of data, which causes table metadata to track a large number of small files. Merging small files into larger ones reduces the metadata required by the table and improves query efficiency. Iceberg and Spark provide the **rewrite_data_files** stored procedure for this purpose.

Rewriting Manifest Files

To optimize write latency for streaming workloads, Iceberg can write new snapshots in "fast append" mode without automatically merging manifest files. This may result in a large number of small manifest files. Iceberg can rewrite the number of manifest files to improve query performance. To achieve this, Iceberg and Spark provide the **rewrite_manifests** stored procedure.

1.3.8 Liquid Clustering Usage

Liquid Clustering Configuration

Liquid Clustering is configured using Iceberg table properties.

```
CREATE TABLE events (
    event_id BIGINT,
    user_id BIGINT,
    event_type STRING,
    event_time TIMESTAMP,
    payload STRING
)
USING iceberg
TBLPROPERTIES (
    'format-version' = '2',
    'write.liquid-clustering.columns' = 'user_id,event_time'
);
```

Specifying Custom Cube Size

```
CREATE TABLE logs (
    log_id BIGINT,
    service_name STRING,
    log_level STRING,
    message STRING,
    timestamp TIMESTAMP
)
USING iceberg
TBLPROPERTIES (
    'format-version' = '2',
    'write.liquid-clustering.columns' = 'service_name,timestamp',
    'write.liquid-clustering.target-cube-size-bytes' = '21474836480',
    'write.liquid-clustering.min-cube-size-bytes' = '5368709120'
);
```

Note: Set **target-cube-size-bytes** to 20 GB (21,474,836,480 bytes) and **min-cube-size-bytes** to 5 GB (5,368,709,120 bytes).

Modifying Liquid Clustering Configuration

```
-- Modify the clustering columns (only adding is supported; deleting existing clustering columns is not supported).
ALTER TABLE events SET TBLPROPERTIES ('write.liquid-clustering.columns' =
'user_id,event_time,event_type');
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Adjust the target Cube size.
ALTER TABLE events SET TBLPROPERTIES ('write.liquid-clustering.target-cube-size-bytes' = '32212254720');

-- Adjust the minimum Cube threshold.
ALTER TABLE events SET TBLPROPERTIES ('write.liquid-clustering.min-cube-size-bytes' = '10737418240');
```

Triggering Cube Rewriting

```
-- Rewrite all Cubes marked as Partial.
CALL spark_catalog.system.rewrite_data_files(
    table => 'db.events',
    options => map('rewrite-all', 'false')
);

-- Rewrite all Cubes (including healthy Cubes).
CALL spark_catalog.system.rewrite_data_files(
    table => 'db.events',
    options => map('rewrite-all', 'true')
);
```

1.4 Iceberg on Flink

1.4.1 Preparations Before Using Iceberg

Iceberg is an open data lake table format that supports features such as ACID transactions, schema evolution, and time travel. AI DataLake Flink integrates with Huawei Cloud LakeFormation through Iceberg Catalog to implement unified metadata management and direct access to Iceberg tables. Before using Iceberg, you need to complete preparatory steps, including creating a LakeFormation instance, planning metadata, and configuring the workspace-level catalog.

Preparations

- **A LakeFormation instance has been created with metadata planned.**

  Ensure that LakeFormation catalog is located in the same region as the AI Data Lake workspace. Otherwise, they cannot be associated in subsequent processes.
- A workspace-level LakeFormation has been configured in AI DataLake.

1.4.2 Managing Iceberg Catalogs

After configuring an Iceberg catalog for an AI DataLake Flink job, you can use AI DataLake Flink to directly access Iceberg tables in Huawei Cloud LakeFormation. This section describes how to create an Iceberg catalog in an AI DataLake Flink job and manage Iceberg databases and tables.

Procedure: Use a tool such as Postman to call the API for **Creating a FlinkJob Job** to submit a Flink job.

Creating an Iceberg LakeFormation Catalog

```
CREATE CATALOG lf_iceberg_catalog WITH (
  'type' = 'iceberg',
  'catalog-type' = 'lakeformation'
);
```

The following table lists the parameters.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Mandatory (Yes/No)	Remarks
type	Catalog type.	Yes	The value is fixed to **iceberg**.
catalog-type	Catalog implementation type.	Yes	The value is fixed to **lakeformation**.
lakecat.catalogname.default	Name of the catalog to be accessed in LakeFormation. The name must be the same as the name of the catalog created in LakeFormation.	No	The default value is the catalog name on the Flink engine, that is, **lf_iceberg_catalog**.

Managing Iceberg Databases

- **Creating a database**

  After creating an Iceberg catalog, create the **iceberg_db** database in **lf_iceberg_catalog**.

  ```
  CREATE DATABASE IF NOT EXISTS lf_iceberg_catalog.iceberg_db
  WITH (
   'location' = 'obs://bucket/iceberg_catalog/iceberg_db/'
   );
  ```

- **Deleting a database**

  ```
  DROP DATABASE lf_iceberg_catalog.iceberg_db; -- Only deletes databases that do not contain any tables.
  ```

Managing Iceberg Tables

- Creating a table

  Create a partitioned table in the **my_db** database under the **iceberg-catalog** catalog. The partition key is **dt**, and the primary keys are **dt**, **shop_id**, and **user_id**. The sample code is as follows:

  ```
  CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table (
    dt STRING,
    shop_id BIGINT,
    user_id BIGINT,
    num_orders INT,
    total_amount INT,
    PRIMARY KEY (dt, shop_id, user_id) NOT ENFORCED
  ) PARTITIONED BY (dt);
  ```

- Modifying a table

 Copyright © Huawei Technologies Co., Ltd.

Operation	Code Example

Temporarily modifying table parameters	You can temporarily modify table parameters by adding an SQL hint after the table name when writing data to the table. These temporary modifications are only effective for the current SQL job.
	• Temporarily set **upsert-enabled** to **true** when writing to the **my_table** table.
	`INSERT INTO tableName /*+ OPTIONS('upsert-enabled'='true') */ SELECT ...;`
	• Temporarily set **monitor-interval** to **10s** when consuming the **my_table** table.
	`SELECT * FROM my_table /*+ OPTIONS('monitor-interval'='10s') */`

Renaming a table	Rename the table from **my_table** to **my_table_new**. Only EXTERNAL and LAKE tables can be renamed. `ALTER TABLE my_table RENAME TO my_table_new;`
Adding a column	Add the **c1** column of the INT type and the **c2** column of the STRING type to the end of the **my_table** table. `ALTER TABLE my_table ADD (c1 INT, c2 STRING);`
Renaming a column	Rename the **c0** column in the **my_table** table to **c1**. `ALTER TABLE my_table RENAME c0 TO c1;`
Dropping a column	Drop columns **c1** and **c2** from the **my_table** table. `ALTER TABLE my_table DROP (c1, c2);`
Modifying column comments	Change the comment of the **buy_count** column in the **my_table** table to **this is buy count**. `ALTER TABLE my_table MODIFY buy_count BIGINT COMMENT 'this is buy count';`

Modifying the column order	• Move the **col_a** column of the DOUBLE type in the **my_table** table to the beginning.
	`ALTER TABLE my_table MODIFY col_a DOUBLE FIRST;`
	• Move the **col_a** column of the DOUBLE type in the **my_table** table to the end of the **col_b** column.
	`ALTER TABLE my_table MODIFY col_a DOUBLE AFTER col_b;`

- Deleting a table
  `DROP TABLE lf_iceberg_catalog.iceberg_db.iceberg_table;`

1.4.3 Iceberg Configurations and Types

WITH Parameters

This section describes how to configure parameters for the Flink connector to read and write Iceberg tables. By configuring these parameters, you can control the connection and read/write behavior between Flink and Iceberg table formats to meet data read/write requirements for various service scenarios.

 Copyright © Huawei Technologies Co., Ltd.

- **Source parameters**

  Table 1-26 Source parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
streaming	No	Boolean	false	true, false	Specifies whether to read the Iceberg table in streaming mode. The value **true** enables streaming read, which continuously monitors new snapshots and outputs incremental data. The value **false** enables batch read, which only reads all data from the current snapshot. Choose based on the service scenario. In real-time ETL scenarios, set this parameter to **true**.
start-snapshot-id	No	Long	None	Valid snapshot ID	Specifies the start snapshot ID for streaming read. This parameter is valid only when **streaming** is set to **true**. If configured, incremental data is read from the snapshot after this one. If not configured, full data is read from the earliest available snapshot. Ensure that the snapshot ID exists and has not expired.
end-snapshot-id	No	Long	None	Valid snapshot ID	Specifies the end snapshot ID for streaming read. After the snapshot is read, the job automatically ends. This parameter is valid only when **streaming** is set to **true**. Ensure that the value of **end-snapshot-id** is greater than that of **start-snapshot-id**.
split-size	No	Long	128 MB	Positive integer (byte count)	Specifies the target size of each split during read. Iceberg splits a data file into multiple splits of this size for parallel read. Adjust the value based on the data volume and parallelism. Splits that are too small cause excessive scheduling overhead, while splits that are too large lead to data skew.

- **Sink parameters**

 Copyright © Huawei Technologies Co., Ltd.

Table 1-27 Sink parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
write-format	No	String	parquet	parquet, orc, avro	Specifies the format of the file to be written to the Iceberg table. • **parquet** indicates the columnar storage format with high compression ratio and good query performance. • **orc** indicates the Optimized Row Columnar format. • **avro** indicates the row-based storage format. Select a value based on the query engine and performance requirements.
commit.interval	No	Duration	1 min	Valid duration, for example, 30 seconds or 5 minutes.	Specifies the commit interval for streaming writes. The buffered data files are committed as a new Iceberg snapshot at the specified interval. When configuring this parameter, you need to balance data visibility and write performance. Short intervals may produce too many small files, while long intervals may result in high data latency.
write.target-file-size-bytes	No	Long	128 MB	Positive integer (byte count)	Specifies the target file size for writes. A new file is rolled out when the written data volume reaches this threshold. The configuration needs to be adjusted based on storage and query performance requirements. Small files may affect query efficiency, while large files may increase write latency.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
upsert-enabled	No	Boolean	false	true, false	Specifies whether to enable the UPSERT write mode. After enabled, data with the same primary key will be updated instead of being appended. Ensure that the primary key has been defined for the Iceberg table. Otherwise, an error will be reported during data writing (subject to the actual product support).
compression-codec	No	String	zstd	uncompressed, snappy, zstd, gzip	Specifies the compression codec for writing data files. Different compression algorithms offer different trade-offs between compression ratio and speed. **zstd** provides a good balance. Select a value based on the storage cost and query performance requirements.

Data Type Mapping

Flink	Iceberg
boolean	boolean
tinyint	integer
smallint	integer
integer	integer
bigint	long
float	float
double	double
char	string
varchar	string
string	string
binary	binary
varbinary	fixed

 Copyright © Huawei Technologies Co., Ltd.

Flink	Iceberg
decimal	decimal
date	date
time	time
timestamp	timestamp without timezone
timestamp_ltz	timestamp with timezone
array	list
map	map
multiset	map
row	struct

1.4.4 Iceberg on Flink DDL Syntax

Basic Table Creation Statement

The most basic table creation statement specifies the table name, column definitions, and table properties.

```
CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table (
  id BIGINT COMMENT 'unique id',
  data STRING NOT NULL
) WITH ('format-version'='2');
```

The following common clauses are supported:

- **PARTITION BY (column1, column2, ...)**: Configures partitioning. Note that Flink does not support hidden partitioning.
- **COMMENT 'table document'**: Sets the table description.
- **WITH ('key'='value', ...)**: Sets table configurations (stored in Iceberg table properties).

PRIMARY KEY (Primary Key Constraint)

You can declare a primary key constraint for one or more columns. The primary key must be unique and cannot contain NULL values. The primary key constraint is required for the UPSERT mode.

Table creation statement with a primary key constraint

```
CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table (
  id BIGINT COMMENT 'unique id',
  data STRING NOT NULL,
  PRIMARY KEY(id) NOT ENFORCED
) WITH ('format-version'='2');
```

PARTITIONED BY (Partitioned tables)

Use the **PARTITIONED BY** clause to create a partitioned table.

 Copyright © Huawei Technologies Co., Ltd.

Partitioned table creation statement

```
CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table (
  id BIGINT COMMENT 'unique id',
  data STRING NOT NULL
) PARTITIONED BY (data)
WITH ('format-version'='2');
```

CREATE TABLE LIKE (Replicating the Table Structure)

Create a new table with the same structure, partitions, and table properties as an existing table.

```
CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table (
  id BIGINT COMMENT 'unique id',
  data STRING
);
CREATE TABLE lf_iceberg_catalog.iceberg_db.iceberg_table_like LIKE
lf_iceberg_catalog.iceberg_db.iceberg_table;
```

1.4.5 Querying and Writing to Iceberg

Important Notes

- Checkpointing is required for streaming writes to Iceberg.
- Streaming reads are not supported for Iceberg tables that use UPSERT writes.

Code Examples

Example 1: Append data without a primary key to an Iceberg table.

```
CREATE CATALOG lf_iceberg_catalog
WITH
  (
    'type' = 'iceberg',
    'catalog-type' = 'lakeformation'
  );
CREATE TABLE IF NOT EXISTS lf_iceberg_catalog.iceberg_db.iceberg_table (
  id BIGINT,
  data STRING
)WITH('format-version' = '2');

CREATE TABLE kafka_source (id BIGINT, data STRING)
WITH
  (
    'connector' = 'kafka',
    'topic' = 'user-events',
    'properties.bootstrap.servers' = '127.0.0.1:9092',
    'properties.group.id' = 'flink-consumer-group',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'json'
  );

INSERT INTO
  lf_iceberg_catalog.iceberg_db.iceberg_table
SELECT
  *
FROM
  kafka_source
```

Example 2: Read an Iceberg table in streaming mode.

```
CREATE CATALOG lf_iceberg_catalog
WITH
  (
```

 Copyright © Huawei Technologies Co., Ltd.

```
   'type' = 'iceberg',
   'catalog-type' = 'lakeformation'
  );

CREATE TABLE sink (id BIGINT, data STRING)
WITH
  ('connector' = 'print');

INSERT INTO
  sink
SELECT
  *
FROM
  lf_iceberg_catalog.iceberg_db.iceberg_table /*+ OPTIONS('streaming'='true', 'monitor-interval'='60s') */;
```

1.5 Iceberg Development FAQs

1.5.1 How Do I Handle Concurrent Data Commit Conflicts?

When writing data concurrently to an Iceberg table, two common types of conflicts can arise: concurrent data commit conflict and optimistic lock concurrent metadata commit conflict.

This section describes the concurrent data commit conflict.

Job failures due to data conflicts may occur under the following concurrent scenarios:

- Concurrent execution of DML operations on the same partition, such as performing **Update**, **Delete**, **Insert Overwrite**, or **Insert** operations on the same partition.
- Simultaneous execution of Compaction and DML operations.

The following describes concurrent data conflicts that may be encountered by the Iceberg operation types currently supported by each engine.

- When the Insert Overwrite operation is performed, the following conflicts may occur:
  - Data file conflict: For partitioned tables, if during data conflict detection, new data files produced by other committed operations within the partitions targeted by the **Insert Overwrite** are identified, a data conflict is detected. For non-partitioned tables, if any new data files resulting from other committed operations are recognized during data conflict detection, a data conflict is flagged.
  - Delete file conflict: In the case of partitioned tables, if during data conflict detection, it is found that data files have been removed or delete files have been added within the partitions affected by the **Insert Overwrite** by other operations, this signifies a data conflict. Similarly, for non-partitioned tables, detecting the removal of data files or the addition of delete files by other operations during data conflict checks indicates a data conflict has occurred.
- When the Delete operation is performed, the following conflicts may occur:

  Data file conflict: During data conflict detection, if it is identified that the data file associated with the delete file has been deleted by another operation, a data conflict is detected.

 Copyright © Huawei Technologies Co., Ltd.

- When the Update operation is performed, the following conflicts may occur:
  - Data file conflict: During data conflict detection, if it is identified that the data file associated with the delete file has been deleted by another operation, a data conflict is detected.
  - Delete file conflict: A delete conflict is identified when the system detects that another operation has generated a new delete file for the partition targeted by the update operation during conflict detection.

Table 1-28 Concurrency conflict matrix

Current Operation/ Committed Operation	insert	insert overwrite	delete
insert	No conflict	No conflict	No conflict
insert overwrite	May conflict	May conflict	May conflict
delete	No conflict	May conflict	No conflict
update	No conflict	May conflict	May conflict

NOTE

The preceding concurrency conflict matrix indicates whether a conflict will occur upon committing the current operation if another operation initiates and completes its commit between the start and submission of the current operation.

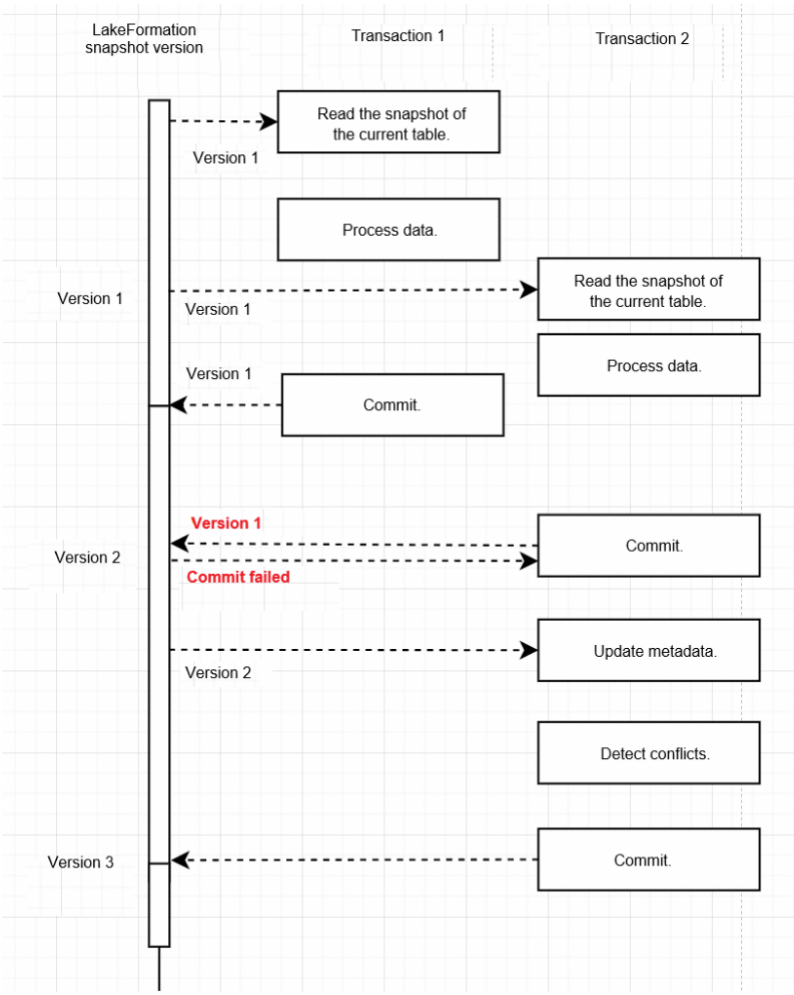
1.5.2 How Do I Handle Optimistic Lock Concurrent Commit Conflicts?

When writing data concurrently to an Iceberg table, two common types of conflicts can arise: concurrent data commit conflict and optimistic lock concurrent metadata commit conflict.

This section describes the optimistic lock concurrent metadata commit conflict.

Cause of optimistic lock concurrent metadata commit conflicts: Multiple modification jobs (including Insert, Update, Delete, MergeInto, and table service operations) are simultaneously submitted on LakeFormation, leading to concurrent commit conflicts. For example:

 Copyright © Huawei Technologies Co., Ltd.



In the preceding figure, transaction 1 and transaction 2 are two DML transactions. Transaction 1 processes and commits data based on metadata version 1 and updates the metadata version to version 2. However, Transaction 2 had already started data processing based on metadata version 1 before transaction 1 was committed. During its initial commit attempt, LakeFormation rejects the commit because the metadata has already been updated to version 2. Transaction 2 must then update the metadata and resubmit it.

In such scenarios, if the concurrency volume is extremely high or the conflict detection time for a particular transaction is excessively long, it can lead to an endless cycle of "commit conflict -> update metadata -> detect conflict -> resubmit", resulting in business conflict errors.

To address these issues, Iceberg introduces four parameters, as described in **Table 1-29**.

 Copyright © Huawei Technologies Co., Ltd.

Table 1-29 Parameter descriptions

Parameter	Description	Recommended Value for High-Concurrency Scenarios	Recommended Value for Table Service Scenarios
commit.retry.num-retries	Maximum number of retries	10	4
commit.retry.min-wait-ms	Minimum wait time after a failed commit	100	1000
commit.retry.max-wait-ms	Maximum wait time after a failed commit	10000	60000
commit.retry.total-timeout-ms	Maximum execution duration for the commit process	1800000	1800000

1.5.3 How Do I Use LakeFormation to Maintain Tables?

Internal Table Services

When creating or editing an Iceberg table, you can configure the LakeFormation table service options (only available for dedicated instances). To enable the table service for existing instances, perform the following steps:

Step 1 Log in to the**LakeFormation console**.

Step 2 Select the target LakeFormation instance from the drop-down list box on the left.

Step 3 On the **Overview** page, check whether the internal table service is enabled.

If the internal table service has been enabled for the current instance, the button for enabling the internal table service is not displayed.

Step 4 To enable the internal table service for the current instance, click the option for enabling the internal table service in the upper right corner of the page, select the operation impact, and click **OK**.

----End

NOTE

After the internal table service is enabled, you can create internal tables and functions. To disable the internal table service, you need to completely clear the internal tables. The internal table service is automatically disabled when the instance is deleted.

After the internal table service is enabled, you need to configure the table service to enable compression, snapshot, and orphaned file deletion.

 Copyright © Huawei Technologies Co., Ltd.

Compaction

Whether to combine small data files into larger, more efficient files to improve table performance.

If this option is selected, you need to set the following compression parameters:

- **Compaction Strategy**:
  - **Binpack**: Combines small files into larger files, typically targeting sizes over 100 MB, while applying any pending deletes. This compression policy is the default and recommended option for most use cases.
  - **Sort**: Organizes data of specified columns, which are sorted hierarchically during compression to improve the query performance of filter operations. This policy is recommended when query operations frequently filter specific columns. Before using this policy, you need to define the sorting order using the **sort-order** property in the column-store table.
  - **Z-Order**: Optimizes data organization by combining multiple properties into a single scalar value that can be used for sorting, enabling efficient queries across multiple dimensions. This policy is recommended when you need to query data across multiple dimensions. Before using this policy, you need to use the **sort-order** table property in the Apache Iceberg table to define the sorting order.
- **Minimum Input Files**: The minimum number of data-files to be present in a partition before compaction will actually compact files. The value ranges from **3** to **100**, with **5** as the default value.
- **Compacted File Size**: The target file size after compaction, in MB. The value ranges from **128** to **2048**, with **512** as the default value.

Snapshot Retention

Whether to optimize table storage by removing old snapshots from metadata overhead and expiring files that are no longer needed.

If this option is selected, you need to set the following snapshot retention parameters:

- **Snapshot Retention Period**: How long Iceberg retains historical snapshots before deleting them, in hours. The value ranges from **24** to **1440**, with **72** as the default value.
- **Minimum Snapshot to Retain**: The minimum number of snapshots to retain. The value ranges from **1** to **100**, with **10** as the default value.
- **Delete Associated Files**: Whether to delete files associated with the expired snapshots.

  By default, the snapshot retention strategy deletes both the metadata and data files of expired snapshots. To retain the metadata and delete only the data files, clear the **Delete Associated Files** option.
- **Execution Interval**: The interval at which the task runs, in hours. The value ranges from **1** to **168**, with **24** as the default value.

Orphan File Deletion

Whether to periodically delete orphaned files.

 Copyright © Huawei Technologies Co., Ltd.

If this option is selected, you need to set the following parameters for deleting orphaned files:

- **Orphan File Retention Period**: The retention period for orphan files, in hours. The value ranges from **24** to **1440**, with **72** as the default value.
- **Execution Interval**: The interval at which the task runs, in hours. The value ranges from **1** to **168**, with **24** as the default value.

NOTE

- The orphan file deletion strategy will delete files identified as orphans, which are files that are not referenced by Iceberg metadata.
- To ensure no files are mistakenly considered as orphaned files and deleted, make sure that the provided table location and any sub-paths do not overlap with or contain any other table or non-table data. The deletion operation cannot be undone.

1.5.4 How Do I Use an Engine to Maintain Tables?

Suggestion 1: Periodically Compressing Data Files

Reason: Too many small files will result in high metadata overhead and file opening costs, affecting query performance.

Method:

```
-- Default binpack policy (combining small files)
CALL spark_catalog.system.rewrite_data_files('db.table');
-- Sorting policy (sorting by specified columns)
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   strategy => 'sort',
   sort_order => 'id DESC NULLS LAST, name ASC NULLS FIRST'
 );
-- Z-Order optimization (multi-column interleaved sorting to improve multi-column query performance)
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   strategy => 'sort',
   sort_order => 'zorder(c1,c2,c3)'
 );
-- Enable partial progress and clean up dangling deletes.
CALL spark_catalog.system.rewrite_data_files(
   table => 'db.table',
   options => map(
     'min-input-files', '2',
     'remove-dangling-deletes', 'true',
     'partial-progress.enabled', 'true'
   )
);
```

Key parameters:

- **target-file-size-bytes**: the target file size (512 MB by default)
- **min-input-files**: the minimum number of files that trigger rewriting (5 by default)
- **rewrite-job-order**: the sorting policy (bytes-asc/desc, files-asc/desc)

Aura implementation:

In Aura, the OPTIMIZE syntax is used to rewrite specified data in a target table.

```
OPTIMIZE table_name REWRITE DATA
    [ WITH OPTIONS (option_key = option_value [, ...]) ]
```

 Copyright © Huawei Technologies Co., Ltd.

```
  [ WHERE condition ]
  [ ORDER BY target_list [ ASC | DESC ] ]
```

Parameters:

- **table_name**: the name of the target table. Enter an existing table name.
- **condition**: an expression that returns a boolean value, used to determine which files need to be rewritten.
- **target_list**: sort keys for data rewriting.
- **option_key**, **option_value**: configuration options for data rewriting.

Suggestion 2: Periodically Rewriting Position Delete Files

Reason: Compress small position delete files and clear dangling deletes (delete records that point to non-existent data files).

Method:

```
-- Compress and clear dangling deletes.
CALL spark_catalog.system.rewrite_position_delete_files('db.table');
-- Forcibly rewrite all files.
CALL spark_catalog.system.rewrite_position_delete_files(
   table => 'db.table',
   options => map('rewrite-all', 'true')
);
```

Suggestion 3: Periodically Clearing Expired Snapshots

Reason: Each write, update, deletion, upsert, or compression operation generates a new snapshot. Failure to clear these snapshots will cause continuous storage consumption.

Method:

```
-- Delete snapshots that have been stored for more than seven days and retain the latest 100 snapshots.
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   older_than => CURRENT_TIMESTAMP - INTERVAL '7' DAYS,
   retain_last => 100,
   stream_results => true
  -- Prevent driver OOM caused by a large amount of data.
  -- Delete specific snapshots.
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   snapshot_ids => ARRAY(123)
 );
  -- Delete unused metadata during the cleanup.
CALL spark_catalog.system.expire_snapshots(
   table => 'db.table',
   older_than => TIMESTAMP '2024-01-01 00:00:00',
   clean_expired_metadata => true
 );
```

Notes:

- By default, branches and tags never expire. To configure their validity periods, set the table property **history.expire.max-ref-age-ms**.
- Set **stream_results** to **true** to prevent driver OOM when there is a large amount of data.

Aura implementation: Refer to the **iceberg_expire_snapshots** table service. For details about the syntax, see **Iceberg Table Service Functions**.

 Copyright © Huawei Technologies Co., Ltd.

Suggestion 4: Periodically Deleting Orphaned Files

Reason: Delete orphaned files that are no longer referenced by any metadata file to free up storage space.

Method:

```
-- Check the files to be cleared in the trial run. (You are advised to perform this operation in the production environment first.)
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  dry_run => true
);
-- Perform the actual cleanup (delete orphaned files that have been stored for more than seven days).
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  older_than => CURRENT_TIMESTAMP - INTERVAL '7' DAYS
);
-- Delete orphaned files from a specific directory.
CALL spark_catalog.system.remove_orphan_files(
  table => 'db.table',
  location => 'table_location/data'
);
```

Aura implementation: Refer to the **iceberg_remove_orphanfiles** table service. For details about the syntax, see **Iceberg Table Service Functions**.

Suggestion 5: Periodically Rewriting Manifest Files

Reason: Sort data files in the manifest by partition spec fields to optimize the query planning performance.

Method:

```
-- Rewrite the manifest.
CALL spark_catalog.system.rewrite_manifests('db.table');
-- Specify the partition spec.
CALL spark_catalog.system.rewrite_manifests(
   table => 'db.table',
   spec_id => 1
);
```

Aura implementation: Refer to the **iceberg_rewrite_manifests** table service. For details about the syntax, see **Iceberg Table Service Functions**.

Suggestion 6: Creating Tags for Important Versions

Reason: Facilitate quick rollback to a specific version to avoid irrecoverable issues caused by misoperations.

Method:

```
-- Switch to a specified snapshot.
CALL spark_catalog.system.set_current_snapshot(
   table => 'db.table',
   snapshot_id => 123
);
-- Switch to a specified tag.
CALL spark_catalog.system.set_current_snapshot(
   table => 'db.table',
   ref => 'my_tag'
);
-- Roll back to a specified snapshot.
CALL spark_catalog.system.rollback_to_snapshot('db.table', 1);
-- Roll back by time.
```

 Copyright © Huawei Technologies Co., Ltd.

```
CALL spark_catalog.system.rollback_to_timestamp(
  'db.table',
  TIMESTAMP '2024-06-30 00:00:00.000'
);
```

Aura implementation: Refer to the **iceberg_set_current_snapshot**, **iceberg_rollback_to_snapshot** and **iceberg_rollback_to_timestamp** table services. For details about the syntax, see **Iceberg Table Service Functions**.

Suggestion 7: Using a Snapshot Table For Testing Before Migration

Reason: Creating a lightweight temporary copy for testing does not affect the source table and helps avoid migration risks.

Method:

```
-- Create a snapshot table.
CALL spark_catalog.system.snapshot('db.source_table', 'db.snapshot_table');
```

Note: The snapshot table is not the sole owner of data files and cannot physically delete files using **expire_snapshots**.

Suggestion 8: Performing Formal Migration Using migrate

Reason: Replace existing Hive/Spark tables with Iceberg tables and retain the original table as a backup.

Method:

```
-- Migrate and add properties.
CALL spark_catalog.system.migrate(
  'db.old_table',
  map('foo', 'bar')
);
-- Do not retain backups.
CALL spark_catalog.system.migrate(
  table => 'db.old_table',
  drop_backup => true
);
```

Suggestion 9: Calculating Table Statistics

Reason: Collect the NDV statistics of columns to optimize the query plan.

Method:

```
-- Calculate statistics of the entire table.
CALL spark_catalog.system.compute_table_stats('db.table');
-- Specify the snapshot and column.
CALL spark_catalog.system.compute_table_stats(
  table => 'db.table',
  snapshot_id => 'snap1',
  columns => array('col1', 'col2')
);
```

Suggestion 10: Calculating Partition Statistics

Reason: Incrementally collect partition statistics to improve the partition pruning efficiency.

Method:

```
-- Calculate partition statistics.
CALL spark_catalog.system.compute_partition_stats('db.table');
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Specify a snapshot.
CALL spark_catalog.system.compute_partition_stats(
   table => 'db.table',
   snapshot_id => 'snap1'
);
```

 Copyright © Huawei Technologies Co., Ltd.

2 Lance Development Guide

2.1 Introduction to Lance

2.1.1 Lance Functions

Lance is a high-performance columnar data storage format built on the Apache Arrow ecosystem. Developed primarily by the LanceDB team, it aims to achieve almost the same data access performance as the Arrow in-memory format. It also provides transaction management, indexing, and incremental update. Lance is suitable not only for offline batch processing and analysis, but also for online query, vector search, and AI applications. It is a new storage format designed for AI and data lake convergence.

Key Features

- Lance's underlying data structures are highly compatible with Apache Arrow, employing a standardized columnar storage approach that supports primitive types, complex nested types, and vector data structures. Once loaded into memory, data can be directly processed by Arrow compute engines, eliminating frequent data conversion and copying overhead.
- Lance incorporates a multi-version concurrency control (MVCC) mechanism that supports data snapshots, transaction commits, and rollback operations. This enables strict read-write isolation and version rollback, ensuring data consistency and reliability.
- Lance is explicitly optimized for high-dimensional vector data, supporting efficient storage and indexing of vector fields. It supports approximate nearest neighbor (ANN) search and semantic retrieval use cases, providing a unified data foundation for AI systems.
- Lance supports continuous append writes and incremental update operations. Background compaction and merging continuously optimize the underlying storage layout to maximize query efficiency.
- Core components of Lance are implemented in Rust, with official bindings available for Python, Rust, and other languages, facilitating seamless integration and deployment across diverse system architectures.

 Copyright © Huawei Technologies Co., Ltd.

- Lance supports both local file systems and object storage. It features optimized mechanisms for range reads, lazy loading, and metadata caching designed specifically for large-scale data management in cloud and data lake environments.

Key Advantages

- **Near-memory speed**: Leveraging the Arrow in-memory format and a zero-copy mechanism, Lance delivers near-memory-level throughput for operations like data scanning, filtering, projection, and vector retrieval, significantly outperforming traditional columnar formats.
- **Full mutability support**: Unlike Parquet, which is structurally optimized for append-only data, Lance natively allows updating, deleting, and merging existing records, making it ideal for dynamically changing datasets.
- **Unified multimodal storage**: Lance can store both structured fields and vector embeddings within the same table. This enables unified management and processing of multimodal data, supporting applications such as recommendation systems, search engines, and bioinformatics.
- **Extensible integration interfaces**: Lance provides low-level interfaces for storage and query engines, facilitating deep integration with database systems, distributed computing frameworks, and machine learning platforms while reducing maintenance costs.
- **Simplified deployment**: Lance organizes data strictly as a set of files without deploying a standalone service instance. This lightweight architecture enables rapid deployment across both research and production environments.

Table Organizational Structure

- **Overview of Lance table format**: The Lance table format uniformly manages structured data and vector data. Built upon the Apache Arrow data model, it employs a tiered storage architecture and metadata management layer to achieve efficient storage, access, and version control. Logically, a Lance table functions as a standard two-dimensional relational table; physically, it is composed of multiple data fragments and metadata files.
- **Logical table structure**: At the logical level, a Lance table adopts the standard relational model consisting of rows and columns, where each row represents a complete record and columns map to explicit data types. The table structure is defined by a schema that describes field names, types, and nested relationships. Lance supports a broad range of data types, including integers, floating-point numbers, strings, binary data, lists, structs, and high-dimensional vectors.
- **Physical storage architecture**: At the physical level, Lance employs a segmented architecture that divides data into independent, manageable "fragments." Within each fragment, data is stored columnar-wise without the restrictive row-group boundaries found in Parquet. Different columns are organized as contiguous blocks to facilitate efficient scanning and vectorized computation. Each data fragment maps to independent data files, supplemented by column statistics, offset indexes, and checksum information to accelerate queries and ensure data integrity.
- **Schema and metadata management**: Lance maintains table structure information, data versioning, and fragment index relationships through centralized metadata files. The schema serves as the foundation for table

 Copyright © Huawei Technologies Co., Ltd.

access and computation. Lance supports schema evolution, allowing for field addition, deletion, and type extension while maintaining backward compatibility for historical data through version control.

- **Version control and snapshot mechanism**: Lance implements an MVCC mechanism that generates a new snapshot version for each write or update operation. Each version corresponds to a complete set of data fragment references, enabling persistent storage of historical states. Based on this snapshot mechanism, the system supports historical data queries, data rollback, and point-in-time recovery, significantly enhancing data security and reliability.

Functions

- **Powerful hybrid search**

  Lance supports powerful hybrid search, integrating vector similarity search, full-text search, and SQL analytics on the same data set. All query types are accelerated by the corresponding secondary indexes defined in the Lance specification. You can perform semantic searches on embedded vectors, execute BM25 searches on keywords, and apply complex SQL predicates, all through a unified interface on the same table.

- **Blazing-fast random access**

  Lance offers 100 times faster random access compared to Parquet or Iceberg. Unlike traditional formats, Lance can maintain high performance even when randomly accessing scattered rows across the entire dataset. With its highly optimized file format, efficient row addressing at the table level, and secondary indexing, Lance can instantly access individual records across multiple files. This makes it ideal for real-time ML services, random sampling, and interactive applications.

- **Native multimodal data support**

  Lance supports storing images, videos, audio, text, and embeddings alongside traditional tabular data in a single unified format. Its blob encoding efficiently handles large binary objects through lazy loading, while optimized vector storage accelerates similarity searches. This is particularly suitable for AI/ML workloads, where raw data, machine learning features, generated titles, and embeddings need to be stored together for multimodal retrieval and generative AI (GenAI) workflows.

- **Schema evolution**

  In most open table formats, schema evolution involves only metadata and is fast. However, backfilling column values in existing rows typically requires a full table rewrite. Lance supports data evolution, an efficient schema evolution with backfilling, making it highly suitable for ML feature engineering, embedding vectors, and media content management. Adding a new column with data is as simple as writing a new Lance file to a Lance table, without the need to rewrite the entire dataset.

2.1.2 Lance Table Directory Structures

The Lance table directory is the initial location where a dataset is created. Each Lance table has only one root directory, which serves as the primary storage location for dataset files. The root directory of a Lance table contains standard subdirectory structures (**data/**, **_versions/**, **_transactions/**, **_deletions/**, **_indices/**, **_refs/**, and **tree/**), which are used to organize data set files.

 Copyright © Huawei Technologies Co., Ltd.

```
{table_name}/
  data/
    *.lance                 -- Column data file
  _versions/
    *.manifest              -- Manifest file (one for each version)
    latest_version_hint.json  -- Latest version information
  _transactions/
    *.txn                   -- Transaction file (commit coordination)
  _deletions/
    *.arrow                 -- Deletion vector file (Arrow format)
    *.bin                   -- Deletion vector file (bitmap format)
  _indices/
    {UUID}/
      ...                   -- Index content (varying by index type)
  _refs/
    tags/
      *.json                -- Tag metadata
    branches/
      *.json                -- Branch metadata
  tree/
    {branch_name}/
      ...                   -- Branch column data file
```

- **Data files**

  Format: **data/{uuid-based-filename}.lance**

  Data files use UUID-based file names optimized for S3 throughput. The file name is generated from a 16-byte UUID, with the first three bytes converted into a 24-character binary string and the remaining 13 bytes converted into a 26-character hexadecimal string, resulting in a 50-character file name. The binary prefix (rather than hexadecimal) provides maximum entropy for each character, enabling S3's internal partitions to quickly identify access patterns and scale accordingly, thereby minimizing throttling.

  Example: 00011111101100011010111107c58ef4c3cbdbf814d0d5d0d97.lance

- **Manifest file list**

  Manifest files are stored in the **_versions/** directory, and their naming scheme supports atomic commits.

  – V1: **{version}.manifest -** *Monotonically increasing version number* (for example, **1.manifest, 2.manifest**)

  – V2: **{u64::MAX - version:020}.manifest** - *Reverse lexicographic order*, which helps efficiently discover the latest version (for example, the first version of V2 is **18446744073709551614.manifest**).

  _versions/latest_version_hint.json records the latest committed version in JSON format.

  The content of the **latest_version_hint.json** file is as follows:

  ```
  {"version": 42}
  ```

  This file speeds up the discovery of the latest version in storage systems where list operations are costly. Readers can read hints and probe for later versions through HEAD requests instead of listing the entire directory. If a hint is missing or expired, the reader rolls back to a full list operation. This hint is purely an optimization method. It can always be safely deleted without affecting correctness and can be ignored by readers that do not recognize it. Writers can choose not to write this file.

- **Transaction files**

  Format: **_transactions/{read_version}-{uuid}.txn**

  read_version indicates the table version on which the transaction is built.

 Copyright © Huawei Technologies Co., Ltd.

Example: **_transactions/5-550e8400-e29b-41d4-a716-446655440000.txn**

- **Deletion files**

  Format: **_deletions/{fragment_id}-{read_version}-{id}.{extension}**

  Deletion files use two extensions: **.arrow** for the Arrow IPC format (sparse deletions) and **.bin** for the Roaring bitmap format (dense deletions).

  Example: **_deletions/42-10-a1b2c3d4.arrow**

- **Index files**

  Each index's content is stored in the **_indices/{UUID}** directory under the base path. This location is referred to as the **index directory**. The actual content stored in the index directory depends on the index type. These can be any files defined by the index implementation. However, they are typically composed of Lance files that contain the index data structures. This allows the reuse of existing Lance file format code to read and write index data.

2.1.3 Lance Indexes

Lance treats indexes as independent, redundant data structures, hierarchically built on top of table row identifiers. This design frees the file format from built-in search structures and allows the index format to evolve independently of the table layout. Lance supports three types of indexes: scalar indexes, vector indexes, and system indexes.

Scalar Indexes

Scalar indexes accelerate queries on scalar data types such as integers, timestamps, and strings. The scalar indexes supported by Lance are listed in **Table 2-1**.

Table 2-1 Scalar index types

Index Type	Storage Files	Accelerated Queries	Description
Zone Map	zonemap.lance	Equals, Range, IsIn, IsNull	Data is divided into fixed-size regions, and **min**/**max**/**null_count** statistics are maintained for predicate pushdown and scan pruning. False positives may be included and need to be rechecked.
BTree	page_lookup.lance + page_data.lance	Equals, Range, IsIn, IsNull	Two-level structure with an in-memory BTree cache at the upper layer and sub-indexes storing sorted values and row IDs at the lower layer. The indexes balance the memory structures and disk structures.

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Storage Files	Accelerated Queries	Description
Bitmap	bitmap_page_lookup.lance	Equals, Range, IsIn, IsNull	Bit arrays are used to represent the presence or absence of values. The index is suitable for low-cardinality columns and provides extremely fast query performance.
Bloom Filter	bloomfilter.lance	Equals, IsIn, IsNull	The index adopts a probabilistic data structure that allows for fast membership testing. It is space-efficient, with possible false positives but no false negatives. The index is implemented through the Split Block Bloom Filter (SBBF) to optimize the SIMD operation.
Full Text Search	tokens.lance + docs.lance + invert.lance + metadata.lance	contains_tokens, match, phrase, boolean, multi_match, boost	An inverted index that maps terms to the documents that contain them. The index supports BM25 scoring, phrase queries, and multiple tokenizers.
Label List	bitmap_page_lookup.lance	array_has / array_contains, array_has_all, array_has_any	The label list index is optimized for columns that contain multiple labels or tags per row. It uses underlying bitmap indexes to provide efficient set-based queries for multi-valued columns.
N-gram	ngram_postings.lance	contains	The N-gram index breaks text into overlapping sequences (trigrams) to enable efficient substring matching. It provides fast text search by indexing all triple sequences in text after applying ASCII folding and lowercasing.
R-Tree	page_data.lance	Intersects, Contains, Within, Touches, Crosses, Overlaps, Covers, CoveredBy, IsNull	The index is built based on bounding boxes to organize data. It is designed to accelerate rectangle-based pruning operations.

 Copyright © Huawei Technologies Co., Ltd.

Vector Indexes

Vector indexes are specialized for approximate nearest neighbor search (ANN) of high-dimensional embeddings. Lance divides each vector index into three parts: clustering, sub-index, and quantization.

- Clustering: Partitions all vectors into distinct, non-overlapping clusters. Lance currently supports Inverted Files (IVF) as the primary clustering mechanism and uses the k-means clustering algorithm to partition vectors.
- Sub-index: Determines how vectors are organized for search. Supported types include FLAT (exact search) and HNSW (hierarchical navigable small world graph for fast approximate search).
- Quantization: Defines how vectors are stored and compressed. Options include Product Quantization (PQ), Scalar Quantization (SQ), RabitQ (RQ), and FLAT (no quantization).

Index types are typically denoted as **{*clustering*}_{*sub_index*}_{*quantization*}**. If the sub-index is FLAT, it is usually omitted, simplifying the notation to **{*clustering*}_{*quantization*}**.

Table 2-2 Vector index types

Index Type	Name	Description
IVF_PQ	Inverted file + Product quantization	Combines IVF clustering and PQ compression for efficient storage and search.
IVF_HNSW_SQ	Inverted file + HNSW + Scalar quantization	Uses IVF coarse clustering and HNSW fine-grained search, combined with scalar quantization.
IVF_SQ	Inverted file + Scalar quantization	Combines IVF clustering and scalar quantization for balanced compression.
IVF_RQ	Inverted file + RabitQ	Combines IVF clustering and RabitQ, using binary quantization for extreme compression.
IVF_FLAT	Inverted file + FLAT	Uses IVF clustering and exact vector storage for precise search within clusters.

 Copyright © Huawei Technologies Co., Ltd.

System Indexes

System indexes are auxiliary structures that support internal table maintenance and row identifier parsing. They are not directly queried by end users. For example:

- **Fragment Reuse Index (FRI)**: Optimizes fragment operations during compaction and dataset updates, supporting efficient remapping after compression.
- **MemWAL Index**: A centralized structure of all MemWAL metadata. It stores configuration (shard specs and indexes to be maintained), merge progress, and shard status snapshots.

2.1.4 Lance Table Schema Evolution

A schema describes the structure of a Lance table, including all fields, data types, and metadata. It uses a logical type system that represents data types as strings mapped to Apache Arrow. Each field has a unique identifier (field ID). Fields can be renamed or reordered without breaking references, enabling powerful schema evolution and version tracking. Field IDs ensure that data files can be correctly interpreted even if the schema changes over time. During table creation, field IDs are assigned to all fields in a depth-first order, starting from 0.

- Adding a column: Assign a new field ID and add it to the schema.
- Deleting a column: Remove a field from the schema. Its ID can be reused if it is no longer referenced.
- Renaming a column: Change the field name while retaining the ID.
- Reordering columns: Change the field order in the schema while retaining the IDs.
- Type evolution: Change the data type of a field. This operation may require rewriting the columns in the data, depending on the changed type.

2.1.5 Data Types Supported by Lance Tables

Table 2-3 Data types supported by Lance

Lance Data Type	Description
null	Null type (no value)
bool	Boolean value (true/false)
int8, int16, int32, int64	Signed 8-bit, 16-bit, 32-bit, and 64-bit integers
uint8, uint16, uint32, uint64	Unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers
halffloat	IEEE 754 half-precision floating-point number (16-bit)
float	IEEE 754 single-precision floating-point number (32-bit)
double	IEEE 754 double-precision floating-point number (64-bit)

 Copyright © Huawei Technologies Co., Ltd.

Lance Data Type	Description
string	Variable-length UTF-8 encoded string
large_string	Variable-length UTF-8 string (supporting large offsets)
binary	Variable-length binary data
large_binary	Variable-length binary data (supporting large offsets)
decimal:128:p:s	decimal:<*Bit width*>:<*Precision*>:<*Scale*>. The precision (p) is the total number of digits (up to 38 digits for **Decimal128**), and the scale (s) is the number of digits after the decimal point (0 ≤ s ≤ p).
decimal:256:p:s	decimal:<*Bit width*>:<*Precision*>:<*Scale*>. The precision (p) is the total number of digits (up to 76 digits for **Decimal256**), and the scale (s) is the number of digits after the decimal point (0 ≤ s ≤ p).
date32:day	Day-based date (days since epoch)
date64:ms	Millisecond-based date (milliseconds since epoch)
time32:s, time32:ms	Time in seconds since midnight, time in milliseconds since midnight
time64:us, time64:ns	Time in microseconds since midnight, time in nanoseconds since midnight
timestamp:unit:tz	timestamp:<*Unit*>:<*Time zone*>. The unit can be s (second), ms (millisecond), us (microsecond), or ns (nanosecond). The time zone is an IANA time zone string (such as UTC or America/New_York) or a hyphen (-) indicating no time zone.
duration:s, duration:ms, duration:us, duration:ns	Duration in seconds, duration in milliseconds, duration in microseconds, and duration in nanoseconds
struct	Nested type
list or list.struct (if the element is a struct)	Variable-length value list, variable-length struct value list
large_list or large_list.struct	Variable-length list (supporting large offsets), variable-length struct value list (supporting large offsets)
fixed_size_list:type:size	Fixed-length list, particularly applicable to vector embeddings. For example, **fixed_size_list:float:128** indicates a fixed-length list of 128 floating point numbers.
fixed_size_binary:size	Fixed-size binary data. For example, **fixed_size_binary:16** is used for MD5 hashing, and **fixed_size_binary:32** is used for SHA-256 hashing.

 Copyright © Huawei Technologies Co., Ltd.

Lance Data Type	Description
dict:value_type:key_type:bool	Dictionary value type It is in the format of **dict:**<*Value type*>:<*Key type*>:<*Whether ordered*>. For example, **dict:string:int16:false** indicates a dictionary-encoded string using int16 keys.
map	Key-value pair (Currently, only unordered keys are supported.)

2.1.6 Lance Namespace and LakeFormation Catalog

Lance Namespace

Lance Namespace is a namespace used for logical isolation and hierarchical organization of tables and datasets. It provides naming uniqueness and permission control, and supports classified management of data assets across multiple teams and projects. It defines a standardized interface for catalog interactions, such as table discovery, table location parsing, and coordinated commits. Lance Namespace abstracts implementations like LakeFormation Catalog through a unified interface called **LanceNamespace**.

LakeFormation Catalog

LakeFormation Catalog is used to manage table collections and provides functions such as table discovery, management, and transaction coordination.

LakeFormation, as a mature lakehouse governance tool, offers core capabilities such as metadata registration, fine-grained permission control, and data access audit. It has the following features:

- Metadata can be centrally stored and accessible in LakeFormation.
- Multiple key functions are supported, covering database, table, and location management, as well as metadata conversion. Tables support metadata management, property configuration, and persistence operations.
- The atomic semantics provided by LakeFormation are used together with the external manifest transaction commit mechanism of Lance to commit Lance transactions.
- Permission integration between Lance and LakeFormation is established cross all layers, including the LakeFormation user layer, OBS layer, and engine layer.

2.2 Lance on Aura

2.2.1 Preparations Before Using Lance

Before using Aura to perform read and write operations on Lance data tables, ensure that you have obtained the required access permissions for LakeFormation. Otherwise, the operation may fail due to insufficient permissions.

 Copyright © Huawei Technologies Co., Ltd.

For details about permissions, see **User and Permission Management**.

2.2.2 Lance Configurations and Types

Mapping between Aura types and Lance types

Table 2-4 Lance types corresponding to Aura types

Aura	Lance
smallint	int16
tinyint	int8
integer	int32
bigint	int64
numeric	Decimal128
decimal	Decimal128
real	float32
float4	float32
double precision	float64
float8	float64
boolean	boolean
char	string
varchar	string
text	string
date	date
timestamp without time zone	timestamp without time zone
bytea	binary
array	list
struct	struct

2.2.3 Lance on Aura DDL Syntax Description

Constraints

Lance tables do not support the creation of partitions and bucket columns.

 Copyright © Huawei Technologies Co., Ltd.

Creating a Lance Table

When using the **CREATE TABLE** syntax to create a Lance table, you need to specify **STORE AS Lance**. For details about the syntax, see **CREATE TABLE**. The metadata of the created Lance table is stored in LakeFormation, including the table name, table path, current table version, and table parameters. The actual data files of the Lance table are stored in the OBS bucket.

Example:

```
CREATE TABLE lance_tbl(
col1 int,
col2 array<double>,
col3 bytea
)
TABLEPROPERTIES (
'enable.blob.col3'='true',
'arrow.fixed-size-list.col2' = '128'
) STORE AS Lance;
```

Supported TABLEPROPERTIES parameters:

Table 2-5

Parameter	Type	Description
enable.blob.col_name	Boolean	Specifies the **col_name** column as a BLOB column. Only supported for columns of bytea type. For more information about BLOB column support in Lance, see **Blob Columns**.
arrow.fixed-size-list.col_name	Integer	Specifies that the **col_name** column is stored as a fixed-length vector using Arrow's FixedSizeList type. The value ranges from 1 to 2147483647. Only supported for one-dimensional arrays of the floating point numbers.
max.rows.per.file	Integer	Specifies the maximum number of rows that can be written to a single file. The value ranges from 1 to 9223372036854775807. The default value is **1048576**.
max.bytes.per.file	Integer	Specifies the maximum number of bytes that can be written to a single file. The value ranges from 1 to 9223372036854775807. The default value is 90 GiB.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
lance.encoding.compression	String	Specifies the compression algorithm. The value is a string and can be set to **zstd**, **lz4**, or **fsst**. The **zstd** algorithm supports configuration of the compression level, which ranges from 0 to 22. If no compression level is specified, the default level 3 is used. Compression is disabled by default. For details about the three compression modes, see **Compression Modes Supported by Lance**.
lance.batch.readahead	Integer	Specifies the number of batches to prefetch. The value ranges from 1 to 9223372036854775807. The default value is **4**.
lance.fragment.readahead	Integer	Specifies the number of fragments to prefetch. The value ranges from 1 to 9223372036854775807. The default value is **32**.
lance.block.size	Integer	Specifies the minimum number of bytes per I/O request sent to the storage layer. The value ranges from 40 to 34359738368. The default value is 1 MB.
metadata.cache.size	Integer	Specifies the size of the metadata cache, in bytes. The value ranges from 1 to 34359738368. The default value is 1 GiB.
io.buffer.size	Integer	Specifies the size of the I/O buffer, in bytes. The value ranges from 1 to 34359738368. The default value is 2 GiB.
index.cache.size	Integer	Specifies the size of the index cache, in bytes. The value ranges from 1 to 34359738368. The default value is 6 GiB.
index.maximum.nprobes	Integer	Specifies the maximum number of partitions to be selected during index search. The value must be greater than **0**. By default, all partitions are searched.
index.minimum.nprobes	Integer	Specifies the minimum number of partitions to be selected during index search. The value must be greater than **0**. The default value is **1**.
index.refine.factor	Integer	Specifies the candidate result amplification factor for exact-distance-based reranking during index search. The value must be greater than **0**. By default, reranking is not performed.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description
index.explore.factor	Integer	Specifies the capacity of the priority queue during HNSW index search. The value must be greater than **0**. The default value is **1.5k**.

NOTE

Description of some table parameters:

- **enable.blob.col_name**: To specify a column of **bytea** type with the column name **blob_col1** as a BLOB column, write it as **enable.blob.blob_col1 = true**.
- **arrow.fixed-size-list.col_name**: To define a fixed-length vector column named **col2** with a vector dimension of 1024, configure it as **arrow.fixed-size-list.col2 = 1024**.
- **lance.encoding.compression**: Specifies the compression method, which can be **zstd**, **lz4**, or **fsst**, for example, **lance.encoding.compression = lz4**. If the compression level needs to be configured for **zstd**, you can write it as **lance.encoding.compression = zstd:22**.

Truncating a Lance Table

Use the **TRUNCATE TABLE** syntax to clear data in a table. For details about the syntax, see **TRUNCATE**.

Example:

```
TRUNCATE TABLE lance_tbl;
```

Clearing Historical Versions of a Lance Table

You can use the VACUUM TABLE syntax to clear historical versions of a Lance table and data files, transaction files, and index files that are no longer referenced to reclaim the OBS storage space. After operations such as INSERT, UPDATE, DELETE, INSERT OVERWRITE, and ALTER INDEX OPTIMIZE are performed on a Lance table, a new version is generated. The old version and its referenced files are retained and can be cleared using VACUUM TABLE.

The following clearing options are supported:

- **older_than_seconds**: An integer ranging from 1 to 31536000. Only the old versions whose generation time exceeds the specified threshold are deleted. If this parameter is omitted, the Lance kernel uses the default value (14 days).
- **delete_rate_limit**: An integer ranging from 1 to 2147483647. It indicates the maximum number of deletion operations per second, which is used to prevent OBS traffic limiting from being triggered.

VACUUM TABLE returns a row of clearing statistics, which contains six columns:

 Copyright © Huawei Technologies Co., Ltd.

Table 2-6 Returned columns

Column	Type	Description
bytes_removed	bigint	Number of bytes removed during clearing
old_versions	integer	Number of old versions removed
data_files_removed	integer	Number of data files removed
transaction_files_removed	integer	Number of transaction files removed
index_files_removed	integer	Number of index files removed
deletion_files_removed	integer	Number of deletion-marked files that are removed

Example:

Remove old versions generated 2 seconds ago.

```
VACUUM TABLE lance_tbl WITH (older_than_seconds = 2);
```

Use the default parameters (the Lance kernel deletes old versions generated 14 days ago by default).

```
VACUUM TABLE lance_tbl;
```

Compacting Lance Table Data

The compaction operation rewrites data in a Lance table, merges multiple small fragments into a larger fragment, and reorganizes the row groups within the fragment based on the configuration. This reduces the number of fragments, lowers metadata overhead, and improves query performance. After multiple INSERT, DELETE, and other operations, a Lance table can accumulate multiple fragments and deletion files. Compaction can merge these small fragments and, based on the configuration, determine whether to materialize deletions of data that has been marked for deletion.

The following table lists the supported compaction options.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-7 Parameters

Field	Type	Value Range	Default Value	Description
target_rows_per_fragment	bigint	[1, int32)	1048576 (about 1 million lines)	Specifies the target number of rows per fragment. Compaction rewrites the fragments that meet the criteria to ensure that the new fragments have a row count as close as possible to the value of this parameter. (The number of rows in the new fragment does not have to strictly equal the value of this parameter, and not all fragments with fewer rows than this value will be merged separately.)
max_rows_per_group	bigint	[1, int32)	1024	Specifies the maximum number of rows in a row group (**Arrow RecordBatch**) within each fragment. If the row count of a fragment exceeds the value, the fragment is divided into multiple row groups.
max_bytes_per_file	bigint	[1, int64)	Default value of **write_dataset**	Specifies the maximum size (in bytes) of a single **.lance** file.

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Value Range	Default Value	Description
materialize_deletions	boolean	true/false	true	Specifies whether to materialize deletions of data that has been marked for deletion during compaction. • If this parameter is set to **true**, the deleted data will be removed from the new fragment. • If this parameter is set to **false**, the deletion files are retained and the original fragments are referenced.
materialize_deletions_threshold	real	[0.0, 1.0]	0.1 (10%)	Specifies the threshold for determining when to materialize deletions during the compaction process. When **materialize_deletions** is set to **true**, materialized deletions are performed only for fragments whose deletion ratio exceeds the threshold. Deletion ratio = Number of deleted rows/Total number of rows in the fragment

 Copyright © Huawei Technologies Co., Ltd.

Field	Type	Value Range	Default Value	Description
num_threads	bigint	[0, int32)	Number of CPU cores	Specifies the number of threads used during compaction (controlling the CPU parallelism). If this parameter is set to **0** or not specified, it will automatically use the number of CPU cores on the current server.
batch_size	bigint	[0, int32)	Default value of scanner	Specifies the batch size of data read by the scanner each time (affecting memory usage and processing speed).
defer_index_remap	boolean	true/false	false	Specifies whether to delay index remapping. If this parameter is set to **true**, compaction does not update the associated indexes immediately. Instead, it records the mapping of fragments, and leaves the index remapping to be completed by subsequent operations. This reduces the execution time of compaction.
max_source_fragments	bigint	[0, int32)	Any	Specifies the maximum number of source fragments that a compaction task can merge. This parameter is used to limit the amount of data that can be rewritten at a time.

 Copyright © Huawei Technologies Co., Ltd.

Example:

Basic usage:

```
OPTIMIZE lance_table REWRITE DATA;
```

Specify the number of target fragments.

```
OPTIMIZE lance_table REWRITE DATA WITH OPTIONS ('target_rows_per_fragment' = '10000');
```

Perform multiple-parameter configuration.

```
OPTIMIZE lance_table REWRITE DATA  WITH OPTIONS (    'target_rows_per_fragment' = '10000',
'materialize_deletions' = 'true',    'materialize_deletions_threshold' = '0.2',    'max_source_fragments' = '50');
```

Dropping a Lance Table

Use the **DROP TABLE** command to delete a Lance table. This removes both the table metadata and its data. For managed tables, the metadata and data can be restored from the LakeFormation console after deletion. For details about the syntax, see **DROP TABLE**.

Example:

```
DROP TABLE lance_tbl;
```

Modifying Table Columns

- Adding columns:

  Add one or more columns to a specified table. When adding columns, you must specify the column names and data types. New columns are appended to the end of the existing column list.

  ```
  ALTER TABLE tbl_name ADD COLUMN col1 typ1, col2 typ2, ....;
  ```

- Dropping columns:

  Remove one or more columns from a specified table. You cannot drop all columns from a table.

  ```
  ALTER TABLE tbl_name DROP COLUMNS (col1, col2,...);
  ```

- Renaming columns:

  Rename a specified column. You cannot rename multiple columns at the same time.

  ```
  ALTER TABLE tbl_name RENAME COLUMN old_name TO new_name;
  ```

- Changing column types:

  Change the data type of one or more specified columns.

  ```
  ALTER TABLE tbl_name ALTER COLUMN a TYPE new_type, b TYPE new_type, ...;
  ```

  Currently, only widening conversions (upgrading from lower-precision to higher-precision types) are allowed. Conversions outside the following ranges are not supported:

  - Integer types: From SMALLINT to INT or BIGINT, or from INT to BIGINT.
  - Floating-point types: From FLOAT4 to FLOAT8.
  - Numeric types: From low-precision type to high-precision type. The number of decimal places (scale) cannot be updated.
  - String types: Length increases of VARCHAR or CHAR.

 Copyright © Huawei Technologies Co., Ltd.

Modifying Table Parameters

- Adding or modifying parameters:

  Only the following parameters are supported: **max.rows.per.file**, **max.bytes.per.file**, and **lance.encoding.compression**.

  ```
  ALTER TABLE tbl_name SET TABLEPROPERTIES('max.rows.per.file'='1024', 'max.bytes.per.file'='2048', 'lance.encoding.compression'='zstd');
  ```

- Removing parameters:

  Only the following parameters are supported: **max.rows.per.file**, **max.bytes.per.file**, and **lance.encoding.compression**.

  ```
  ALTER TABLE tbl_name UNSET TABLEPROPERTIES('max.rows.per.file', 'max.bytes.per.file', 'lance.encoding.compression');
  ```

2.2.4 Querying and Writing Data to a Lance Table

Querying a Lance Table

You can query data in a Lance table using the **SELECT** syntax. For details about the syntax, see **SELECT**.

Note:

Only the latest committed data can be queried.

Example:

```
SELECT * FROM lance_tbl;
SELECT a, b FROM lance_tbl WHERE a = 1;
```

Lance supports full-text search using indexes with Boolean operators. In the following example, **content1** and **content2** are columns that have full-text indexes defined on them.

Example:

```
SELECT * FROM lance_tbl WHERE content1@@ 'Index';
SELECT * FROM lance_tbl WHERE content1 @@ 'Search' AND content2 @@ 'index';
SELECT * FROM lance_tbl WHERE content1 @@ 'Arrow' OR content2 @@ 'fox';
SELECT * FROM lance_tbl WHERE content1 @@ 'fox' AND NOT content2 @@ 'lazy';
SELECT * FROM lance_tbl WHERE (content1 @@ 'fox' AND content1 @@ 'brown') OR content2 @@ 'python';
```

Writing Data to a Lance Table

You can insert one or more rows of data into a Lance table using the INSERT command. For details about the syntax, see **INSERT**.

Example:

```
INSERT INTO lance_tbl VALUES (1, 'a', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT OVERWRITE INTO lance_tbl VALUES (1, 'b', 1.1234567, 100.11, '2025-03-03', '2025-03-03 16:00:00');
INSERT INTO lance_tbl (col1, col2, col3, col4, col5, col6) SELECT col1, col2, col3, col4, col5, col6 FROM lance_tbl2 WHERE col1 > 18;
```

2.2.5 Updating a Lance Table

Lightweight Update of Lance Table Data

You can update one or more rows in a Lance table using the **UPDATE** syntax. For details about the syntax, see **UPDATE**.

 Copyright © Huawei Technologies Co., Ltd.

Note:

The **UPDATE FROM** statement and the **UPDATE** statement that contains multi-table joins in the **SET** or **WHERE** clause are not supported.

Example:

```
UPDATE lance_tbl SET b = 1 WHERE a = 1;
```

2.2.6 Using Indexes for Lance Tables

Creating an Index

You can create an index for a vector column in a Lance table. The vector column must be declared as a fixed-size list. For details, see **Creating a Lance Table**.

Supported vector index types include: **ivfflat**, **ivfpq**, **ivfrq**, **ivfsq**, **hnsw**, **hnswpq**, and **hnswsq**. For details about how to create an index, see **CREATE INDEX**.

Distributed creation of vector indexes is supported. The generated index can be either a global index, or multiple local sub-indexes. Each sub-index corresponds to a partition of the table and indexes non-overlapping data. You can control the number of table partitions using the **fq_query_vdn** parameter. During index creation, if **merge_segment** is set to **1**, a global index is generated. Otherwise, multiple sub-indexes are generated. Using multiple sub-indexes accelerates queries.

Table 2-8 Vector index algorithms and their respective use cases

Index Type	Description	Build Overhead	Compression Rate	Query Latency (Small topK)	Recall	Memory Usage	Use Cases	Tuning Parameters
ivfflat	Clustering and bucketing (partitioning). Stores original vectors. During query, it traverses bucket centers and scans only a subset of buckets to measure the exact distance between the query vector and candidate vectors.	Low	N/A	Medium	Extremely high	High	Small to medium tables; high precision requirements; sufficient memory	Build: nlists, sample_rate, max_iters Query: minimum_nprobes, maximum_nprobes, refine_factor

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Description	Build Overhead	Compression Rate	Query Latency (Small topK)	Recall	Memory Usage	Use Cases	Tuning Parameters
ivfpq	Clustering and bucketing. Stores product quantization (PQ) codebook and vector quantization results. Index scans use approximate distances based on quantization. The exact distances can be computed and re-ranked using the refinement process.	Medium	High	Moderately low	Medium	Low	100M+ records; low precision requirements; memory-constrained; large topK; hybrid scalar-vector queries. Increasing refine_factor can improve recall, but values greater than 2 will affect performance.	Build: nlists, sample_rate, max_iters, num_sub_vectors Query: minimum_nprobes, maximum_nprobes, refine_factor
ivfrq	Clustering and bucketing. Stores residuals using RaBitQ (1-bit quantization). Index scans use approximate distances, and the refinement process enables precise re-ranking.	Moderately high	Moderately high	Medium	High	Low to medium	100M+ records; medium precision requirements; memory-constrained; large topK; hybrid scalar-vector queries. Lower refine_factor values yield high recall.	Build: nlists, sample_rate, max_iters, num_bits (currently only 1-bit supported) Query: minimum_nprobes, maximum_nprobes, refine_factor

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Description	Build Overhead	Compression Rate	Query Latency (Small topK)	Recall	Memory Usage	Use Cases	Tuning Parameters
ivfsq	Clustering and bucketing. Stores scalar quantization (8-bit) values. Index scans use approximate distances, and the refinement process enables precise re-ranking.	Medium	Medium	Medium	High	Medium	100M+ records; high precision requirements; sufficient memory; large topK; hybrid scalar-vector queries.	Build: nlists, sample_rate, max_iters, num_bits Query: minimum_nprobes, maximum_nprobes, refine_factor
hnsw	Clustering generates a multi-layer directed graph. Stores original vectors. During a query, it navigates the graph to access a subset of bucket centers; and scans only selected buckets using exact distance computation.	High	N/A	Low	Extremely high	High (>ivfflat)	100M+ records; low-latency requirement; small topK; high precision; hybrid queries with low scalar filter selectivity.	Build: nlists, sample_rate, max_iters, m, ef Query: ef, refine_factor, minimum_nprobes, maximum_nprobes

 Copyright © Huawei Technologies Co., Ltd.

Index Type	Description	BuildOverhead	CompressionRate	Query Latency (Small topK)	Recall	Memory Usage	Use Cases	Tuning Parameters
hnswpq	Clustering generates a multi-layer directed graph. Stores product quantization (PQ) codebook and vector quantization results. Index scans use approximate distances based on quantization. Exact distances can be computed and re-ranked using the refinement process.	Extremelyhigh	High	Low	Medium	Medium	100M+ records; low-latency requirement; small topK; low precision; hybrid queries with low scalar filter selectivity. Increasing refine_factor can improve recall.	Build: nlists, sample_rate, max_iters, m, ef, num_sub_vectors Query: ef, refine_factor, minimum_nprobes, maximum_nprobes
hnswsq	Clustering generates a multi-layer directed graph. Stores scalar quantization (8-bit) values. Index scans use approximate distances, and the refinement process enables precise re-ranking.	High	Medium	Low	High	Moderatelyhigh	100M+ records; low-latency requirement; small topK; moderately high precision; hybrid queries with low scalar filter selectivity.	Build: nlists, sample_rate, max_iters, m, ef, num_bits Query: ef, refine_factor, minimum_nprobes, maximum_nprobes

Example:

Create an index named **lance_index_01** on the Lance table **lance_tbl** using the IVF-PQ algorithm with L2 (Euclidean) distance. The IVF index partitions the vector space into 2 clusters. Each vector is split into 4 sub-vectors, and multiple sub-indexes will be generated.

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE INDEX IF NOT EXISTS lance_index_01 ON lance_tbl USING ivfpq (vector_column vector_l2_ops)
WITH (nlists=2, num_subvectors=4, merge_segment=0);
```

Table 2-9 Scalar index algorithms and their respective use cases

Index Type	Description	Use Cases	Supported Data Types	Parameters
btree	Sorts data by value and organizes it into pages. Stores min/max values per page in memory to build a B-tree index. During a query, it prunes irrelevant pages via the B-tree and scans only a few candidate pages to return exact results.	Range queries, equality queries, IN queries, exact filtering on large datasets	Integer: tinyint, smallint, int, integer, bigint, long Floating-point: real, float, double precision, double Boolean: boolean String: varchar(n), char(n), string, text Time: date, timestamp Decimal: decimal(p,s), numeric(p,s)	N/A

Example:

Create an index named **lance_idx** on the **column_name** column of the Lance table **lance_tbl**, using the B-tree algorithm.

```
CREATE INDEX IF NOT EXISTS lance_idx ON lance_tbl USING btree (column_name);
```

 Copyright © Huawei Technologies Co., Ltd.

Table 2-10 Full-text index algorithms and their respective use cases

Index Type	Description	Use Cases	Supported Data Types	Parameters
inverted	Tokenizes source text into individual terms. During a query, it looks up these terms in the inverted index and returns matching records.	Word matching, phrase matching	• text • varchar • bpchar • text[] • varchar[] • bpchar[]	base_tokenizer, with_position, and max_token_length. For details about the parameters, see **Table 2-12**.

NOTE

In Aura, scalar indexes in Lance format support only the B-tree type, and full-text indexes support only the inverted type.

Example:

Create an index named **lance_fts** on the **column_name** column of the Lance table **lance_tbl**, using the full-text (inverted) index algorithm. The tokenizer is set to **ngram** (which supports Chinese text), position tracking is enabled (for phrase queries), and the maximum token length is set to **20** (tokens exceeding this length will be truncated).

```
CREATE INDEX IF NOT EXISTS lance_fts ON lance_tbl USING inverted (column_name) WITH
(base_tokenizer = 'ngram', with_position = true, max_token_length = 20);
```

Table 2-11 Available tokenizers for full-text indexes

Tokenizer Value	Description	Feature Dependency
simple	Tokenization by whitespace and punctuation (default)	Always available
whitespace	Tokenization by whitespace only	Always available
raw	No tokenization. The entire text is treated as a single token.	Always available

 Copyright © Huawei Technologies Co., Ltd.

Tokenizer Value	Description	Feature Dependency
ngram	N-gram tokenization (overlapping character sequences), which must be used together with **min_ngram_length and max_ngram_length**.	Always available
icu	ICU dictionary-based Unicode tokenization	Always available
lindera/<model>	Lindera Japanese morphological analysis (e.g., **lindera/ipadic**)	You need to manually download the dictionary.
jieba or jieba/<model>	Jieba Chinese word segmentation (**jieba** is equivalent to **jieba/default**)	You need to manually download the dictionary.

Table 2-12 Parameters

Parameter	Type	Default Value	Recommended Value	Description
base_tokenizer	string	simple	• Use **simple** for English. • Use **jieba/ngram** for Chinese. • Use **lindera/ipadic** for Japanese. • Use **ngram** for fuzzy prefix search.	Basic tokenizer, which determines how text is split into tokens. The options are as follows: • **simple** (tokenization by whitespace and punctuation) • **whitespace** (tokenization by whitespace only) • **raw** (no tokenization) • **ngram** (N-Gram overlapping character sequences) • **icu** (ICU Unicode tokenization) • **lindera/<model>** (Japanese morphological analysis) • **jieba** (Chinese word segmentation)

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Default Value	Recommended Value	Description
with_position	bool	false	• Set this parameter to **true** if phrase query is required. • Retain the value **false** for keyword search.	Whether to store token location information in the inverted index. If this parameter is set to **true**, phrase query is supported, but the index size will increase significantly (by more than 10 times). This parameter is not available for the N-gram tokenizer.
max_token_length	uint32	40	• Retain the default value **40** in common scenarios. • Set this parameter to **None** for languages without spaces, such as Chinese and Japanese. • Set this parameter to **None** if **contains_tokens** acceleration is required.	Maximum length of a token. Tokens whose length is greater than or equal to this value will be filtered out. The value **None** indicates no limit on the token length. If this parameter is set to a value other than **None**, index acceleration cannot be used for **contains_tokens** SQL queries.

Querying Using Indexes

On columns where indexes have already been created, you can perform query operations using those indexes. Currently, Aura supports the following index query types for Lance tables: vector index query, scalar index query, hybrid query of vectors and scalars, and full-text index query.

- Vector index query

Example:

```
SELECT * FROM lance_vector ORDER BY vector_col::vector <-> '[-0.2,0.5,-3.3,0.0]'::vector LIMIT 10;
```

This query uses the index on the **vector_col** column in the **lance_vector** table to retrieve the top 10 nearest neighbors for the vector **[-0.2,0.5,-3.3,0.0]**. The **ORDER BY ... LIMIT** syntax triggers an Approximate Nearest Neighbor (ANN) query. The

 Copyright © Huawei Technologies Co., Ltd.

vector column involved in the **ORDER BY** clause must have an available index. The **<->** operator is a distance operator. The following distance operators are currently supported in Aura.

Table 2-13

Operator	Description
<->	L2 distance
<#>	Inner product distance
<=>	Cosine distance

NOTE

The vector column must contain an index that matches the distance operator used in the query. If no matching index exists, the query will fail. If multiple indexes with the same distance operator exist in the vector column, the most recently created index is selected.

- Scalar index query

Example:

```
SELECT * FROM ext_type_cols WHERE big_val > 0 ORDER BY id;
```

Scalar index queries do not have a specific syntax. Actually, they are standard **WHERE** clause queries. If a scalar index exists in the **big_val** column, Aura pushes the **WHERE** predicate down to Lance. The Lance internal optimizer determines whether to use the index.

- Hybrid query of vectors and scalars

Example:

```
SELECT count(1) FROM (
    SELECT id, name FROM lance_mixed_q
    WHERE id > 4000 AND age < 30
    ORDER BY vec::vector <-> '[1,2,...]'::vector LIMIT 1024
 ) t1;
```

The hybrid queries combine the syntax for vector index queries and scalar index queries. Aura employs a **Post-filter** policy: it first performs a vector index query to return the top-K results, and then filters these results based on the **WHERE** conditions.

- Full-text index query

Example:

```
SELECT * FROM fts_test WHERE content @@ 'Aura' ORDER BY id;
SELECT * FROM fts_test WHERE content @@ 'lance' AND id < 5 ORDER BY id;
```

The operator for full-text index query is **@@**. It can be used together with scalar query filter conditions. However, hybrid queries combining full-text indexes and vector indexes are not supported.

 Copyright © Huawei Technologies Co., Ltd.

Showing Existing Indexes

You can print all indexes of a Lance table or all Lance tables in the current database. For details, see **SHOW INDEX**.

Example:

```
SHOW INDEX;
SHOW INDEX lance_tbl;
```

Deleting an Existing Index

You can delete an existing index from a Lance table. For details, see **DROP INDEX**.

Example:

```
DROP INDEX lance_index_01 ON lance_tbl;
```

Rebuilding or Updating an Existing Index

CAUTION

Updating the data in a Lance index column can invalidate some index fragments or the entire index. Therefore, you need to periodically run the **optimize** command to incrementally rebuild the index.

You can rebuild an existing index. The rebuilt index retains the same name and parameters, but the index data is rebuilt from the latest full data of the indexed column. For details, see **ALTER INDEX**.

Example:

```
ALTER INDEX IF EXISTS lance_index_01 ON lance_tbl REBUILD;
```

You can update an existing index. When new data is inserted or deleted, existing indexes are not updated immediately. This means the latest data cannot be retrieved via the index, which may impact query performance (though query results may be correct).

To ensure that an existing index includes the latest data without performing a full rebuild, use the index optimization feature.

For details, see **ALTER INDEX**.

Example:

```
ALTER INDEX IF EXISTS lance_index_01 ON lance_tbl OPTIMIZE;
```

2.2.7 Lance Late Read

Late Read is a performance optimization feature for Lance foreign tables, specifically designed for vector top-*k* queries. By splitting data scan into two phases: early read and late read, this feature significantly reduces the amount of invalid reads on wide columns (such as images, long texts, and original vectors), thereby enhancing query performance.

In a standard execution plan, a query such as **SELECT <*Wide columns*> FROM t ORDER BY vec <-> q LIMIT k** requires reading all projected columns for every

 Copyright © Huawei Technologies Co., Ltd.

candidate row in the entire table before sorting and obtaining top-*k* results. Wide columns are read before sorting and most of them are discarded after sorting.

Late Read divides the scan into two phases:

1. Early read phase: Only the three system columns (**_rowid**, **xc_node_id**, and **_distance**) along with the columns required by the **WHERE** predicates and JOIN keys are read. The IVF index is used to generate a candidate set, and the top-*k* results are obtained after sorting.
2. Late read phase: The top-k **_rowid** values are routed back to the DataNodes where the data resides (via the **RESTORE** stream) based on **xc_node_id**. Only the Lance **take_rows** function is called for these *k* rows to retrieve the wide columns. The amount of data read from wide columns is reduced from O (Number of candidate rows × Width of wide columns) to O (*k* × Width of wide columns).

Late Read Execution Policies

Late Read supports two execution policies, which can be specified through parameters or automatically selected by the optimizer.

- Single-phase policy (**single_phase**)

  A single Late Read operation retrieves vector columns and other wide columns together.

  Figure 2-1 Single-phase Late Read plan

  ```
  id
  1  Vector Result
  2    └── Vector Limit
  3      └── Vector Streaming (GATHER)
  4        └── Vector Sort                          -- Fine ranking
  5          └── Vector Result                      -- Calculate the distance using precise vectors.
  6            └── Vector Foreign Late Read Scan    -- Late read: take_rows retrieves all late-read columns at once.
  7              └── Vector Streaming (RESTORE)     -- Route back to source DN by xc_node_id.
  8                └── Vector Result                -- Drop _distance column.
  9                  └── Vector Limit               -- Coarse ranking: top-k * rf
  10                   └── Vector Sort              --Sort Key: _distance
  11                     └── Vector Streaming (BROADCAST)
  12                       └── Vector Foreign Scan
                         Output: _distance, xc_node_id, _rowid
  ```

  Characteristics: The plan has a shallow hierarchy, with only one **RESTORE** operation and one **take_rows** operation. However, wide columns are fetched for the coarse-ranking candidate set. Some of them may be eliminated in the subsequent fine-ranking, resulting in I/O waste.

- Two-phase policy (**two_phase**)

  This policy splits the Late Read into two distinct steps: First, retrieve only the vector columns to perform fine-ranking and finalize the top-*k* results. Then, retrieve only the wide columns for the final *k* rows.

 Copyright © Huawei Technologies Co., Ltd.

Figure 2-2 Two-phase Late Read plan

```
id
 1  Vector Limit
 2    └── Vector Result
 3      └── Vector Streaming (GATHER)
 4        └── Vector Sort
 5          └── Vector Foreign Late Read Scan              -- Late read 2: Read non-vector columns.
 6            └── Vector Streaming (RESTORE)
 7              └── Vector Limit                           -- Fine ranking: top-k
 8                └── Vector Sort
 9                  └── Vector Streaming (BROADCAST)
10                    └── Vector Result                    -- Calculate the distance using precise vectors.
11                      └── Vector Foreign Late Read Scan  -- Late read 1: Read only vector columns.
13                        └── Vector Streaming (RESTORE)
14                          └── Vector Result              -- Drop _distance column.
15                            └── Vector Limit             -- Coarse ranking: top-k * nf
16                              └── Vector Sort            -- Sort Key: _distance
17                                └── Vector Streaming (BROADCAST)
18                                  └── Vector Foreign Scan
                                    Output: _distance, xc_node_id, _rowid
```

Characteristics: Wide columns are read only for the final *k* rows, minimizing I/O. However, this policy incurs the overheard of an additional **RESTORE** stream and an extra call to Python's **take_rows**. This policy is cost-effective only when there are non-vectorized wide columns are sufficiently large or numerous.

GUC Parameters

late_read_strategy

- **Description**: Specifies the Late Read policy for top-*k* queries on Lance DFS tables. The value is a list. **none**, **single_phase**, **two_phase**, and **auto** options are mutually exclusive options. **nested_late_read** and **join_late_read** are independent switches.
- **Values**:
  - **none**: Late Read is disabled. The query follows the original plan, reading all projected columns during the scan phase.
  - **auto**: The optimizer automatically selects between **none**, **single_phase**, and **two_phase** based on the cost and column width.
  - **single_phase**: The query is forced to follow a single-phase plan, skipping benefit checks.
  - **two_phase**: The query is forced to follow a two-phase plan, skipping benefit checks.
  - **nested_late_read**: An independent switch that allows Late Read to work in nested query blocks (subqueries and CTEs).
    - Default: Late Read is not applied to subqueries and queries in CTEs.
    - Enabled: Late Read is determined independently for each query block. Top-*k* queries in subqueries and CTEs can use Late Read.
  - **join_late_read**: An independent switch that allows Late Read to work across INNER JOINs (only Hash Joins are supported). After the setting, Late Read takes effect only when all conditions below are met:

 Copyright © Huawei Technologies Co., Ltd.

  - All joins are INNER JOINs (presence of OUTER, SEMI, or ANTI joins causes a fallback to the standard plan).
  - There is exactly one Lance top-*k* driving table in the query, and the total number of Lance tables equals this count.
  - Equi-join conditions exist for the driving table.
  - No aggregation, **GROUP BY**, **DISTINCT**, or set operations are present.
- **Default Value**: **none**

enable_topk_compensation

- **Description**: Specifies whether to enable top-*k* compensation read for Lance ANN scans. When enabled, Lance ANN scans use a **filter_ratio**-based *k*-value calculation and a while-loop compensation mechanism to ensure that the final output meets the **LIMIT *n*** requirement after **WHERE** filtering.
- **Values**:
  - **on**: The while-loop compensation mechanism is enabled to ensure that the final output row count satisfies the *k*-value requirement.
  - **off**: The while-loop compensation mechanism is disabled. If the **WHERE** filter rate is high, the output row count may be significantly less than the requested *k*-value.
- **Default Value**: **off**

2.3 Lance Development FAQs

2.3.1 How Do I Handle Lance Table Concurrency Conflicts?

Lance tables use a Multi-Version Concurrency Control (MVCC) model. Each table version is represented by immutable dataset metadata (manifests) that references data fragments (data files), associated index files, and optional transaction files. A read operation targets a specific committed version, enabling snapshot reads. A write operation creates a new version by committing updated metadata, generating atomic, versioned updates at the table level and snapshot-isolation–style guarantees within a single table or dataset. In Lance, a transaction refers to any committed operation—such as a write, update, delete, or index creation—that results in a new immutable table version.

Lance Transaction Commit Process

- Construct a transaction based on a baseline version, for example, by appending new fragments or creating new index references. A transaction is essentially a set of changes applied to the baseline manifest.
- Write new files associated with the transaction—new fragments, deletion files, or index files—to storage. These files are immutable once written.
- Atomically commit the updated metadata by writing a new manifest file as the entry point for the new table version. If concurrent writes cause the atomic commit to fail, Lance initiates conflict detection, followed by retry or rebase.

 Copyright © Huawei Technologies Co., Ltd.

Transaction Conflicts

When multiple transactions are based on the same base version, a conflict window occurs: each attempt competes to publish a different "next version" of the table by committing its own manifest. Lance detects this at commit time and applies conflict resolution logic, which may include merge, rebase, or retry depending on the operation type and compatibility rules. Otherwise, the commit fails with a conflict error.

- Lance transaction operation types

Table 2-14 Lance transaction operation types

Transaction Name	Description	SQL/API (or N/A)
Append	Adds new data fragments without modifying existing data.	INSERT
Delete	Marks rows as deleted using a deletion vector.	N/A
Overwrite	Replaces the entire table with new data, schema, and configuration.	INSERT OVERWRITE, TRUNCATE
CreateIndex	Creates, replaces, or drops secondary indexes (vector, scalar, or full-text).	CREATE INDEX/DROP INDEX/REBUILD INDEX/ OPTIMIZE INDEX
Rewrite	Reorganizes data without changing semantic content (e.g., compaction, deduplication, reordering).	N/A
Merge	Adds new columns, requiring updates across all fragments to include the new schema.	ADD COLUMN, ALTER COLUMN TYPE
Project	Removes or renames columns via metadata-only schema changes.	DROP COLUMN, ALTER COLUMN RENAME
Restore	Restores the table to a previous version.	N/A
ReserveFragments	Pre-allocates fragment IDs for future write operations.	N/A

 Copyright © Huawei Technologies Co., Ltd.

Transaction Name	Description	SQL/API (or N/A)
Clone	Creates a shallow or deep copy of a table; shallow copies reference existing data via base paths.	N/A
Update	Modifies row values without adding or deleting rows.	UPDATE
UpdateConfig	Updates table or schema metadata without modifying data files.	ALTER SET/UNSET
DataReplacement	Replaces data in a specific column range with new data files.	N/A
UpdateMemWalState	Updates the state of the MemWAL (write-ahead log–based index).	N/A
UpdateBases	Adds new base paths to reference external data locations.	N/A

- Lance transaction conflict types

  Table 2-15 Lance transaction conflict types

Conflict Type	Description	Resolution Mechanism
Rebasable Conflict	Transaction can be transformed to incorporate concurrent changes while preserving intent.	Automatically rebased and retried at commit layer; fails if retry limit is exceeded.
Retryable Conflict	Transaction cannot be merged automatically.	Commit fails with non-retryable error.
Incompatible Conflict	Fundamental semantic conflict; retrying disallowed.	Commit fails with non-retryable error.

- Lance transaction conflict matrix:

 Copyright © Huawei Technologies Co., Ltd.

Current Operations \ Historical Operations	Delete	Update	CreateIndex	Rewrite	Overwrite	Append	DataReplament	Merge	Restore	ReserveFragments	Project	UpdateConfig	UpdateMemWalState	Clone	UpdateBases
Delete	?	?	√	?	×	√	?	×	×	√	√	√	×	√	√
Update	?	?	√	?	×	√	?	×	×	√	√	√	?	√	√
CreateIndex	√	√	?	?	×	√	?	√	×	√	√	√	×	√	√
Rewrite	?	?	?	?	×	√	?	×	×	√	√	√	√	√	√
Overwrite	√	√	√	√	?	√	√	√	√	√	√	?	×	√	√
Append	√	√	√	√	×	√	√	√	×	√	√	√	×	√	√
DataReplament	√	√	?	?	×	√	?	√	×	√	√	√	×	√	√
Merge	×	×	√	×	×	×	×	×	×	√	×	√	×	√	√
Restore	√	√	√	√	√	√	√	√	√	√	√	√	×	√	√
ReserveFragments	√	√	√	√	×	√	√	√	×	√	√	√	√	√	√
Project	√	√	√	√	×	√	√	×	×	√	×	√	×	√	√
UpdateConfig	√	√	√	√	?	√	√	√	√	√	√	?	√	√	√
UpdateMemWalState	×	?	√	√	×	×	×	×	×	√	×	√	?	×	√
Clone	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
UpdateBases	√	√	√	√	√	√	√	√	√	√	√	√	√	√	?

Legend: √ indicates a mergeable conflict, × an unresolvable conflict, and ? means retry/rebase required.

Example:

When two Append transactions execute concurrently (i.e., both inserting data into the table), they represent a mergeable conflict. In this case, Lance attempts to merge both sets of appended fragments into a single consistent next version and retries the commit. If the merge succeeds within the retry limit, both transactions succeed without conflict. If retries are exhausted, the commit fails with a conflict error.

 Copyright © Huawei Technologies Co., Ltd.

3 Paimon Development Guide

3.1 Introduction to Paimon

3.1.1 Paimon Functions

Apache Paimon is an open-source data lake storage system designed for streaming data processing and the lakehouse architecture. It combines the low-cost and open storage capabilities of data lakes with the real-time update and transactional consistency capabilities of databases, making it well-suited for scenarios such as real-time data warehouses, offline data warehouses, and unified batch-streaming processing.

Paimon adopts a storage structure similar to LSM-Tree, supporting high-throughput writes, primary key updates, deletions, partial updates, and data compaction. After data is written, it is organized through snapshots, manifest files, and data files. Each commit generates a consistent snapshot, enabling atomic commits, version management, time travel, and fault recovery. For primary key tables, Paimon merges multiple changes based on the primary key. This makes it ideal for processing database CDC data and directly building data lake tables that are updated in real time.

Paimon provides the following features:

- Unified batch-streaming write: Supports continuous Flink streaming writes and Spark batch writes, allowing a single copy of data to serve both streaming and batch processing scenarios.
- Real-time updates for primary key tables: Supports insert, update, and delete operations by primary key based on the LSM-Tree structure, making it suitable for CDC data ingestion and real-time aggregation.
- Append-only tables: A table that does not support primary keys. It is suitable for logs and event streams where updates are not required. This mode offers higher write performance.
- Time travel: Supports querying historical data through snapshots, which can be used for data auditing, historical comparison, and fault recovery.

 Copyright © Huawei Technologies Co., Ltd.

- Schema evolution: Supports schema changes such as adding columns, dropping columns, renaming columns, and changing column types, without rewriting data files.
- Compaction: Merges small files into large ones to optimize query performance and storage efficiency. Both synchronous and asynchronous modes are supported.
- Changelog Producer: Directly generates change logs from Paimon tables, allowing downstream systems to subscribe to incremental data for streaming processing.
- Tag and Branch: Allows you to create tags and branches for table snapshots, facilitating version management and data rollback.

Highlights

- Unified batch-streaming processing: The same dataset supports both streaming reads and batch queries, eliminating the need to maintain separate data pipelines.
- High-throughput writes: Optimized write paths based on LSM-Tree support data write scenarios with millions of TPS.
- Real-time updates: Primary key tables support update and delete operations by primary key, meeting the data change requirements of real-time data warehouses.
- Open ecosystem: Deep integration with Flink and support for multiple compute engines such as Spark.
- Transactional consistency: Each commit generates a consistent snapshot, providing ACID transaction guarantees and isolation for concurrent reads and writes.
- Low-cost storage: The underlying storage is based on HDFS or object storage, combined with the columnar storage format, achieving both low costs and high performance.

Concepts

- Table: A Paimon table is the basic organizational unit of data, containing information such as the schema, partitioning policies, primary key definitions, and attribute configurations. There are two types of tables: Append-Only tables and primary key tables.
- Snapshot: A snapshot is a consistent view of a table at a specific point in time. Each commit generates a snapshot, which serves as the basis for time travel and concurrency control.
- Schema: A schema describes the field definitions, data types, and field IDs of a table. Paimon uses field IDs to ensure data compatibility after schema evolution.
- Manifest: A manifest records the metadata of data files, including the file paths, partition values, row counts, and column statistics. It is used for filtering and scan planning during queries.
- Data file: A data file stores actual data, typically using columnar formats like ORC or Parquet.

 Copyright © Huawei Technologies Co., Ltd.

- Changelog: A changelog records data changes in a table. It is generated through the Changelog Producer mechanism for downstream streaming consumption.
- Commit: An operation that atomically writes data changes to a table and generates a new snapshot. Paimon uses optimistic concurrency control to handle concurrent commits.

File Organization

Paimon separates data into the metadata management layer and data storage layer.

- Metadata management layer:
  - Snapshot files: stored in JSON format, containing the current snapshot information, including the schema, partition specifications, references to all manifest files, and references to the previous snapshot. A new snapshot file is generated for each commit.
  - Manifest files: stored in Avro format, containing a list of data files. Each row records detailed information about a data file, including the file path, partition values, file size, row count, and column-level statistics (such as min/max values, null counts). This allows queries to filter out unnecessary files.
  - Manifest list files: stored in Avro format. A manifest list file is generated for each commit, containing the paths of the manifest files involved in the commit and their partition range information.
- Data storage layer: By default, the columnar file format such as ORC or Parquet is used.

3.1.2 Paimon Table Properties

When creating or modifying a Paimon table, you can use **TBLPROPERTIES** to set related parameters. Paimon table properties are classified into **Core**, **Read**, **Write**, and **Compaction** properties.

Core Properties

Table 3-1 Core properties

Property	Default Value	Description
bucket	-	Number of bucket partitions for even data distribution.
primary-key	-	Primary key column. Use commas (,) to separate multiple columns. Primary key tables support update and delete operations.
partition	-	Partition column. Use commas (,) to separate multiple columns.

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
changelog-producer	none	Changelog producer type. The options are **none**, **input**, **lookup**, and **full-compaction**.
write-mode	auto	Write mode. The options are **auto** and **append-only**.
merge-engine	deduplicate	Merge engine of the primary key table. The value can be **deduplicate**, **partial-update**, **aggregation**, or **first-row**.
format	orc	Data file format. The options are **orc** and **parquet**.
target-file-size	128 MB	Size of the target data file.
file.format	orc	File storage format.
file.compression	zlib(orc)/ snappy(parquet)	File compression algorithm.

Read Properties

Table 3-2 Read properties

Property	Default Value	Description
read.batch-size	1024	Batch size (number of rows) during reading, which controls how many rows are read from the file each time.
read.parquet.vectorization.enabled	false	Whether to enable vectorized reading for Parquet files.
read.orc.vectorization.enabled	false	Whether to enable vectorized reading for ORC files.
read.cache-enabled	true	Whether to enable the read caching for data files and manifest files to improve query performance.
read.cache.max-memory	256 MB	Maximum memory usage of the read cache.

 Copyright © Huawei Technologies Co., Ltd.

Property	Default Value	Description
read.scan.plan-fetch-batch-size	500	Number of manifest files fetched in batches during scan planning.
read.streaming.enabled	false	Whether to enable the streaming read mode for Flink streaming consumption scenarios.
read.streaming.time-gap	1s	Interval between two scans during streaming read.

Write Properties

Table 3-3 Write properties

Property	Default Value	Description
write.parallelism	1	Parallelism for write operations.
write.batch-size	500	Number of rows per batch for writing.
write.auto-compaction	false	Whether to automatically trigger a compaction after data is written.
write.compaction.min-num-messages	1000	Minimum number of messages required for triggering asynchronous compaction.
write.compaction.max-size-amplification	200	Maximum space amplification ratio for compaction, in percentage.
sink.parallelism	-	Parallelism of the Flink Sink operator.
sink.commit.max-retries	10	Maximum number of retries for commits.

 Copyright © Huawei Technologies Co., Ltd.

Compaction Properties

Table 3-4 Compaction properties

Property	Default Value	Description
compaction.min.file-num	5	Minimum number of files required for triggering a compaction.
compaction.max.file-num	50	Maximum number of files that can be merged in a single compaction.
compaction.max.size-amplification	200	Maximum space amplification ratio for compaction, in percentage. Compaction is triggered if the value is exceeded.
compaction.max.output.file-num	5	Maximum number of output files after compaction.
num-sorted-run.compaction-trigger	5	Number of LSM-tree levels that trigger sorted run merging.
num-sorted-run.stop-trigger	50	Threshold for the number of sorted runs that stop writing data and wait for compaction.
compaction.optimization-interval	-	Interval for asynchronous compaction optimization.
sort-engine	loser-tree	Sorting engine type. The value can be **loser-tree** or **janitor**.

3.1.3 Paimon Catalog

Paimon Catalog is the top-tier component within Paimon, responsible for managing the metadata and its associated operations across all Paimon tables. It manages the table schema and metadata, providing APIs for creating, querying, and modifying tables. The Catalog serves as the entry point for users to perform operations on Paimon tables.

The current AI DataLake version supports using LakeFormation Catalog as the Catalog component for Paimon tables.

 Copyright © Huawei Technologies Co., Ltd.

LakeFormation Catalog

LakeFormation Catalog relies on the LakeFormation metadata service to oversee the most recent snapshots.

It uses an optimistic concurrency control (OCC) mechanism to maintain data integrity during concurrent writes across multiple tenants.

All Paimon tables currently created on AI DataLake are LakeFormation Catalog tables.

Paimon supports the following Catalog types:

- Filesystem Catalog: A file-system–based Catalog that stores metadata in a file system. It is suitable for development and testing environments.
- Hive Metastore Catalog: A Catalog based on Hive Metastore that stores metadata in Hive Metastore and is compatible with the Hive ecosystem.
- LakeFormation Catalog: A Catalog based on LakeFormation that stores metadata in LakeFormation and provides fine-grained permission control and data access auditing capabilities.

3.1.4 Paimon Table Schema Evolution

Paimon supports in-place table evolution. You can evolve a table schema simply by modifying the metadata, without the need to create a table or rewrite data. Schema evolution is a metadata-level change, so no data files need to be rewritten to perform the update.

Schema Evolution

Paimon supports the following schema evolution changes:

- Adding a column: Adds a column to a table. The new column is assigned a unique field ID.
- Deleting a column: Removes an existing column from a table. Its field ID will not be reused.
- Renaming a column: Changes the name of an existing column. The field ID remains unchanged.
- Modifying the column order: Adjusts the position of a column in the table.
- Changing the type: Supports upgrading from a lower-precision type to a higher-precision type, such as from INT to BIGINT or from FLOAT to DOUBLE.

  Paimon uses field IDs to ensure data compatibility after schema evolution. Each field is assigned a unique integer ID when it is created. This ID is written into the schema of the data file. When reading a data file, the field IDs are used to map the columns in the metadata to the columns in the file. Any change in the column name or type will not affect the reading of the historical data file.

Evolution Scope Supported by Data Types

- Basic data types:
  - INT can be upgraded to BIGINT.

 Copyright © Huawei Technologies Co., Ltd.

  - FLOAT can be upgraded to DOUBLE.
  - DECIMAL(P, S) can be upgraded to DECIMAL(P', S) (P' > P) to increase the precision.
- Struct type
  - Delete a field from a struct.
  - Add a field to a struct.
  - Rename a field in a struct.
  - Reorder fields in a struct.
  - Evolve the field type in a struct (following the rules for basic data types).
- List type
  - Add a new element field to a list (only when the element is of the struct type).
  - Delete an element field from a list (only when the element is of the struct type).
  - Rename an element field in a list (only when the element is of the struct type).
  - Evolve the element type in a list (following the rules for basic data types).
- Map type
  - Add a new value field to a map (only when the value is of the struct type).
  - Delete a value field from a map (only when the value is of the struct type).
  - Rename a value field in a map (only when the value is of the struct type).
  - Evolve the key or value type in a map (following the rules for basic data types). Note that map keys do not support adding or deleting struct fields that would change equality.

3.1.5 Data Types Supported by Paimon Tables

Basic Data Types

Supported primitive types are defined in **Table 3-5**.

Table 3-5 Table 1 Basic data types

Data Type	Description
BOOLEAN	True or false
INT	32-bit signed integers.
BIGINT	64-bit signed integers
FLOAT	32-bit IEEE 754 floating points.
DOUBLE	64-bit IEEE 754 floating points

 Copyright © Huawei Technologies Co., Ltd.

Data Type	Description
DECIMAL(P, S)	Fixed-point decimals. **P** specifies the precision (the total number of digits), and **S** specifies the scale (the number of digits in the fractional part). The precision must be less than or equal to 38.
DATE	Calendar dates without the time zone or time
TIMESTAMP	Timestamps with microsecond precision, excluding the time zone
TIMESTAMP_LTZ	Timestamps with microsecond precision and the time zone
STRING	Arbitrary-length character sequences encoded with UTF-8
BINARY	Arbitrary-length byte arrays
BYTES	Byte arrays

Nested Data Types

- A struct is a tuple of typed values. Each field in the tuple is named and has an integer ID that is unique in the struct schema. Fields can be defined as either optional or mandatory.
- A list is an ordered collection of values with a certain element type. Elements can be of any type and can be optional or mandatory.
- A map is a collection of key-value pairs with a key type and a value type. Map keys are required and map values can be either optional or required.

3.1.6 Paimon Table Partitions

Paimon supports partitioned tables. Data can be divided into multiple subsets based on specified rules through partitioning, improving query performance and data management efficiency. Partitioning is a key optimization technique in Paimon. During query, partition pruning can significantly reduce the amount of data that needs to be scanned.

Partitioning Type

- Identity partitioning: Directly uses the value of the source field as the partition value. This is the most commonly used partitioning method, such as partitioning by a date field.
- Bucket partitioning: Performs hash calculation on the values of a specified field and then applies the modulo to evenly distribute data across a specified number of buckets. This type of partitioning is suitable for scenarios requiring uniform data distribution and concurrent writes.
- Multi-level partitioning: Supports using multiple partitioning fields simultaneously. For example, data can be partitioned by date and then by bucket, enabling hierarchical data organization.

 Copyright © Huawei Technologies Co., Ltd.

Partition Evolution

Paimon supports partition evolution, allowing you to add or delete partition fields using the **ALTER TABLE** statement. Partition changes only affect newly written data. Existing data files retain their original partition layout without data migration.

Partition evolution rules:

- Adding a partition field: New data is written according to the new partition scheme, while historical data remains unchanged.
- Deleting a partition field: After a partition field is deleted, new data is no longer partitioned based on the field.
- Stable partition field IDs: During partition evolution, partition field IDs remain unchanged to ensure that historical data can be correctly read.

3.2 Paimon on Spark

3.2.1 Interconnecting Spark with Paimon Catalog

AI DataLake Spark allows you to use LakeFormation to manage metadata and authentication (row and column permissions) for Paimon tables. This simplifies data lake governance, enabling unified metadata management, fine-grained permission control, and transactional guarantees. Spark reads and writes to Paimon tables through data catalogs. You can select a data catalog based on your business scenarios.

Notes and Constraints

- **A LakeFormation instance has been created with metadata planned.**

  Ensure that LakeFormation catalog is located in the same region as the AI Data Lake workspace. Otherwise, they cannot be associated in subsequent processes.
- After a Paimon data catalog is configured, the job-level configurations take precedence over endpoint-level configurations. In addition, endpoint-level configurations take effect only after the endpoint is restarted.

Configuring a Data Catalog

Using the Default Data Catalog

Iceberg is used as the default data catalog for AI Data Lake Spark jobs. The system automatically pre-configures **spark.aidatalake.lakehouse.default.format** to **iceberg**.

To set Paimon as the default data directory, set **spark.aidatalake.lakehouse.default.format** to **paimon**.

- Configuration via parameters in **spark_config** of the API

  For details about the API, see **RunSparkJob** or **RunSparkSql**. Example API request:

 Copyright © Huawei Technologies Co., Ltd.

```
"endpoint_name": "spark-sql-test",
 "catalog_context": {
   "catalog_name": "your_catalog_name",// Any catalog.
   "database_name": "default"
 },
 "statement": "SELECT * FROM your_table LIMIT 10",
 "spark_config": {
   "spark.aidatalake.lakehouse.default.format": "paimon"
 }
}
```

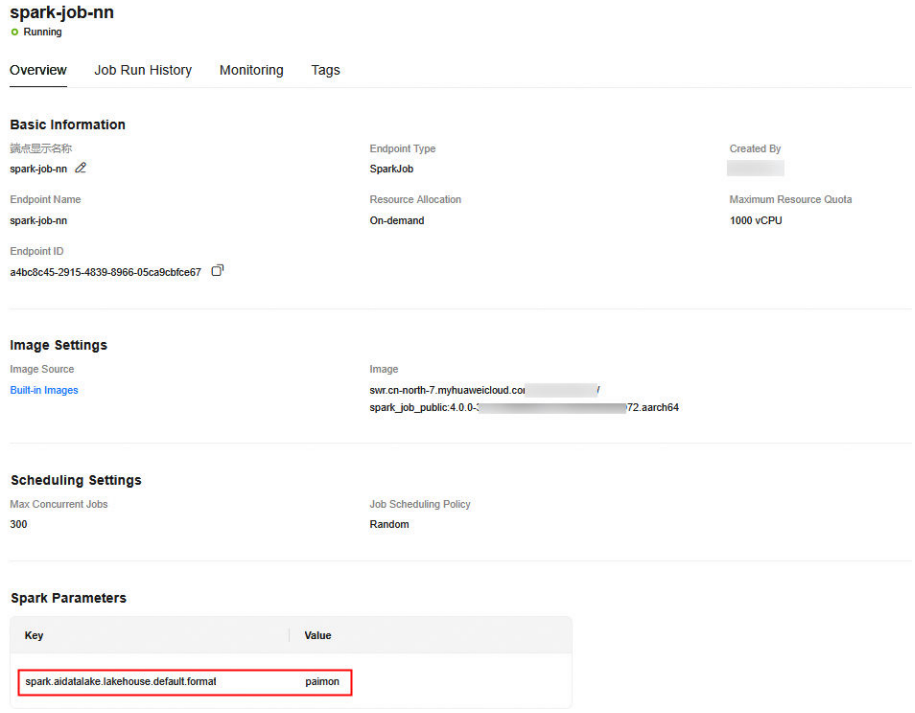
- Configuration via the endpoint creation or modification page
  - Configuration via the endpoint creation page

    Figure 3-1 Configuring on the endpoint creation page

  - Configuration via the endpoint modification page

    Figure 3-2 Configuring on the endpoint modification page

 Copyright © Huawei Technologies Co., Ltd.

Using a Custom Data Catalog

CAUTION

- Paimon does not support views or user-defined functions (UDFs).
- If a non-customized catalog is selected, the system uses the configured default data catalog (specified by **spark.aidatalake.lakehouse.default.format**).

To use a custom Paimon catalog, configure **spark.sql.catalog.{paimon_catalog_name}=org.apache.paimon.spark.SparkCatalog**, where **paimon_catalog_name** needs to be replaced with your target catalog name.

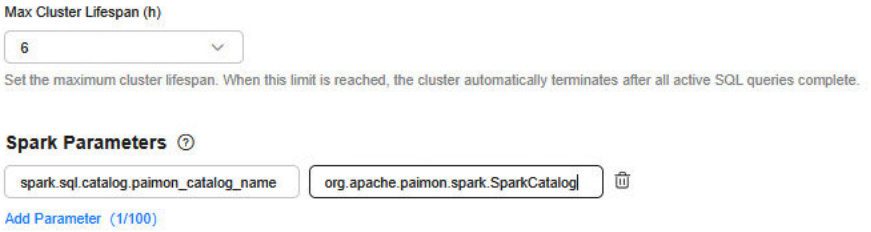
Configuration methods:

- Configuration via parameters in **spark_config** of the API

  For details about the API, see **RunSparkJob** or **RunSparkSql**. The following is an example API request:

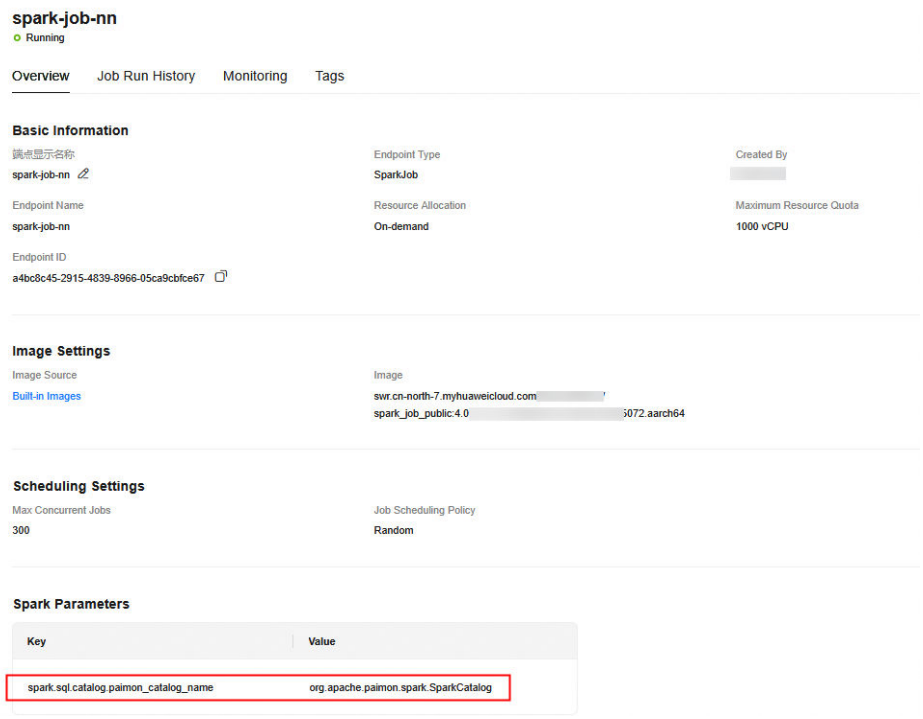
```
"endpoint_name": "spark-sql-test",
 "catalog_context": {
   "catalog_name": "paimon_catalog_name",//The catalog name must be the same as that in the configuration. This catalog can be used only for Paimon.
   "database_name": "default"
 },
 "statement": "SELECT * FROM paimon_catalog_name.db.your_table LIMIT 10",//Three-part naming configuration is supported.
 "spark_config": {
   "spark.sql.catalog.paimon_catalog_name": "org.apache.paimon.spark.SparkCatalog"
 }
}
```

- Configuration via the endpoint creation or modification page
  - Configuration via the endpoint creation page

  - Configuration via the endpoint modification page

 Copyright © Huawei Technologies Co., Ltd.



3.2.2 Paimon Data Types

Spark and Paimon support different type sets. Paimon automatically performs type conversions, but not all mapping types are supported. Therefore, you need to understand the type conversions in Paimon before designing the column types of a table. All Spark data types can be found in the **org.apache.spark.sql.types** package.

Table 3-6 Type conversion

Spark Data Type	Paimon Data Type
StructType	RowType
MapType	MapType
ArrayType	ArrayType
BooleanType	BooleanType
ByteType	TinyIntType
ShortType	SmallIntType
IntegerType	IntType
LongType	BigIntType
FloatType	FloatType
DoubleType	DoubleType
StringType	VarCharType(Integer.MAX_VALUE)

 Copyright © Huawei Technologies Co., Ltd.

Spark Data Type	Paimon Data Type
VarCharType(length)	VarCharType(length)
CharType(length)	CharType(length)
DateType	DateType
TimestampType	LocalZonedTimestampType
TimestampNTZType(Spark3.4+)	TimestampType
DecimalType(precision, scale)	DecimalType(precision, scale)
BinaryType	BinaryType
VariantType	VariantType

3.2.3 DDL Syntax

Creating a Table

You can create Paimon tables in a Paimon catalog. Tables created within the Paimon catalog are managed by the catalog. When a table is deleted from the catalog, its table files are also deleted.

```
CREATE TABLE my_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING
)
PARTITIONED BY (dt, hh)
TBLPROPERTIES (
    'primary-key' = 'dt,hh,user_id',
    'bucket' = '2'
)
LOCATION 'obs://{bucket}/path/to/table';
```

- **PARTITIONED BY**: Configures table partitioning.
- **LOCATION '(fully-qualified-uri)'**: Sets the table location. If **LOCATION** is specified, a foreign table is created. Deleting the table only removes the metadata, not the data files. If data already exists in the specified location, you can omit column definitions, partitions, and properties. The new table will inherit all information from the metadata of the existing table.
- **bucket**: Configures bucketing. A bucket is the smallest storage unit for reads and writes. Unpartitioned tables, or partitions in partitioned tables, are sub-divided into buckets. This distribution enables concurrent read and write operations, significantly improving overall data read and write performance. You can specify bucketing columns by providing the **bucket-key** option. If no **bucket-key** option is specified, the primary key (if defined) will be used as the bucket key.
- **COMMENT 'table documentation'**: Specifies the table description.
- **TBLPROPERTIES ('key'='value', ...)**: Specifies the table configuration.

 Copyright © Huawei Technologies Co., Ltd.

- **primary key**: Specifies the primary key field.
  - You can specify a primary key during Paimon table creation to create a primary key table. Primary keys consist of a set of columns that contain unique values for each record. If you define a table with a primary key, you can insert, update, or delete records in the table. Paimon enforces data ordering by sorting the primary key within each bucket, allowing you to achieve high performance by applying filtering conditions on the primary key. A primary key table adopts the LSM tree structure to store data, enabling efficient read and write operations.
  - If no primary key is specified, an append-only table is created. An append-only tables supports only INSERT operations. They do not generate changelogs and are suitable for scenarios that do not require streaming updates.

Notes

- The partition key must be selected from the primary key.
- Not all primary keys can be set as partition keys.

CTAS Syntax

Paimon supports executing CTAS (CREATE TABLE ... AS SELECT) as an atomic operation.

A new table generated by a CTAS operation does not inherit the partitioning scheme or table properties from the source table specified in the **SELECT** statement. To configure partitioning or table properties for the new table, you can explicitly declare them using the **PARTITIONED BY** clause and **TBLPROPERTIES** clause in the CTAS statement.

```
-- Basic CTAS
CREATE TABLE my_table_as USING paimon AS SELECT * FROM my_table;
-- CTAS with partitioning and primary key:
CREATE TABLE my_table_all_as USING paimon PARTITIONED BY (dt) TBLPROPERTIES ('primary-key' = 'dt,hh') AS SELECT * FROM my_table_all;
```

Notes

For append-only tables (tables without a primary key), if the **bucket** property is specified, the **bucket-key** property must also be specified.

RTAS Syntax

Paimon supports executing RTAS (REPLACE TABLE ... AS SELECT) as an atomic operation. Atomic table replacement creates a new snapshot based on the SELECT query results while preserving the table's history.

```
-- Basic RTAS
REPLACE TABLE paimon.db.sample USING paimon AS SELECT ...;
CREATE OR REPLACE TABLE paimon.db.sample USING paimon AS SELECT ...;
-- RTAS with partitioning and bucketing:
REPLACE TABLE paimon.db.sample  USING paimon  PARTITIONED BY (pt)  TBLPROPERTIES ('bucket' = '2', 'bucket-key' = 'id')  AS SELECT ...;
```

Deleting a Table

```
DROP TABLE my_table;
```

 Copyright © Huawei Technologies Co., Ltd.

- **MANAGED**: Deletes both metadata and data files in OBS.
- **EXTERNAL**: Deletes only metadata. Data files in OBS are retained.

ALTER TABLE

Paimon supports the following **ALTER TABLE** operations:

Setting/Removing table properties

```
ALTER TABLE my_table SET TBLPROPERTIES ('write-buffer-size' = '256 MB'); ALTER TABLE my_table UNSET TBLPROPERTIES ('write-buffer-size');
```

Setting/Removing table comments

```
ALTER TABLE my_table SET TBLPROPERTIES ('comment' = 'table comment'); ALTER TABLE my_table UNSET TBLPROPERTIES ('comment');
```

Renaming a table

```
ALTER TABLE my_table RENAME TO my_table_new;
```

Note: Exercise caution when using this syntax in object storage (such as OBS), as the renaming operations in object storage are not atomic.

Adding a column

```
ALTER TABLE my_table ADD COLUMNS (c1 INT, c2 STRING);
-- Add a nested column to the struct type.
ALTER TABLE my_table ADD COLUMN v.f3 STRING;
-- Add a nested column to a struct element in an array.
ALTER TABLE my_table ADD COLUMN v.element.f3 STRING;
-- Add a nested column to the value struct in a map.
ALTER TABLE my_table ADD COLUMN v.value.f3 STRING;
```

Specifying the column position

```
ALTER TABLE my_table ADD COLUMN c INT FIRST; ALTER TABLE my_table ADD COLUMN c INT AFTER b;
```

Renaming a column

```
ALTER TABLE my_table RENAME COLUMN c0 TO c1;
-- Rename a nested column.
ALTER TABLE my_table RENAME COLUMN v.f1 to f100;
ALTER TABLE my_table RENAME COLUMN v.element.f1 to f100;
ALTER TABLE my_table RENAME COLUMN v.value.f1 to f100;
```

Dropping a column

```
ALTER TABLE my_table DROP COLUMNS (c1, c2);
-- Drop a nested column.
ALTER TABLE my_table DROP COLUMN v.f2;
ALTER TABLE my_table DROP COLUMN v.element.f2;
ALTER TABLE my_table DROP COLUMN v.value.f2;
```

Dropping a partition

```
ALTER TABLE my_table DROP PARTITION (`id` = 1, `name` = 'paimon');
```

Modifying column comments

```
ALTER TABLE my_table ALTER COLUMN buy_count COMMENT 'buy count';
```

Modifying the column position

```
ALTER TABLE my_table ALTER COLUMN col_a FIRST; ALTER TABLE my_table ALTER COLUMN col_a AFTER col_b;
```

Modifying the column type (widening)

 Copyright © Huawei Technologies Co., Ltd.

```
ALTER TABLE my_table ALTER COLUMN col_a TYPE DOUBLE;
-- Modify the nested column type.
ALTER TABLE my_table ALTER COLUMN v.f2 TYPE BIGINT;
ALTER TABLE my_table ALTER COLUMN v.element.f2 TYPE BIGINT;
ALTER TABLE my_table ALTER COLUMN v.value.f2 TYPE BIGINT;
```

Tag Management

Creating a Tag

```
-- Create a tag based on the latest snapshot.
ALTER TABLE T CREATE TAG `TAG-1`;
-- Create the tag if it does not exist.
ALTER TABLE T CREATE TAG IF NOT EXISTS `TAG-1`;
-- Create a tag that can be retained for seven days.
ALTER TABLE T CREATE TAG `TAG-2` RETAIN 7 DAYS;
-- Create a tag based on a specified snapshot.
ALTER TABLE T CREATE TAG `TAG-3` AS OF VERSION 1;
-- Create or replace a tag.
ALTER TABLE T CREATE OR REPLACE TAG `TAG-5` AS OF VERSION 2 RETAIN 24 HOURS;
```

Deleting a tag

```
ALTER TABLE T DELETE TAG `TAG-1`;
ALTER TABLE T DELETE TAG IF EXISTS `TAG-1`;
ALTER TABLE T DELETE TAG `TAG-1,TAG-2`;
```

Renaming a tag

```
ALTER TABLE T RENAME TAG `TAG-1` TO `TAG-2`;
```

Viewing tags

```
SHOW TAGS T;
```

3.2.4 Writing Data to a Paimon Table

INSERT INTO

```
INSERT INTO my_table VALUES (1, 'Alice', '2024-01-01');
INSERT INTO my_table SELECT * FROM source_table;
```

INSERT OVERWRITE

```
-- Overwrite the entire table.
INSERT OVERWRITE my_table SELECT ...;
-- Overwrite specified partitions.
INSERT OVERWRITE my_table PARTITION (key1 = value1, key2 = value2) SELECT ...;
```

Dynamic partition overwrite

Spark uses static overwrite mode by default. To enable dynamic overwrite, set the following parameters:

```
SET spark.sql.sources.partitionOverwriteMode=dynamic;
INSERT OVERWRITE my_table VALUES (3, 'p1');
```

TRUNCATE TABLE

```
TRUNCATE TABLE my_table;
```

UPDATE (only for primary key tables)

```
UPDATE my_table SET name = 'Alice Updated' WHERE id = 1;
-- Update a field within a struct.
UPDATE t SET s.c2 = 'a_new' WHERE s.c1 = 1;
```

 Copyright © Huawei Technologies Co., Ltd.

Note: Primary Key tables do not support updating the values of primary key columns.

DELETE (only for primary key tables)

```
DELETE FROM my_table WHERE currency = 'UNKNOWN';
```

MERGE INTO (only for primary key tables)

```
MERGE INTO target
USING source
ON target.a = source.a
WHEN MATCHED THEN
   UPDATE SET *
WHEN NOT MATCHED THEN
   INSERT *;
```

Conditional MERGE INTO

```
MERGE INTO target
USING source
ON target.a = source.a
WHEN MATCHED AND target.a = 5 THEN
   UPDATE SET b = source.b + target.b
WHEN MATCHED AND source.c > 'c2' THEN
   UPDATE SET *
WHEN MATCHED THEN
   DELETE
WHEN NOT MATCHED AND c > 'c9' THEN
   INSERT (a, b, c) VALUES (a, b * 1.1, c)
WHEN NOT MATCHED THEN
   INSERT *;
```

Note: In the **UPDATE** clause, primary key tables do not support updating the values of primary key columns.

3.2.5 Querying Data in a Paimon Table

Batch query

Paimon's batch read returns all data from a table snapshot. By default, it returns the latest snapshot.

```
SELECT * FROM my_table;
```

Metadata Columns

Paimon supports reading hidden metadata columns.

Metadata Column	Description
__paimon_file_path	Path of the data file where the record is located.
__paimon_partition	Partition value of the record.
__paimon_bucket	Bucket ID of the record.
__paimon_row_index	Row index of the record in the data file.

```
SELECT *, __paimon_file_path, __paimon_partition, __paimon_bucket, __paimon_row_index FROM t;
```

 Copyright © Huawei Technologies Co., Ltd.

Note: Only append-only tables or tables with deletion vectors support querying metadata columns.

Time travel

```
-- Query by snapshot ID
SELECT * FROM t VERSION AS OF 1;
-- Query by timestamp
SELECT * FROM t TIMESTAMP AS OF '2023-06-01 00:00:00.123';
-- Query by tag
SELECT * FROM t VERSION AS OF 'my-tag';
```

Incremental query

```
-- Read incremental data between snapshots 12 and 20.
SELECT * FROM paimon_incremental_query('tableName', 12, 20);
-- Read incremental data between two specific timestamps.
SELECT * FROM paimon_incremental_between_timestamp('tableName', '2025-03-12 00:00:00', '2025-03-12 00:08:00');
```

Query optimization

It is recommended to specify filter conditions for partitions and primary keys during query to accelerate data skipping. Supported filter functions include:

- =, <, <=, >, >=
- IN (...)
- LIKE 'abc%'
- IS NULL

Paimon sorts data by primary key to optimize the performance of point queries and range queries. When using composite primary keys, it is recommended to match the leftmost prefix of the primary key in the query condition to improve query efficiency.

3.2.6 System Tables

Paimon allows you to access system tables to obtain detailed information, such as the snapshots, schemas, and options. System tables are identified by appending a dollar sign ($) and the metadata table name to the original table name. In some cases, the table name needs to be enclosed with back quotes to avoid syntax parsing conflicts.

```
SELECT * FROM my_catalog.my_db.`my_table$snapshots`
```

 Copyright © Huawei Technologies Co., Ltd.

Table 3-7 System tables

System Table Name (Suffix)	Function Description	Example
$snapshots	Queries the snapshot history of a table, including the snapshot ID, commit time, and record count occurred in the snapshot. It is used to understand commit and expiration information about that table and supports data time travel.	`SELECT * FROM my_table$snapshots;`
$schemas	Queries the historical schemas of the table to track structural changes over time.	`SELECT * FROM my_table$schemas;`
$options	Queries the table options specified by DDL. Options that are not displayed will use their default values.	`SELECT * FROM my_table$options;`
$audit_log	Queries the changelogs of the table. Through this table, you can get the rowkind column (+I, -U, +U, and -D) for filtering and auditing operations.	`SELECT * FROM my_table$audit_log;`
$ro	Read-optimized table. Only files that do not require merging are scanned. The read data may not be the latest, but the read performance is higher.	`SELECT * FROM my_table$ro;`
$files	Queries file details for a specific snapshot, including partition, bucket, file path, record count, size, and key-value range.	`SELECT * FROM my_table$files;`
$tags	Queries the tag history of the table, including the snapshots based on which the tags are created and some snapshot history. You can time travel to specific tag data by tag name.	`SELECT * FROM my_table$tags;`
$consumers	Queries the next snapshot information consumed by all consumers.	`SELECT * FROM my_table$consumers;`
$manifests	Queries all manifest files contained in the latest or specified snapshot of the current table.	`SELECT * FROM my_table$manifests;`

 Copyright © Huawei Technologies Co., Ltd.

System Table Name (Suffix)	Function Description	Example
$aggregation_fields	Queries the historical aggregation information of the table.	SELECT * FROM my_table $aggregation_fields;
$partitions	Queries the partition files of a table.	SELECT * FROM my_table$partitions;

Global System Table

Global system tables contain the statistical and configuration information for all tables. You can access global system table through a reference system database called **sys**.

Table 3-8 Global system tables

System Table Name	Function Description	Example
sys.all_table_options	Similar to the Options Table, but shows all the table options in all databases.	SELECT * FROM sys.all_table_options;
sys.catalog_options	Queries the options of a catalog. Options that are not displayed will use their default values.	SELECT * FROM sys.catalog_options;

3.2.7 Paimon Table Procedure Management

Paimon provides a rich set of procedures for table management, which can be invoked using the **CALL** statement.

compact

- **Function**: Compacts data files to optimize storage.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **partitions**: Partition filter. Commas (,) indicate AND, and semicolons (;) indicate OR.
  - **where**: Partition predicate (cannot be used together with **partitions**).
  - **order_strategy**: Sorting strategy (order/zorder/hilbert/none)
  - **order_by**: Columns to sort by.
  - **options**: Dynamic options.

 Copyright © Huawei Technologies Co., Ltd.

  - **partition_idle_time**: Partition idle time.
  - **compact_strategy**: Compaction strategy (full/minor)
- **Example**:

```
-- Basic compaction
CALL sys.compact(table => 'T')
-- Compaction with specific partitions
CALL sys.compact(table => 'T', partitions => 'p=0;p=1')
-- Compaction with sorting
CALL sys.compact(table => 'T', order_strategy => 'zorder', order_by => 'a,b')
-- Compaction with WHERE condition
CALL sys.compact(table => 'T', where => 'p>0 and p<3', order_strategy => 'zorder', order_by => 'a,b')
-- Minor compaction
CALL sys.compact(table => 'T', compact_strategy => 'minor')
```

compact_manifest

- **Function**: Compacts manifest files to optimize metadata storage.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **options**: Dynamic options.
- **Example**:

```
CALL sys.compact_manifest(table => 'paimon.db.table');
```

purge_files

- **Function**: Clears all data files of a table.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
- **Example**:

```
-- Risky operation: All data will be deleted.
CALL sys.purge_files(table => 'paimon.db.table');
```

expire_snapshots

- **Function**: Clears expired snapshots.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **retain_max**: Maximum number of snapshots to retain.
  - **retain_min**: Minimum number of snapshots to retain.
  - **older_than**: Deletes snapshots created before the timestamp.
  - **max_deletes**: Maximum number of snapshots that can be deleted at a time.
  - **options**: Dynamic options.
- **Example**:

```
CALL sys.expire_snapshots(table => 'default.T', retain_max => 10)
```

rollback

- **Description**: Rolls back to a specified snapshot or tag.
- **Parameters**:
  - **table** (mandatory): Target table identifier.

 Copyright © Huawei Technologies Co., Ltd.

  - **version**: Snapshot ID or tag name (discarded).
  - **snapshot**: Snapshot ID.
  - **tag**: Tag name.
- **Example**:

```
-- Roll back to a tag.
CALL sys.rollback(table => 'default.T', tag => 'tag1')
-- Roll back to a snapshot.
CALL sys.rollback(table => 'default.T', snapshot => 2)
-- Use the version parameter.
CALL sys.rollback(table => 'default.T', version => 'my_tag')
CALL sys.rollback(table => 'default.T', version => 10)
```

rollback_to_watermark

- **Function**: Rolls back to the snapshot corresponding to a specified watermark.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **watermark**: Watermark.
- **Example**:

```
CALL sys.rollback_to_watermark(table => 'default.T', watermark => 1730292023000)
```

create_tag

- **Function**: Creates a tag based on a snapshot.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **tag** (mandatory): Tag name.
  - **snapshot**: Snapshot ID.
  - **time_retained**: Tag retention period.
- **Example**:

```
-- Create a tag based on snapshot 10.
CALL sys.create_tag(table => 'default.T', tag => 'my_tag', snapshot => 10, time_retained => '1 d')
-- Create a tag based on the latest snapshot.
CALL sys.create_tag(table => 'default.T', tag => 'my_tag')
```

create_tag_from_timestamp

- **Function**: Creates a tag based on a specified timestamp.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **tag** (mandatory): Tag name.
  - **timestamp** (mandatory): Timestamp.
  - **time_retained**: Tag retention period.
- **Example**:

```
CALL sys.create_tag_from_timestamp(table => 'default.T', tag => 'my_tag', timestamp =>
1724404318750, time_retained => '1 d')
```

delete_tag

- **Function**: Deletes a tag.
- **Parameters**:

 Copyright © Huawei Technologies Co., Ltd.

  - **table** (mandatory): Target table identifier.
  - **tag** (mandatory): Tag name. Use commas (,) to separate multiple tag names.
- **Example**:
```
CALL sys.delete_tag(table => 'default.T', tag => 'my_tag')
```

rename_tag

- **Function**: Renames a tag.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **tag** (mandatory): Original tag name.
  - **target_tag** (mandatory): New tag name.
- **Example**:
```
CALL sys.rename_tag(table => 'default.T', tag => 'tag1', target_tag => 'tag2')
```

replace_tag

- **Function**: Replaces the information of an existing tag.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **tag** (mandatory): Tag name.
  - **snapshot**: Snapshot ID.
  - **time_retained**: Retention period.
- **Example**:
```
CALL sys.replace_tag(table => 'default.T', tag => 'tag1', snapshot => 10, time_retained => '1 d')
```

expire_tags

- **Function**: Expires tags older than a specified time.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **older_than** (mandatory): Expiration time threshold.
- **Example**:
```
CALL sys.expire_tags(table => 'default.T', older_than => '2024-09-06 11:00:00')
```

create_branch

- **Function**: Creates a branch.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **branch** (mandatory): Branch name.
  - **tag**: Tag name.
- **Example**:
```
CALL sys.create_branch(table => 'test_db.T', branch => 'test_branch')
CALL sys.create_branch(table => 'test_db.T', branch => 'test_branch', tag => 'my_tag')
```

 Copyright © Huawei Technologies Co., Ltd.

delete_branch

- **Function**: Deletes a branch.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **branch** (mandatory): Branch name. Use commas (,) to separate multiple branch names.
- **Example**:
  ```
  CALL sys.delete_branch(table => 'test_db.T', branch => 'test_branch')
  ```

fast_forward

- **Function**: Fast-forwards a branch to the main branch.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **branch** (mandatory): Branch name.
- **Example**:
  ```
  CALL sys.fast_forward(table => 'test_db.T', branch => 'test_branch')
  ```

expire_partitions

- **Function**: Clears expired partitions.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **expiration_time**: Expiration time.
  - **timestamp_formatter**: Timestamp formatter.
  - **timestamp_pattern**: Timestamp pattern.
  - **expire_strategy**: Expiration strategy (values-time/update-time).
  - **max_expires**: Maximum number of expired records.
  - **options**: Dynamic options.
- **Example**:
  ```
  CALL sys.expire_partitions(table => 'default.T', expiration_time => '1 d', timestamp_formatter => 'yyyy-MM-dd', timestamp_pattern => '$dt', expire_strategy => 'values-time')
  ```

mark_partition_done

- **Description**: Marks a partition as done.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **partitions** (mandatory): Partitions (separated by semicolons).
- **Example**:
  ```
  -- Mark a single partition as done.
  CALL sys.mark_partition_done(table => 'default.T', partitions => 'day=2024-07-01')
  -- Mark multiple partitions as done.
  CALL sys.mark_partition_done(table => 'default.T', partitions => 'day=2024-07-01;day=2024-07-02')
  ```

remove_orphan_files

- **Function**: Deletes orphan files.

 Copyright © Huawei Technologies Co., Ltd.

- **Parameters**:
  - **table** (mandatory): Target table identifier (wildcard pattern **db.*** is supported)
  - **older_than**: Deletes orphan files generated before the timestamp.
  - **dry_run**: Only views orphan files and does not delete them.
  - **parallelism**: Degree of parallelism.
  - mode: Execution mode (local/distributed).
- **Example**:

```
CALL sys.remove_orphan_files(table => 'default.T', older_than => '2023-10-31 12:00:00')
CALL sys.remove_orphan_files(table => 'default.*', older_than => '2023-10-31 12:00:00')
CALL sys.remove_orphan_files(table => 'default.T', older_than => '2023-10-31 12:00:00', dry_run =>
true)
```

remove_unexisting_files

- **Function**: Deletes files that are referenced in the manifest but do not exist.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **dry_run**: Only views the files and does not delete them.
  - **parallelism**: Degree of parallelism.
- **Example**:

```
CALL sys.remove_unexisting_files(table => 'mydb.myt')
CALL sys.remove_unexisting_files(table => 'mydb.myt', dry_run => true)
```

reset_consumer

- **Function**: Resets or deletes a consumer.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - **consumerId** (mandatory): Consumer ID.
  - **nextSnapshotId**: ID of the next snapshot.
- **Example**:

```
-- Reset the consumer snapshot ID.
CALL sys.reset_consumer(table => 'default.T', consumerId => 'myid', nextSnapshotId => 10)
-- Delete a consumer.
CALL sys.reset_consumer(table => 'default.T', consumerId => 'myid')
```

clear_consumers

- **Function**: Clears consumers.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
  - includingConsumers: Consumers to be included (regular expressions are supported).
  - excludingConsumers: Consumers to be excluded (regular expressions are supported).
- **Example**:

```
-- Clear all consumers.
CALL sys.clear_consumers(table => 'default.T')
-- Clear specified consumers.
CALL sys.clear_consumers(table => 'default.T', includingConsumers => 'myid.*')
```

 Copyright © Huawei Technologies Co., Ltd.

```
-- Exclude specified consumers.
CALL sys.clear_consumers(table => 'default.T', includingConsumers => '', excludingConsumers =>
'myid1.*')
```

refresh_object_table

- **Function**: Refreshes an object table.
- **Parameters**:
  - **table** (mandatory): Target table identifier.
- **Example**:

```
CALL sys.refresh_object_table('default.T')
```

3.3 Paimon on Flink

3.3.1 Preparations for Using Paimon

Paimon is a data lake storage format that integrates streaming and batch processing. It supports real-time updates and streaming consumption of primary key tables, making it ideal for scenarios where real-time data warehouses and data lakes are integrated. AI DataLake Flink integrates with Huawei Cloud LakeFormation through Paimon Catalog to implement unified metadata management and direct access to Paimon tables. Before using Paimon, you need to create a LakeFormation instance, plan metadata, configure the workspace-level catalog, and ensure that the underlying file system (such as OBS) is available.

Preparations

- **A LakeFormation instance has been created with metadata planned.**

  Ensure that LakeFormation catalog is located in the same region as the AI Data Lake workspace. Otherwise, they cannot be associated in subsequent processes.
- A workspace-level LakeFormation catalog has been configured in AI DataLake.

3.3.2 Managing Paimon Catalogs

After configuring a Paimon catalog for an AI DataLake Flink job, you can use AI DataLake Flink to directly access Paimon tables in Huawei Cloud LakeFormation. This section describes how to create a Paimon catalog in an AI DataLake Flink job and manage Paimon databases and tables.

Operation method: Use a tool such as Postman to call the API for **Creating a Flink Job** to submit a Flink job.

Creating a Paimon Catalog

```
CREATE CATALOG lf_paimon_catalog
WITH
  (
    'type' = 'paimon',
    'metastore' = 'lakeformation'
  );
```

 Copyright © Huawei Technologies Co., Ltd.

Configuration parameters

Parameter	Description	Mandatory (Yes/No)	Remarks
type	Catalog type.	Yes	The value is fixed to **paimon**.
metastore	Catalog implementation type.	Yes	The value is fixed to **lakeformation**.
lakeformation.catalog-name	Name of the catalog in LakeFormation to be accessed. The name must be the same as the name of the catalog created in LakeFormation.	No	The default value is the catalog name on the Flink engine side, that is, **lf_paimon_catalog**.

Managing Paimon Databases

- **Creating a database**

```
CREATE DATABASE IF NOT EXISTS lf_paimon_catalog.paimon_db
WITH (
  'location' = 'obs://bucket/path/paimon-warehouse/paimon_db'
);
```

- **Deleting a database**

```
DROP DATABASE lf_paimon_catalog.paimon_db; -- Only deletes databases that do not contain any tables.
```

Managing Paimon Tables

- Creating a table

  Create a partitioned table in the **paimon_db** database under the **lf_paimon_catalog** catalog. The sample code is as follows:

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
  id BIGINT,
  data STRING,
  PRIMARY KEY (`id`,data) NOT ENFORCED
) PARTITIONED BY (data);
```

- Modifying the table structure

 Copyright © Huawei Technologies Co., Ltd.

Operation	Code Example
Modifying/ Adding table properties	ALTER TABLE my_table SET ('write-buffer-size' = '256 MB');
Removing table properties	ALTER TABLE my_table RESET ('write-buffer-size');
Modifying/ Adding table comments	ALTER TABLE my_table SET ('comment' = 'table comment');
Removing table comments	ALTER TABLE my_table RESET ('comment');

Temporarily modifying table parameters	You can temporarily modify table parameters by adding an SQL hint after the table name when writing data to the table. These temporary modifications are only effective for the current SQL job.
	• Temporarily set **upsert-enabled** to **true** when writing to the **my_table** table.
	`INSERT INTO tableName /*+ OPTIONS('upsert-enabled'='true') */SELECT ...;`
	• Temporarily set **monitor-interval** to **10s** when consuming the **my_table** table.
	`SELECT * FROM my_table /*+ OPTIONS('monitor-interval'='10s') */`

Renaming a table	Rename the table from **my_table** to **my_table_new**. Only EXTERNAL tables can be renamed. `ALTER TABLE my_table RENAME TO my_table_new;`
Adding a column	Add the **c1** column of the INT type and the **c2** column of the STRING type to the end of the **my_table** table. `ALTER TABLE my_table ADD (c1 INT, c2 STRING);`
Renaming a column	Rename the **c0** column in the **my_table** table to **c1**. `ALTER TABLE my_table RENAME c0 TO c1;`
Dropping a column	Drop columns **c1** and **c2** from the **my_table** table. `ALTER TABLE my_table DROP (c1, c2);`

 Copyright © Huawei Technologies Co., Ltd.

Operation	Code Example
Modifying column comments	Change the comment of the **buy_count** column in the **my_table** table to **this is buy count**. `ALTER TABLE my_table MODIFY buy_count BIGINT COMMENT 'this is buy count';`

Modifying the column order	• Move the **col_a** column of the DOUBLE type in the **my_table** table to the beginning.
	`ALTER TABLE my_table MODIFY col_a DOUBLE FIRST;`
	• Move the **col_a** column of the DOUBLE type in the **my_table** table to the end of the **col_b** column.
	`ALTER TABLE my_table MODIFY col_a DOUBLE AFTER col_b;`

Modifying the column type	ALTER TABLE my_table MODIFY col_a DOUBLE;

- Deleting a table
  `DROP TABLE lf_paimon_catalog.paimon_db.paimon_table;`

3.3.3 Paimon Configuration and Data Types

WITH Parameters

This section describes how to configure parameters for the Flink connector to read and write Paimon tables. By configuring these parameters, you can control the connection and read/write behavior between Flink and Paimon table formats to meet data read/write requirements for various service scenarios.

- **Source parameters**

 Copyright © Huawei Technologies Co., Ltd.

Table 3-9 Source parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
scan.mode	No	String	default	default/latest-full/latest/compacted-full/from-snapshot/from-snapshot-full/from-timestamp/from-creation-timestamp/incremental	Specifies the read mode. The options are: • **default**: Determined by the system automatically. • **latest-full**: Emits the latest snapshot first, then continuously reads changes. • **latest**: Continuously reads only the latest changes (ignores historical changes). • **compacted-full**: Emits the latest compacted snapshot. • **from-snapshot**: Starts from a specified snapshot (excluding the initial snapshot). • **from-snapshot-full**: Starts from a specified snapshot (including the initial snapshot). • **from-timestamp**: Starts from a specified timestamp. • **from-creation-timestamp**: Starts from a specified creation timestamp. • **incremental**: Reads changes between two snapshots incrementally. The Flink runtime determines the source data scanning policy and starting position based on this mode, which affects the range of initial read data.
scan.snapshot-id	No	Long	None	Positive integers	Specifies the starting snapshot ID. This parameter is valid only when **scan.mode** is set to **from-snapshot**. The Flink runtime starts reading data from the table status corresponding to the snapshot. If the snapshot ID does not exist, the job fails to start.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
scan.timestamp-millis	No	Long	None	Timestamps in milliseconds	Specifies the starting timestamp. This parameter is valid only when **scan.mode** is set to **from-timestamp**. The Flink runtime starts reading data from the snapshot corresponding to the timestamp and automatically selects the latest snapshot not later than the timestamp as the starting point.
scan.bounded.watermark-interval	No	Duration	None	Positive integers	Specifies the watermark interval for bounded reads. It is used to control the watermark advancement in bounded stream mode. The Flink runtime advances the watermark at this interval. When the watermark exceeds the data range, the source automatically terminates.
lookup.cache.max-rows	No	Integer	None	Positive integers	Specifies the maximum number of rows in the dimension table lookup cache. This parameter applies only to the Lookup JOIN scenarios. The Flink runtime maintains an LRU cache in memory. Once the cached rows reach the upper limit, the least recently accessed records are eliminated. Increasing this value improves the query hit ratio but consumes more memory.
lookup.cache.ttl	No	Duration	None	Positive integers + Time unit	Specifies the time to live (TTL) for the dimension table lookup cache. This parameter applies only to the Lookup JOIN scenarios. The Flink runtime invalidates cached records after they exceed the TTL, and reloads them from the Paimon storage upon next access. This balances data timeliness and query performance.

- **Sink parameters**

 Copyright © Huawei Technologies Co., Ltd.

Table 3-10 Sink parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
write-mode	No	String	auto	auto/append-only/change-log	Specifies the write mode. The options are: • **auto**: Automatically selects a mode based on presence of a primary key. • **append-only**: Appends data (no primary key required). • **change-log**: Writes change logs (a primary key table is required). Flink determines the writer type and data processing mode based on this value. In **auto** mode, it automatically uses **change-log** if there is a primary key, otherwise **append-only**.
merge-engine	No	String	deduplicate	deduplicate/partial-update/aggregation/first-row	Specifies the merge engine type. This parameter only applies to primary key tables. • **deduplicate**: Retains only the latest records. • **partial-update**: Updates partial fields (NULL values do not overwrite non-NULL values). • **aggregation**: Updates data based on aggregation functions. • **first-row**: Retains the first row after deduplication (DELETE and UPDATE_BEFORE records are not supported). The Flink runtime determines the merge policy for records with the same primary key based on this engine, which affects the final written data. For streaming queries with **partial-update** and **aggregation**, the changelog producers of **lookup** or **full-compaction** are required.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
write-buffer-size	No	MemorySize	256 mb	Positive integers	Sets the size of the write buffer. The Flink runtime buffers write data in memory and flushes it to disk when the buffer is full. Increasing this value reduces the flushing frequency and improves the write throughput, but increases memory usage.
write.max-writers-per-task	No	Integer	1	Positive integers	Sets the maximum number of writers per task. The Flink runtime creates the specified number of writers in each sink subtask for parallel writes. A larger value can improve single-task write parallelism but increase memory and file handle consumption.
changelog-producer	No	String	none	none/input/lookup/full-compaction	Specifies the changelog producer mode. When enabled, complete changelogs are generated for downstream streaming consumption. **none** indicates that no changelog is generated. **input** indicates that the input record is directly transferred. **lookup** indicates that a changelog is generated through lookup. **full-compaction** indicates that a changelog is generated during full compaction. The Flink runtime generates corresponding changelogs when committing data, which can be consumed incrementally by downstream sources.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
sink.parallelism	No	Integer	None	Positive integers	Specifies the parallelism of the sink operator. If this parameter is not set, the global parallelism of the job is used. The Flink runtime creates the specified number of sink subtasks based on this value. Each subtask is responsible for writing data to a subset of buckets. It is recommended to keep this value consistent with the number of buckets to avoid data skew.
compaction.optimization-enabled	No	Boolean	false	true/false	Specifies whether to enable compaction optimization. After this function is enabled, Flink automatically merges small files during the writes, reducing files and improving query performance. However, this may increase the computing overhead during writes.
snapshot.time-retained	No	Duration	1h	Positive integers	Set the snapshot retention period. The Flink runtime automatically cleans up snapshots that exceed this duration to free up storage space. A larger value retains more historical data for incremental reads and time-travel queries, but increases storage consumption.
snapshot.num-retained.min	No	Integer	10	Positive integers	Set the minimum number of snapshots to retain. Even if snapshots exceed the retention duration, at least this number of snapshots will be kept. The Flink runtime uses this value as the lower limit when cleaning up snapshots, ensuring that sufficient snapshots are available for incremental reads.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
snapshot.num-retained.max	No	Integer	None	Positive integers	Set the maximum number of snapshots to retain. After this parameter is set, the Flink runtime cleans up the earliest snapshots when the snapshot count exceeds this upper limit, preventing unlimited snapshot growth and storage consumption.

- **Sequence field parameters**

  Table 3-11 Sequence field parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
sequence.field	No	String	None	Valid column name	Specifies the sorting field for the primary key table. Flink determines the order of records based on the value of this field. A larger value indicates a newer record. If this parameter is not specified, the order is determined based on the data arrival time. This is useful for resolving out-of-order records in multi-stream update scenarios.
rowkind.field	No	String	None	Valid column name	Specifies the row type field. The field values can be **+I** (INSERT), **-U** (UPDATE_BEFORE), **+U** (UPDATE_AFTER), or **-D** (DELETE). The Flink runtime determines the change type of a record based on the value of this field, eliminating the need to rely on Flink's retraction mechanism. This applies to scenarios where change records are directly read from external systems such as Kafka.

 Copyright © Huawei Technologies Co., Ltd.

- **Lookup parameters**

  Table 3-12 Lookup parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
lookup.async	No	Boolean	false	true/false	Determines whether to enable asynchronous lookup. After this function is enabled, Flink uses an asynchronous thread pool to concurrently query Paimon storage, which significantly improves the throughput of lookup joins but increases memory consumption.
lookup.async-thread-number	No	Integer	16	Positive integers	Specifies the thread pool size for asynchronous lookup. This parameter is valid only when **lookup.async** is set to **true**. The Flink runtime uses this number of threads to concurrently execute lookup queries. A larger value improves the concurrent query capability but increases the memory and CPU usage.
lookup.cache	No	String	AUTO	AUTO / FULL / MEMORY	Specifies the lookup cache mode. • **AUTO**: Selected by the system. • **FULL**: Full cache (recommended for small tables). • **MEMORY**: Memory-only cache. The Flink runtime determines the caching policy based on this mode. In **FULL** mode, the entire table data is loaded to the cache during the initial loading, and subsequent queries directly hit the cache.

- **Bucketing parameters**

 Copyright © Huawei Technologies Co., Ltd.

Table 3-13 Bucketing parameters

Parameter	Mandatory (Yes/No)	Type	Default Value	Value Range	Description
bucket	No	Integer	-1	-2, -1, or a positive integer	Sets the number of buckets. • **-1** (default): Dynamic bucketing • **-2**: Postpone bucketing (for append-only tables) • A positive integer: A fixed number of buckets. For primary key tables using fixed bucketing, it is recommended to set this value equal to the Sink parallelism to avoid data skew. The Flink runtime distributes data into corresponding buckets based on the hash value of the bucketing key. The number of buckets affects data distribution balance and parallel read capability.
bucket-key	No	String	Primary key column	Valid column name	Specifies the bucketing key. If not specified, the primary key column is used as the bucketing key. This parameter is valid only for primary key tables. Flink performs bucket routing based on the hash value of this column. Selecting a column with uniform business data distribution as the bucketing key can improve the write and query performance.

Data Type Mapping

Flink Data Type	Paimon Data Type
BOOLEAN	BOOLEAN
TINYINT	TINYINT
SMALLINT	SMALLINT
INT	INT
BIGINT	BIGINT
FLOAT	FLOAT
DOUBLE	DOUBLE

 Copyright © Huawei Technologies Co., Ltd.

Flink Data Type	Paimon Data Type
DECIMAL	DECIMAL
STRING	STRING
CHAR	CHAR
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
TIMESTAMP WITH LOCAL TIMEZONE	TIMESTAMP WITH LOCAL TIMEZONE
BINARY	BINARY
BYTES	BYTES

3.3.4 Paimon on Flink DDL Syntax

CREATE TABLE

Tables created within the Paimon Catalog are managed by the Catalog. When a table is deleted from the Catalog, its table files are also deleted.

Basic table creation statement (with the primary key):

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING,
    PRIMARY KEY (dt, hh, user_id) NOT ENFORCED
);
```

Partitioned table creation statement

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING,
    PRIMARY KEY (dt, hh, user_id) NOT ENFORCED
) PARTITIONED BY (dt, hh);
```

Statistics Mode

Paimon automatically collects statistics on data files to accelerate queries. It supports four modes:

- **full**: Collects complete metrics (**null_count**, **min**, and **max**).
- **truncate(length)**: Collects **null_count** and truncated **min**/**max** values. The default value is **truncate(16)**, which prevents excessive column size from bloating manifest files.

 Copyright © Huawei Technologies Co., Ltd.

- **counts**: Collects only **null_count**.
- **none**: Disables metadata statistics collection.

The statistics collection mode is configured using **metadata.stats-mode**. The default value is **truncate(16)**. You can configure it by field using **fields.{field_name}.stats-mode**.

In **none** mode, **metadata.stats-dense-store** defaults to **true**. This significantly reduces the manifest storage size but requires Paimon SDK 0.9.1 or later.

Field Default Value

Paimon supports setting default values for fields by configuring **fields.{field_name}.default-value** in the table properties. Note that default values cannot be set for partition fields and primary key fields.

CREATE TABLE AS SELECT (CTAS)

CTAS allows you to create and populate a table based on query results. For example, the statement **CREATE TABLE table_b AS SELECT id, name FROM table_a** is equivalent to creating a table and then inserting data into it.

```
/* Streaming mode requires checkpointing to be enabled. */
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT
);
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_as AS SELECT * FROM
lf_paimon_catalog.paimon_db.paimon_table;
```

Partitioned table CTAS

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING
) PARTITIONED BY (dt, hh);
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_as WITH ('partition' = 'dt') AS SELECT * FROM
lf_paimon_catalog.paimon_db.paimon_table;
```

CTAS with modified options

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT
) WITH ('file.format' = 'orc');
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_as WITH ('file.format' = 'parquet') AS SELECT *
FROM lf_paimon_catalog.paimon_db.paimon_table;
```

Primary key CTAS

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING,
    PRIMARY KEY (dt, hh, user_id) NOT ENFORCED
);
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_as WITH ('primary-key' = 'dt,hh') AS SELECT *
FROM lf_paimon_catalog.paimon_db.paimon_table;
```

Primary key + partitioned CTAS

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING,
    PRIMARY KEY (dt, hh, user_id) NOT ENFORCED
) PARTITIONED BY (dt, hh);
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_as WITH ('primary-key' = 'dt,hh', 'partition' =
'dt') AS SELECT * FROM lf_paimon_catalog.paimon_db.paimon_table;
```

CREATE TABLE LIKE

This statement creates a new table with the same schema, partitioning, and table properties as an existing table. *Using **EXCLUDING OPTIONS** copies the schema but excludes the table properties.*

```
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table (
    user_id BIGINT,
    item_id BIGINT,
    behavior STRING,
    dt STRING,
    hh STRING,
    PRIMARY KEY (dt, hh, user_id) NOT ENFORCED
);
CREATE TABLE lf_paimon_catalog.paimon_db.paimon_table_like LIKE
lf_paimon_catalog.paimon_db.paimon_table (EXCLUDING OPTIONS);
```

3.3.5 Querying and Writing Data to a Paimon Table

Important Notes

- The underlying file system (such as OBS) must be available.
- PRIMARY KEY must be defined for a primary key table, and the primary key column cannot be NULL.
- Checkpointing must be enabled for streaming writes to Paimon. Data is committed only when a checkpoint is triggered.
- For primary key tables, it is recommended that the number of buckets matches the sink parallelism to avoid data skew.
- The partition key of Paimon must be included in the primary key.
- Flink streaming jobs do not support INSERT OVERWRITE.
- Once a Paimon table is managed via a Catalog, renaming the table is not allowed.
- Paimon does not support Sink materializer. Please set **table.exec.sink.upsert-materialize** to **NONE**.

Code Examples

- **Example 1: Append data without a primary key to a Paimon append table.**

```
CREATE CATALOG lf_paimon_catalog
WITH
  (
    'type' = 'paimon',
    'metastore' = 'lakeformation'
  );

CREATE DATABASE IF NOT EXISTS lf_paimon_catalog.paimon_db
WITH (
```

 Copyright © Huawei Technologies Co., Ltd.

```
  'location' = 'obs://bucket/path/paimon-warehouse/paimon_db'
);

CREATE TABLE IF NOT EXISTS lf_paimon_catalog.paimon_db.paimon_table (
  id BIGINT,
  customer_id STRING,
  amount DOUBLE,
  region STRING
);

CREATE TABLE kafka_source(
  id BIGINT,
  customer_id STRING,
  amount DOUBLE,
  region STRING
)
WITH
  (
    'connector' = 'kafka',
    'topic' = 'user-events',
    'properties.bootstrap.servers' = '127.0.0.1:9092',
    'properties.group.id' = 'flink-consumer-group',
    'scan.startup.mode' = 'latest-offset',
    'format' = 'json'
  );

INSERT INTO
  lf_paimon_catalog.paimon_db.paimon_table
SELECT
  *
FROM
  kafka_source;
```

- **Example 2: Read a Paimon append table.**

```
CREATE CATALOG lf_paimon_catalog
WITH
  (
    'type' = 'paimon',
    'metastore' = 'lakeformation'
  );

CREATE TABLE sink (
  id BIGINT,
  customer_id STRING,
  amount DOUBLE,
  region STRING
)
WITH
  ('connector' = 'print');

INSERT INTO
  sink
SELECT
  *
FROM
  lf_paimon_catalog.paimon_db.paimon_table /*+ OPTIONS('streaming'='true', 'monitor-interval'='60s')
*/;
```

3.4 Paimon Development FAQs

3.4.1 How Do I Handle Concurrent Data Commit Conflicts?

Paimon uses the optimistic concurrency control (OCC) to handle concurrent writes. When multiple jobs concurrently write data to the same Paimon table, commit

 Copyright © Huawei Technologies Co., Ltd.

conflicts may occur. Paimon detects these conflicts via its snapshot mechanism and retries the operation or reports an error.

Concurrent commit conflicts are classified into the following types:

- Data conflict: Multiple concurrent jobs modify the data in the same partition or with the same primary key, causing the job that is committed later to detect a data change conflict.
- Metadata conflict: Multiple concurrent jobs submit snapshot files simultaneously. The job that is committed later finds that the snapshot version has been updated by another job, resulting in a metadata conflict.

Conflict Handling Strategies

Paimon adopts different strategies based on the type of conflict and table scheme.

- For concurrent append operations on an append-only table, Paimon automatically retries the commit. It merges the data from multiple appends and re-commits them.
- For primary key tables, if concurrent writes target different primary keys, Paimon automatically retries and merges the data. If concurrent writes target the same primary key, the later committed jobs will fail and an error will be reported.
- In scenarios where compaction and DML operations are executed concurrently, Paimon supports asynchronous compaction. Compaction does not block DML operations, but there may be temporary resource contention.

Suggestions

- When performing concurrent DML operations on the same partition, it is recommended that you specify partition conditions in the **WHERE** clause to reduce the scope of potential conflicts.
- In high-frequency concurrent write scenarios, you can use bucketing to distribute data evenly across different buckets. This reduces the probability of different jobs writing to the same data file.
- Set the **sink.commit.max-retries** parameter properly to increase the number of commit retries and improve the success rate of concurrent writes.

3.4.2 How Do I Handle Excessive Small Files?

Frequent INSERT, UPDATE, and DELETE operations on Paimon tables generate a large number of small files, which negatively impacts query performance and storage utilization.

Excessive small file are mainly caused by the following issues:

- Queries need to open and read a large number of small files.
- Too many manifest files cause metadata bloat.

Solutions

Paimon provides the following mechanisms to handle excessive small files:

 Copyright © Huawei Technologies Co., Ltd.

- Synchronous compaction: Automatically triggers compaction during data writing to merge small files into large ones. You can enable this function by setting **write.auto-compaction** to **true**.
- Asynchronous compaction: Triggers background asynchronous compaction tasks through Flink or Spark to merge small files without affecting write performance. You can adjust the trigger interval of asynchronous compaction by setting the **compaction.optimization-interval** parameter.
- Write parameter tuning: Properly sets the **target-file-size** parameter to control the target file size, and sets the **compaction.min.file-num** and **compaction.max.file-num** parameters to control the triggering conditions and merging scope of the compaction.
- Sorted Run merging: For primary key tables, Paimon reduces the file count by merging sorted runs based on the LSM-Tree structure. You can control the number of sorted runs that trigger a merge using the **num-sorted-run.compaction-trigger** parameter.

Suggestions

- In streaming write scenarios, you are advised to enable asynchronous compaction to avoid the performance penalty associated with synchronous compaction.
- Regularly monitor the number of files and average file size in the table. Trigger compaction promptly if the file count grows abnormally.
- Configure compaction parameters appropriately to avoid resource waste and increased write latency caused by excessive compactions.

3.4.3 How Do I Achieve Data Compatibility After Schema Evolution?

Paimon uses field IDs to ensure data compatibility after schema evolution. Each field is assigned a unique integer ID when it is created. This ID is persisted in the schema of the data file. Even if the column name or type changes, historical data files can still be correctly read using the field ID.

Important Notes

- **Uniqueness of field IDs**: The ID of each field is unique throughout the table's lifecycle. After a field is deleted, its ID will not be reused. This ensures that historical data files can be correctly parsed.
- **Type change restrictions**: Only upgrades from lower-precision types to higher-precision types (such as from INT to BIGINT or from FLOAT to DOUBLE) are supported. Incompatible type conversions (such as from STRING to INT) are not allowed.
- **Files written by non-Paimon engines**: Data files written by other engines (such as Hive and Spark) do not contain field IDs in their schemas. When these files are read, column names are used for mapping. If column names are reused after schema evolution, it may lead to type mismatches or reading incorrect historical data.
- **Nested type evolution**: For nested types such as **struct**, **list**, and **map**, the evolution of internal fields also follows the field ID mechanism. This ensures compatibility for historical data within nested structures.

 Copyright © Huawei Technologies Co., Ltd.

Suggestions

- Avoid frequent modifications to the table structure, as each schema evolution increases the complexity of metadata.
- When deleting a column, it is recommended to use soft deletion (marking the column as unused instead of physically deleting it). This preserves the field ID for historical data backtracking.
- For columns that require data backfilling, you are advised to use compactions to populate the new column values in newly written files. This avoids the need for a full table rewriting.

3.4.4 How Do I Maintain Tables?

Routine maintenance of Paimon tables is crucial for ensuring query performance and storage utilization. Key maintenance tasks include clearing expired snapshots, merging small files, and rewriting manifest files.

Clearing Expired Snapshots

Each batch written to the table generates a new snapshot. Paimon tracks these snapshots in the table metadata until they expire. Frequent commits may lead to snapshot stacking, so routine maintenance is required, especially for streaming tables.

- **snapshot.num-retained.max**: Controls the maximum number of snapshots to retain. The default value is **10**.
- **snapshot.num-retained.min**: Controls the minimum number of snapshots to retain. The default value is **3**.
- **snapshot.time-retained**: Controls the time range for retaining snapshots. Snapshots older than this range will be automatically deleted.

Merging Small Files

Streaming writes often result in small data batches, leading to a large number of small files. Compaction merges small files into larger ones, reducing the number of manifest files and optimizing query efficiency.

- Synchronous compaction: Manually triggered using the ALTER TABLE statement or automatically triggered during data writes.
- Asynchronous compaction: Triggered by submitting asynchronous compaction tasks via Flink or Spark to merge small files in the background.

Rewriting Manifest Files

Paimon continuously generates new manifest files during the write process. To optimize query planning performance, you can merge multiple small manifest files into large ones.

- Perform the compaction to indirectly trigger manifest file merging.
- Regularly perform table maintenance to keep the number of manifest files within a reasonable range.

 Copyright © Huawei Technologies Co., Ltd.

Suggestions

- Suggestion 1: Regularly delete expired snapshots to prevent metadata bloat caused by an excessive number of snapshots.
- Suggestion 2: Periodically perform the compaction to merge small files and optimize query performance.
- Suggestion 3: Create tags for critical data versions to facilitate quick rollback to a specific version.
- Suggestion 4: Monitor the number of files and storage size of tables to detect and resolve storage issues promptly.

Copyright © Huawei Technologies Co., Ltd.

4 Spark UDF Development Guide

4.1 Calling UDFs in Spark SQL Jobs

Scenario

AI DataLake allows you to use Hive user-defined functions (UDFs) to query data. UDFs take effect only on a single row of data and are applicable to inserting and deleting a single data record.

Constraints

If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

Environment Preparations

Before you start, set up the development environment.

Table 4-1 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required.
IntelliJ IDEA	IntelliJ IDEA 2019.1 or a compatible version is required.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The UDF development process is as follows:

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-1 UDF development process

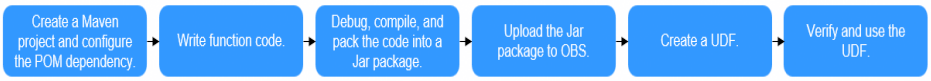


Table 4-2 Process description

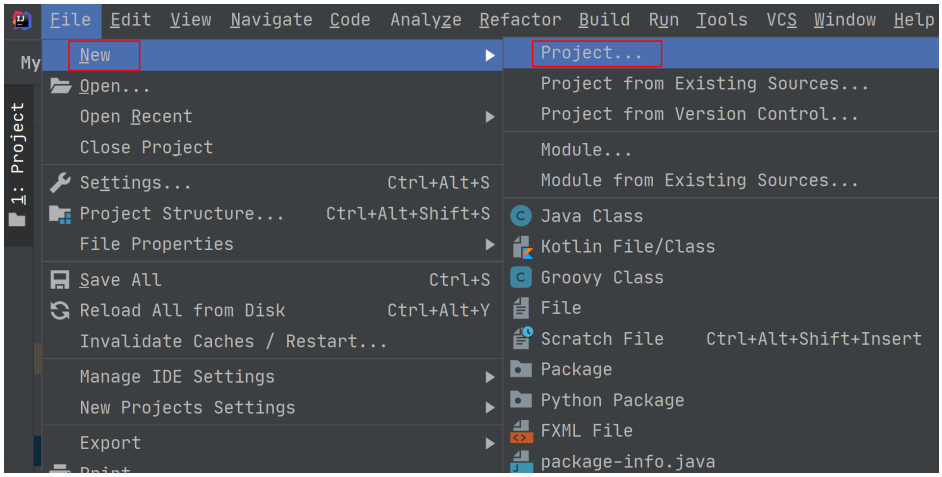
No.	Step	Operation Interface	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write UDF code by referring to steps in **Procedure**.
2	Write UDF code.		
3	Debug, compile, and pack the code into a JAR package.		
4	Upload the JAR package to OBS.	OBS console	Upload the UDF JAR file to an OBS directory.
5	Create a UDF.	DataArts Studio console	Create a UDF on the **Develop Script** page of the DataArts Studio console.
6	Verify and use the UDF.	DataArts Studio console	On the **Develop Script** page of the DataArts Studio console, use the created UDF in a Spark SQL job.

Procedure

1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

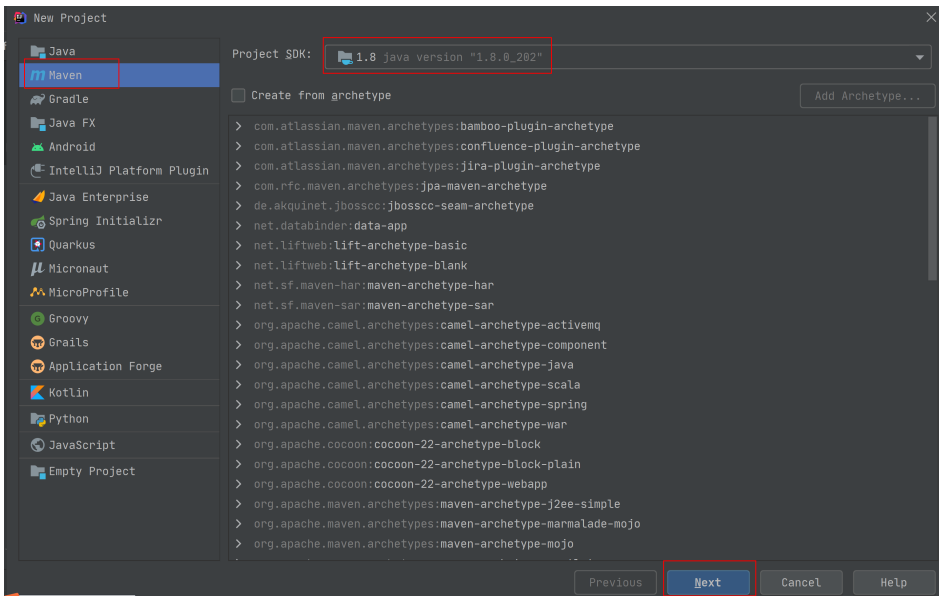
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-2 Creating a project



b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

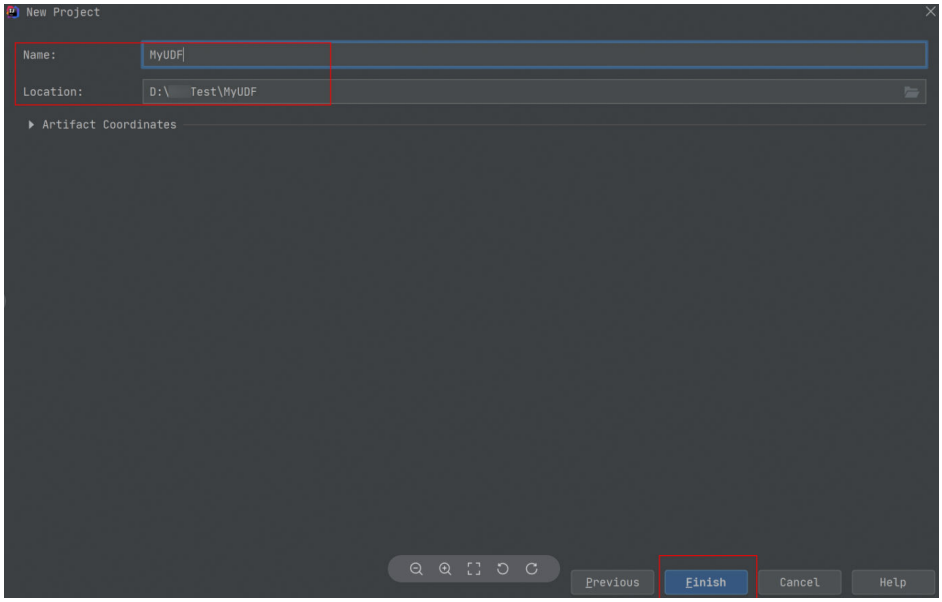
Figure 4-3 Choosing Maven



c. Set the project name, configure the storage path, and click **Finish**.

 Copyright © Huawei Technologies Co., Ltd.

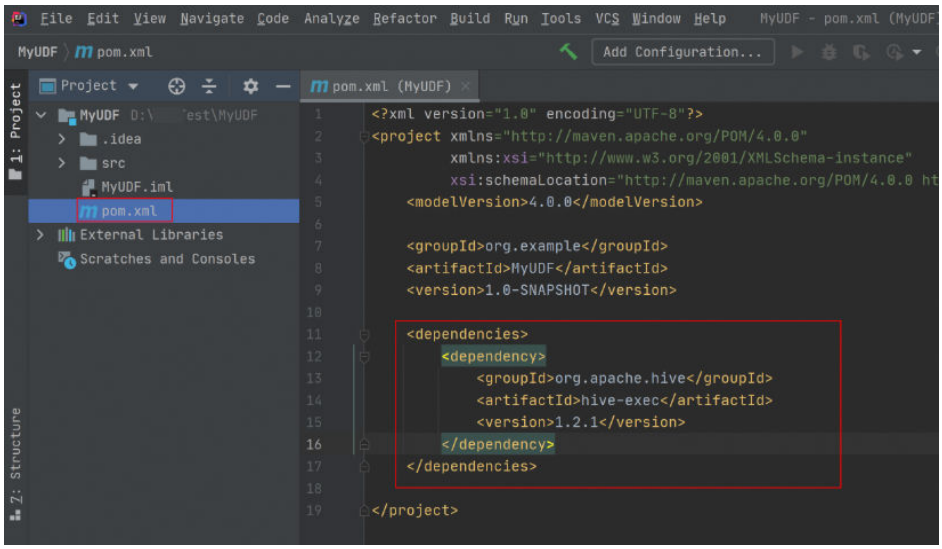
Figure 4-4 Creating a project



d. Add the following content to the **pom.xml** file.

```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

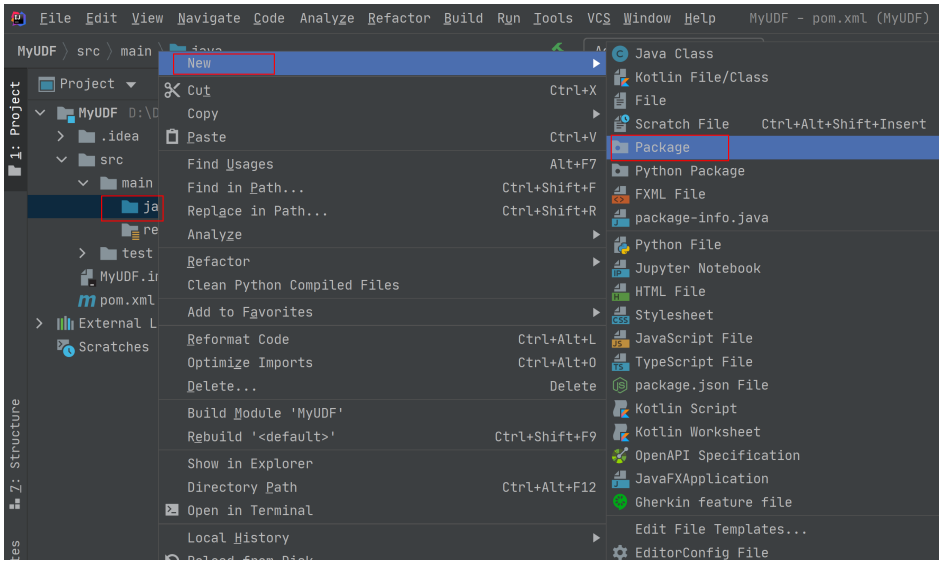
Figure 4-5 Adding configurations to the POM file



e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

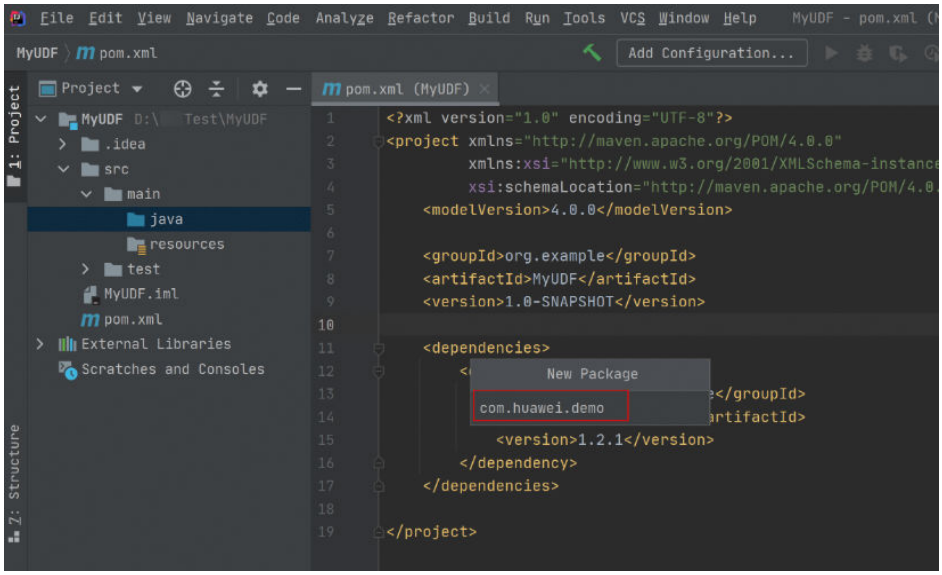
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-6 Creating a package and a class file



Set the package name as you need. In this example, set **Package** to **com.huawei.demo**. Then, press **Enter**.

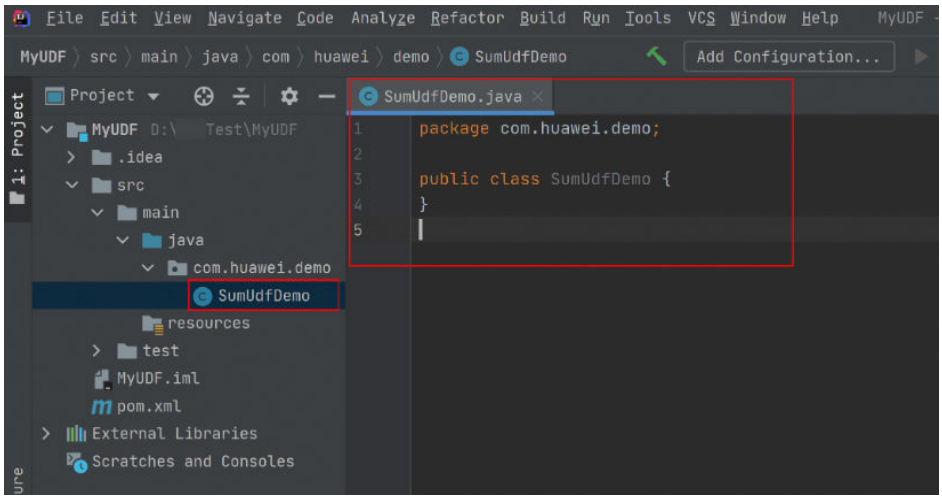
Figure 4-7 Customizing a package



Create a Java class file in the package path. In this example, the Java Class file is **SumUdfDemo**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-8 Creating a Java class file



2. Write UDF code. Pay attention to the following requirements when you implement the UDF:

   a. The UDF must inherit **org.apache.hadoop.hive.ql.exec.UDF**.

   b. The **evaluate** function must be implemented, which can be reloaded.

   For details about how to implement the UDF, see the following sample code:

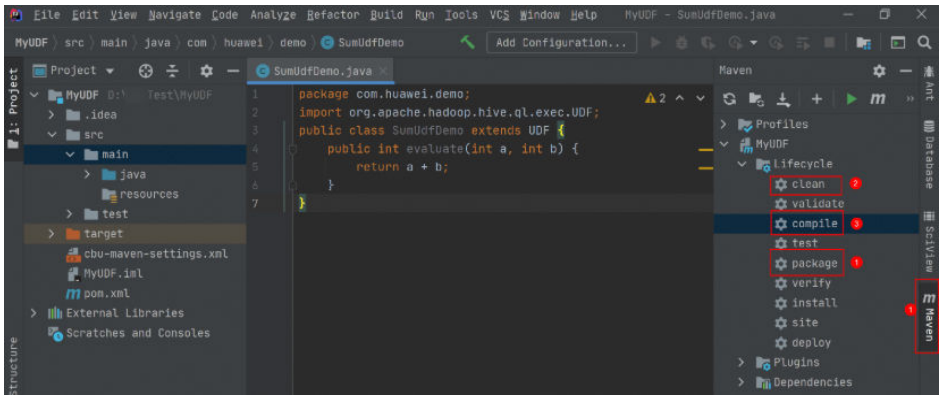
```
package com.huawei.demo;
import org.apache.hadoop.hive.ql.exec.UDF;
 public class SumUdfDemo extends UDF {
   public int evaluate(int a, int b) {
    return a + b;
 }
}
```

3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.

   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

   After the compilation is successful, click **package**.

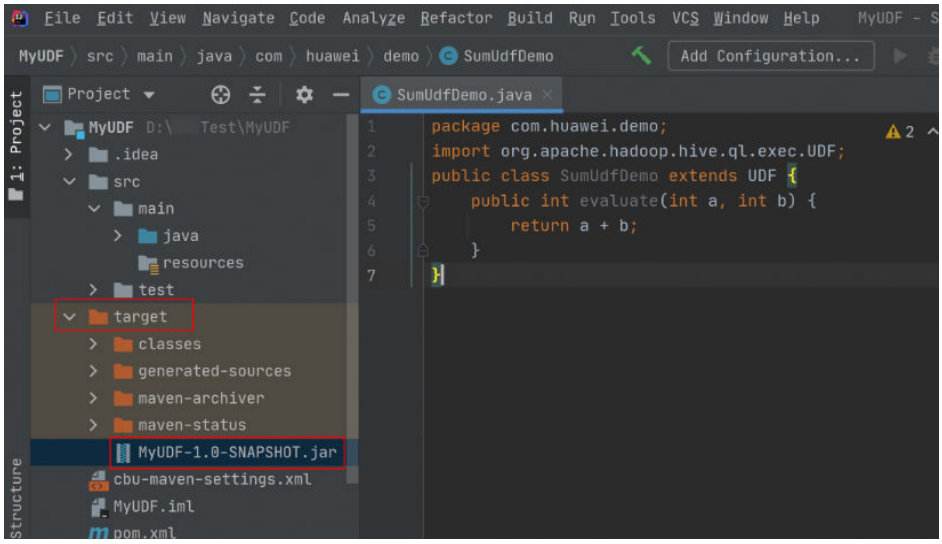
Figure 4-9 Compiling and packaging



The generated JAR package is stored in the **target** directory. In this example, the generated JAR package is named **MyUDF-1.0-SNAPSHOT.jar** and stored in the **D:\AIDataLakeTest\MyUDF\target** directory.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-10 Generating a JAR file



4. Log in to the OBS console and upload the JAR file to the OBS path.

   NOTE

   The region of the OBS bucket to which the JAR file is uploaded must be the same as the region of the AI DataLake workspace. Cross-region operations are not allowed.

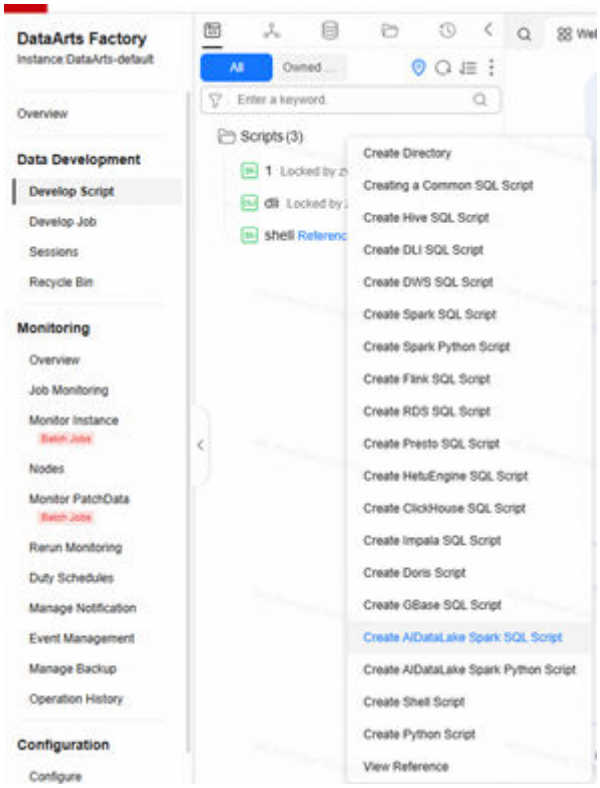
5. Create the UDF.

   a. On the **AI DataLake console**, click **DataArts Studio** in the lower left corner.

   b. On the **Develop Script** page, create a Serverless Spark SQL script and select the endpoint, catalog, and database.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-11 Creating a Serverless Spark SQL script



c. In the SQL editing area, enter the following command for creating the UDF:

```
CREATE FUNCTION TestSumUDF AS 'com.huawei.demo.SumUdfDemo' using jar 'obs://aidatalake-test-obs01/MyUDF-1.0-SNAPSHOT.jar';
```

d. Click **Execute**.

6. Call the UDF.

   Use the UDF created in **5** in the SELECT statement as follows:

   ```
   SELECT TestSumUDF(1,2);
   ```

7. (Optional) Delete the UDF.

   If the UDF is no longer used, run the following statement to delete it:

   ```
   DROP FUNCTION TestSumUDF;
   ```

4.2 Calling UDAFs in Spark SQL Jobs

Scenario

AI DataLake allows you to use a Hive User Defined Aggregation Function (UDAF) to process multiple rows of data. Hive UDAF is usually used together with groupBy. It is similar to SUM() and AVG() functions commonly used in SQL.

Constraints

If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

 Copyright © Huawei Technologies Co., Ltd.

Environment Preparations

Before you start, set up the development environment.

Table 4-3 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required. For details, visit **Java Downloads**.
IntelliJ IDEA	**IntelliJ IDEA** 2019.1 or later is required.
Maven	Basic configurations of the development environment. For details about how to get started, see **Downloading Apache Maven** and **Installing Apache Maven**. Maven is used for project management throughout the lifecycle of software development.

Development Process

The UDAF development process is as follows:

Figure 4-12 UDAF development process

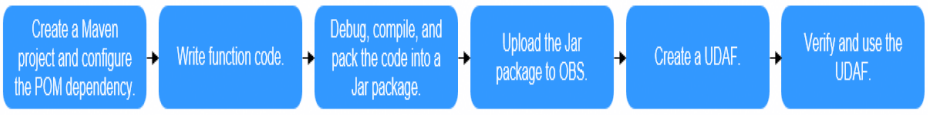


Table 4-4 Process description

No.	Step	Operation Interface	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write the UDAF code by referring to the **Procedure**.
2	Write the UDAF code.		
3	Debug, compile, and pack the code into a JAR package.		
4	Upload the JAR package to OBS.	OBS console	Upload the UDAF JAR file to an OBS path.
5	Create the UDAF.	DataArts Studio console	Create a UDAF on the **Develop Script** page of the DataArts Studio console.

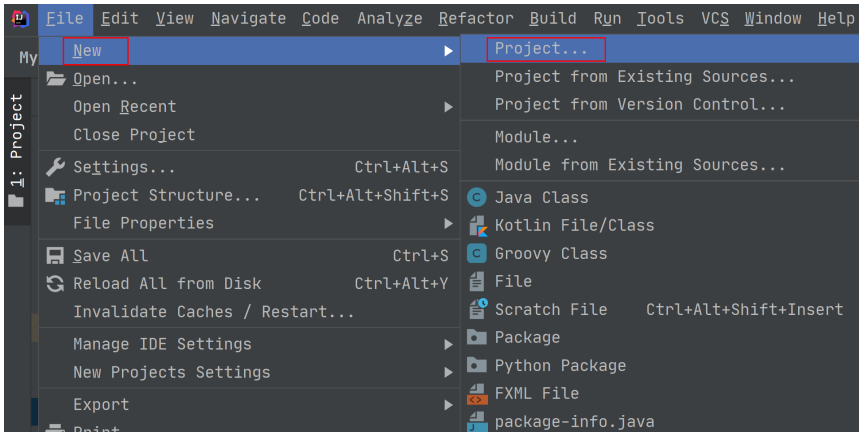
Copyright © Huawei Technologies Co., Ltd.

No.	Step	Operation Interface	Description
6	Verify and use the UDAF.	DataArts Studio console	On the **Develop Script** page of the DataArts Studio console, use the created UDAF in a Spark SQL job.

Procedure

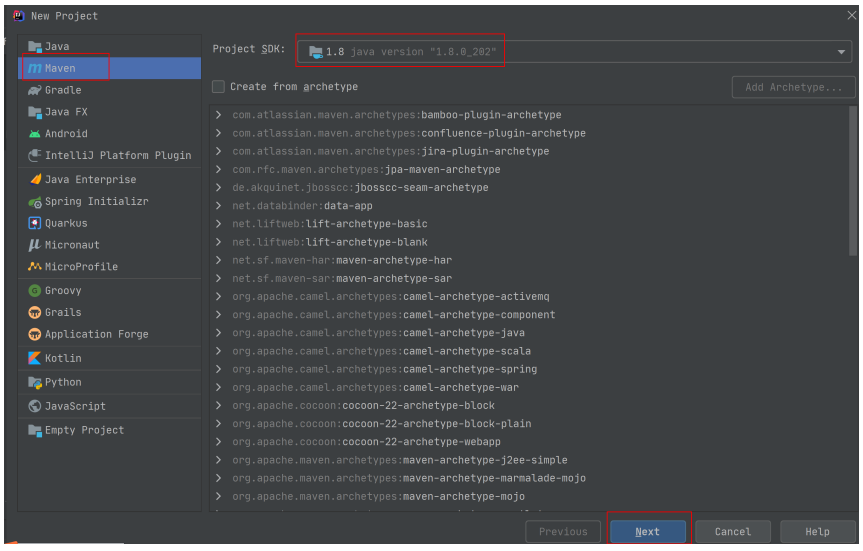
1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

   Figure 4-13 Creating a project

   b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

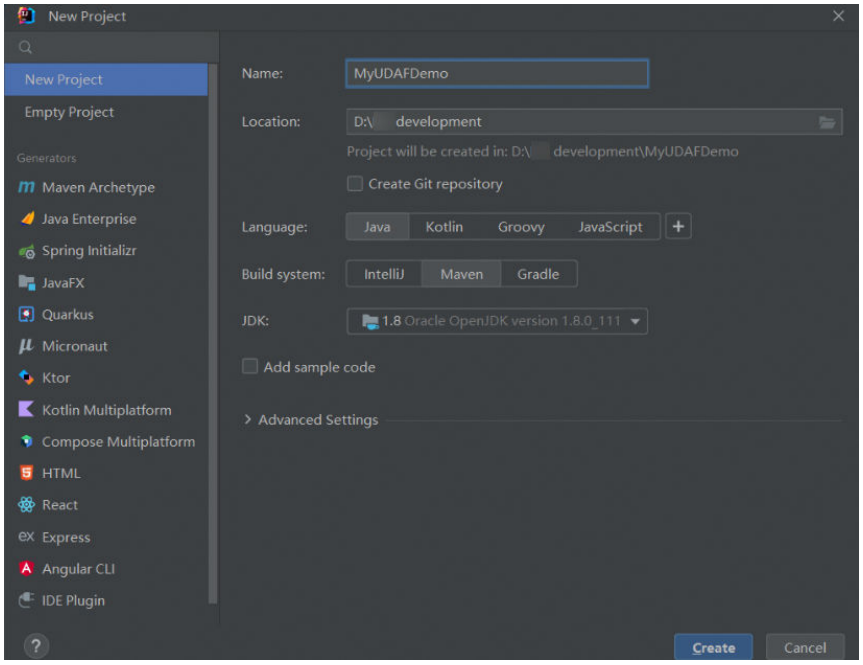
   Figure 4-14 Configuring the project SDK

   c. Specify the project name and the project path, and click **Create**. In the displayed page, click **Finish**.

 Copyright © Huawei Technologies Co., Ltd.

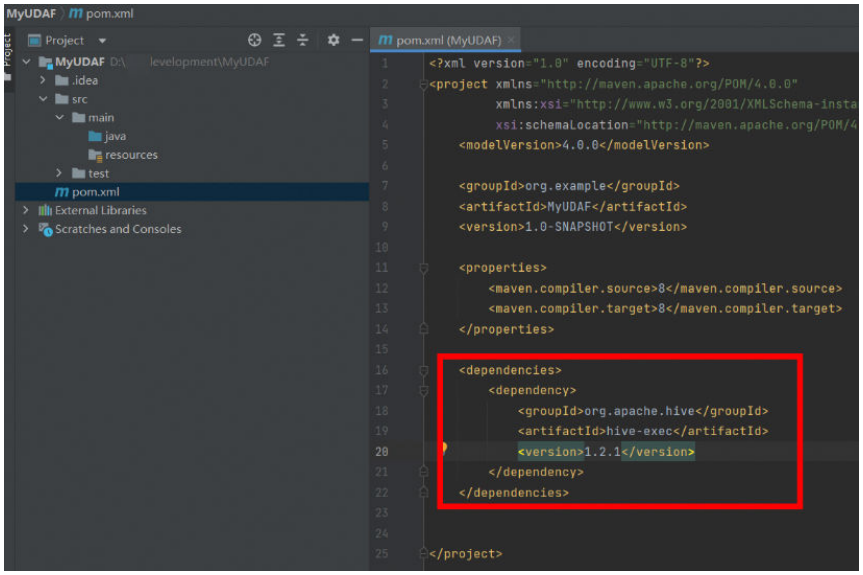
Figure 4-15 Setting project information



d. Add the following content to the **pom.xml** file.

```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

Figure 4-16 Adding configurations to the POM file

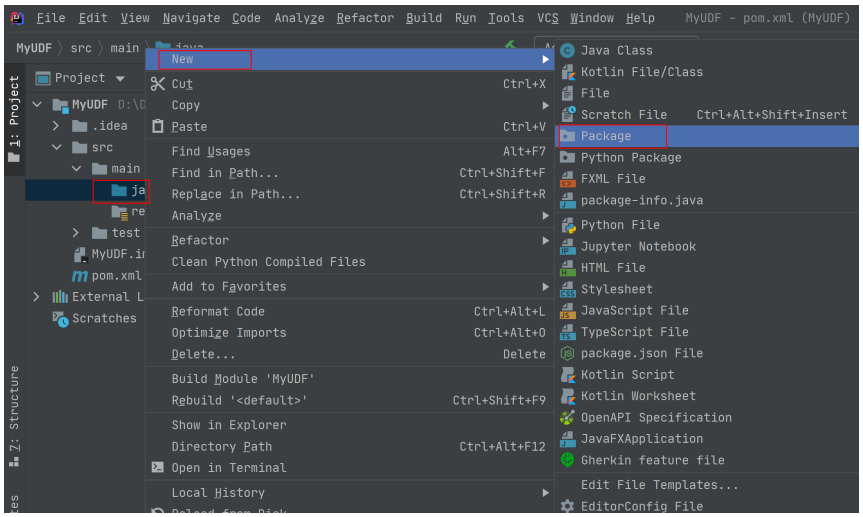


e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

Set **Package** as required. In this example, set **Package** to **com.aidatalake.demo**.

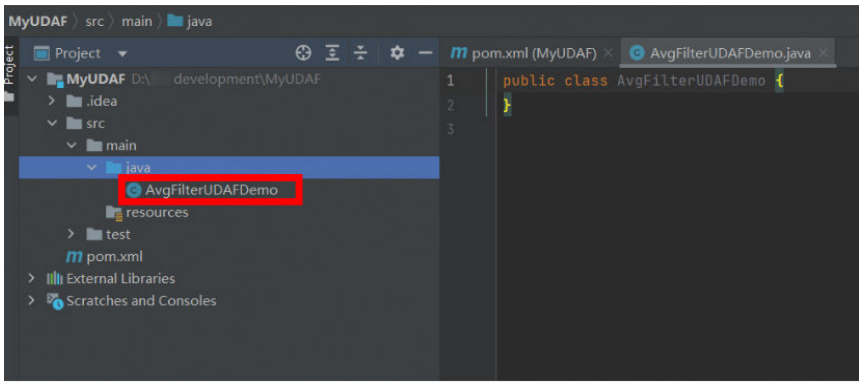
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-17 Creating a package



Create a Java class file in the package path. In this example, the Java Class file is **AvgFilterUDAFDemo**.

Figure 4-18 Creating a class



2. Write the UDAF code. Pay attention to the following requirements when you implement the UDAF:
   - The UDAF class must inherit from **org.apache.hadoop.hive.ql.exec.UDAF** and **org.apache.hadoop.hive.ql.exec.UDAFEvaluator** classes. The function class must inherit from the UDAF class, and the **Evaluator** class must implement the **UDAFEvaluator** interface.
   - The **Evaluator** class must implement the **init**, **iterate**, **terminatePartial**, **merge**, and **terminate** functions of **UDAFEvaluator**.
     - The **init** function overrides the **init** function of the **UDAFEvaluator** interface.
     - The **iterate** function receives input parameters for internal iteration.
     - The **terminatePartial** function has no parameter. It returns the data obtained after the **iterate** traversal is complete. **terminatePartial** is similar to Hadoop **Combiner**.
     - The **merge** function receives the return values of **terminatePartial**.

 Copyright © Huawei Technologies Co., Ltd.

- The **terminate** function returns the aggregated result.

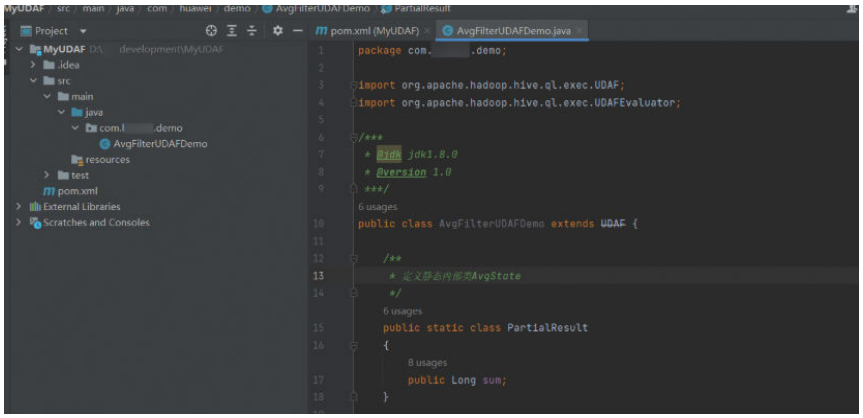
For details about how to implement the UDAF, see the following sample code:

```
package com.aidatalake.demo;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

/***
 * @jdk jdk1.8.0
 * @version 1.0
 ***/
public class AvgFilterUDAFDemo extends UDAF {

    /**
     * Defines the static inner class AvgFilter.
     */
    public static class PartialResult
    {
        public Long sum;
    }

    public static class VarianceEvaluator implements UDAFEvaluator {

        // Initializes the PartialResult object.
        private AvgFilterUDAFDemo.PartialResult partial;

        // Declares a VarianceEvaluator constructor that has no parameters.
        public VarianceEvaluator(){

            this.partial = new AvgFilterUDAFDemo.PartialResult();

            init();
        }

        /**
         * Initializes the UDAF, which is similar to a constructor.
         */
        @Override
        public void init() {

            // Sets the initial value of sum.
            this.partial.sum = 0L;
        }

        /**
         * Receives input parameters for internal iteration.
         * @param x
         * @return
         */
        public void iterate(Long x) {
            if (x == null) {
                return;
            }
            AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
            tmp9_6.sum = tmp9_6.sum | x;
        }

        /**
         * Returns the data obtained after the iterate traversal is complete.
         * terminatePartial is similar to Hadoop Combiner.
         * @return
         */
        public AvgFilterUDAFDemo.PartialResult terminatePartial()
        {
            return this.partial;
        }
```

 Copyright © Huawei Technologies Co., Ltd.

```
    /**
     * Receives the return values of terminatePartial and merges the data.
     * @param
     * @return
     */
    public void merge(AvgFilterUDAFDemo.PartialResult pr)
    {
        if (pr == null) {
            return;
        }
        AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
        tmp9_6.sum = tmp9_6.sum | pr.sum;
    }

    /**
     * Returns the aggregated result.
     * @return
     */
    public Long terminate()
    {
        if (this.partial.sum == null) {
            return 0L;
        }
        return this.partial.sum;
    }
  }
}
```

Figure 4-19 Writing UDAF code

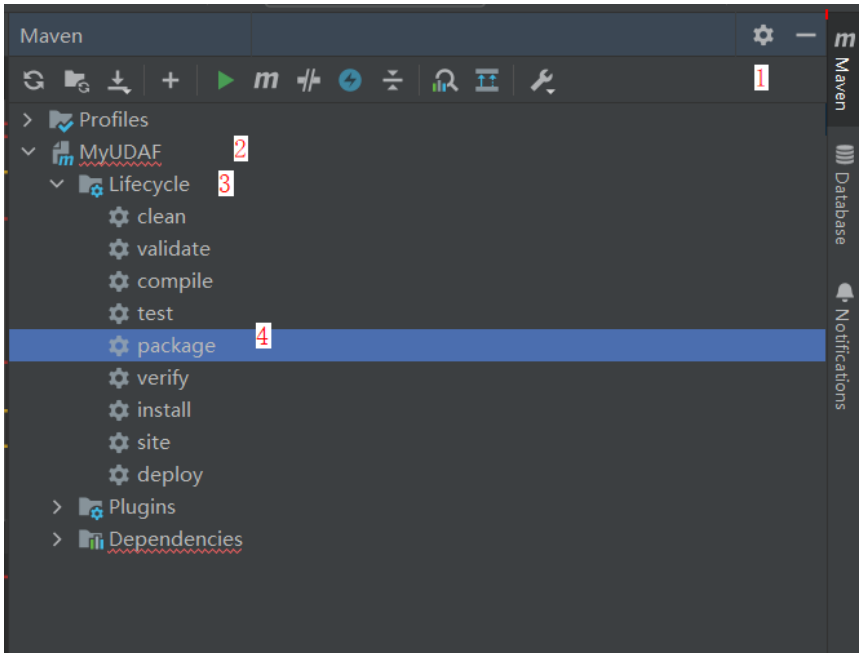


3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.
   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

      After the compilation is successful, click **package**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-20 Exporting the JAR file



b. The generated JAR package is stored in the **target** directory. In this example, the generated file is named **MyUDAF-1.0-SNAPSHOT.jar** and stored in the **D:\AIDataLakeTest\MyUDAF\target** directory.

4. Log in to the OBS console and upload the JAR file to the OBS path.

   NOTE

   The region of the OBS bucket to which the JAR file is uploaded must be the same as the region of the AI DataLake workspace. Cross-region operations are not allowed.

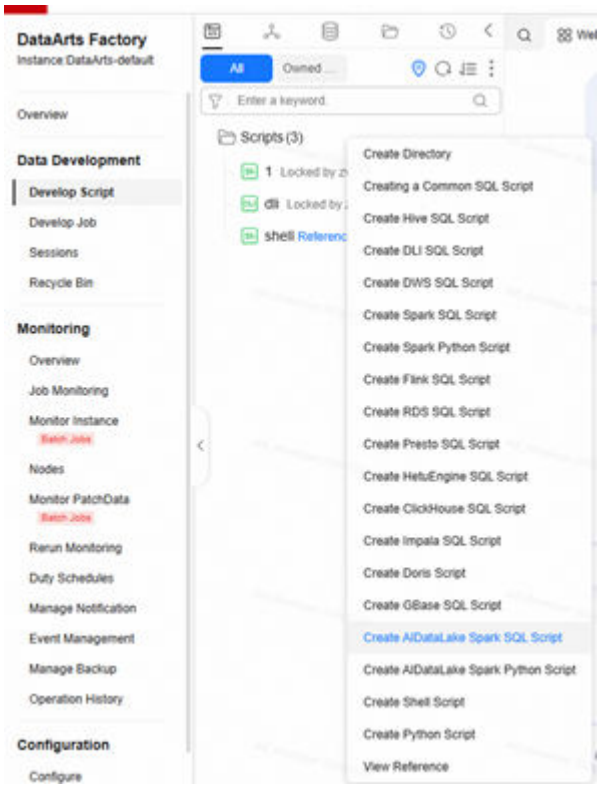
5. Create the UDAF.

   a. On the **AI DataLake console**, click **DataArts Studio** in the lower left corner.

   b. On the **Develop Script** page, create a Serverless Spark SQL script and select the endpoint, catalog, and database.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-21 Creating a Serverless Spark SQL script



c. In the SQL editing area, enter the following command for creating the UDAF:

```
CREATE FUNCTION AvgFilterUDAFDemo AS 'com.aidatalake.demo.AvgFilterUDAFDemo' using jar 'obs://aidatalake-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
```

or

```
CREATE OR REPLACE FUNCTION AvgFilterUDAFDemo AS 'com.aidatalake.demo.AvgFilterUDAFDemo' using jar 'obs://aidatalake-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
```

NOTE

If the reloading function of the UDAF is enabled, the create statement changes.

d. Click **Execute**.

6. Use the UDAF.

   Use the UDAF in the SELECT statement as follows:

   ```
   select AvgFilterUDAFDemo(real_stock_rate) AS show_rate FROM dw_ad_estimate_real_stock_rate LIMIT 1000;
   ```

7. (Optional) Delete the UDAF.

   If the UDAF is no longer used, run the following statement to delete it:

   ```
   Drop FUNCTION AvgFilterUDAFDemo;
   ```

4.3 Calling UDTFs in Spark SQL Jobs

Scenario

AI DataLake allows you to use Hive User-Defined Table-Generating Functions (UDTFs) to customize table-valued functions. UDTFs are used for one-in-multiple-

 Copyright © Huawei Technologies Co., Ltd.

out data operations, specifically, UDTF reads a row of data and outputs multiple values.

Constraints

If you use a static class or interface in a UDF, add **try catch** to capture exceptions. Otherwise, package conflicts may occur.

Environment Preparations

Before you start, set up the development environment.

Table 4-5 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required.
IntelliJ IDEA	IntelliJ IDEA 2019.1 or a compatible version is required.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

Development Process

The UDTF development process is as follows:

Figure 4-22 UDTF development process

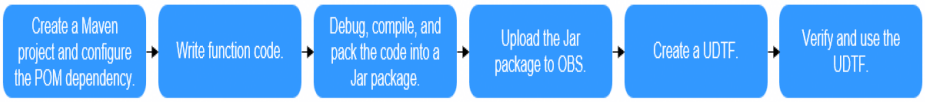


Table 4-6 Process description

No.	Step	Operation Interface	Description
1	Create a Maven project and configure the POM file.	IntelliJ IDEA	Write the UDTF code by referring to the **Procedure**.
2	Write the UDTF code.		

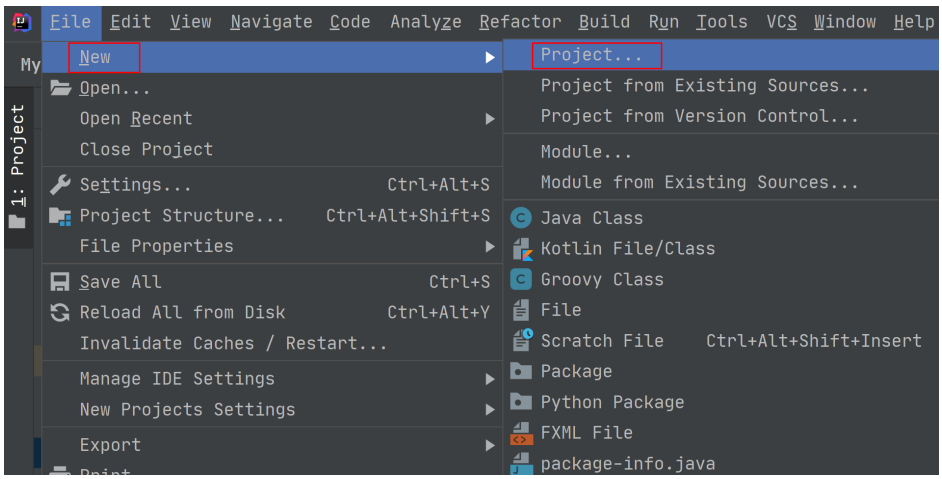
 Copyright © Huawei Technologies Co., Ltd.

No.	Step	Operation Interface	Description
3	Debug, compile, and pack the code into a JAR package.		
4	Upload the JAR package to OBS.	OBS console	Upload the UDTF JAR file to an OBS directory.
5	Create a UDTF.	DataArts Studio console	Create a UDTF on the **Develop Script** page of the DataArts Studio console.
6	Verify and use the UDTF.	DataArts Studio console	On the **Develop Script** page of the DataArts Studio console, use the created UDTF in a Spark SQL job.

Procedure

1. Create a Maven project and configure the POM file. This step uses IntelliJ IDEA 2020.2 as an example.
   a. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

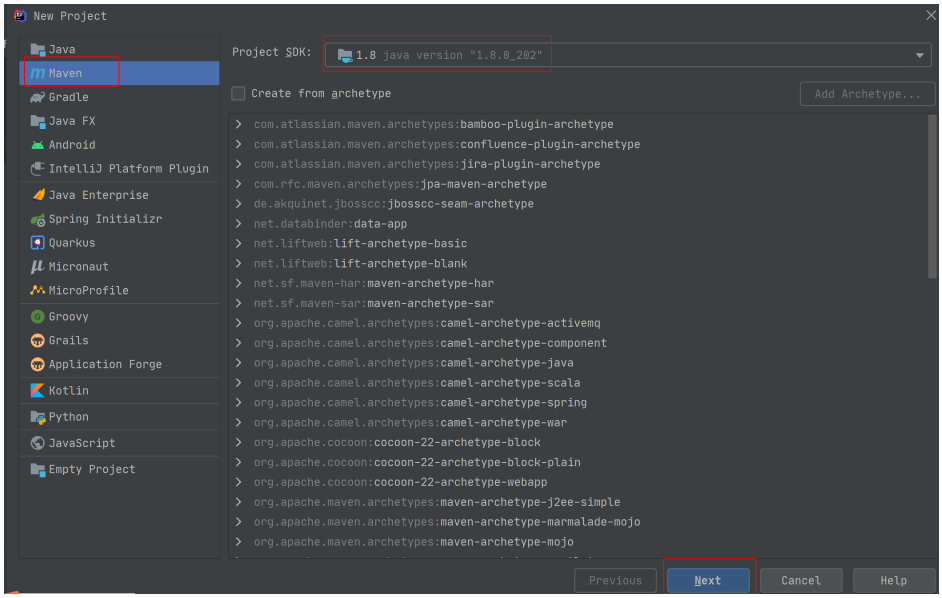
      Figure 4-23 Creating a project

   b. Choose **Maven**, set **Project SDK** to **1.8**, and click **Next**.

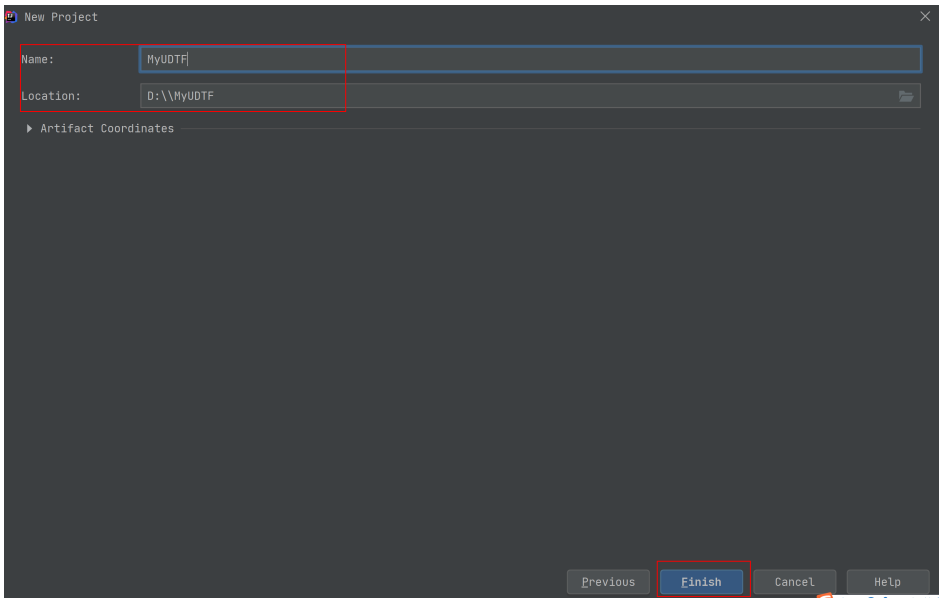
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-24 Choosing Maven



c. Set the project name, configure the storage path, and click **Finish**.

Figure 4-25 Creating a project

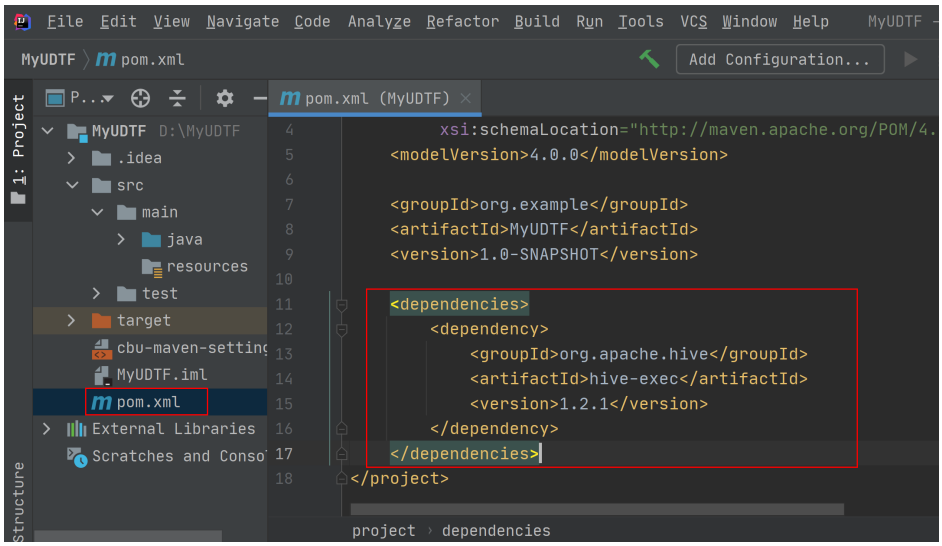


d. Add the following content to the **pom.xml** file.

```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

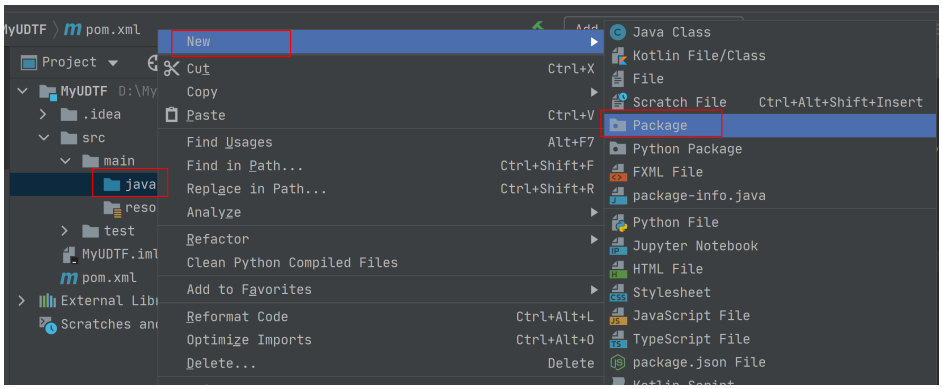
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-26 Adding configurations to the POM file



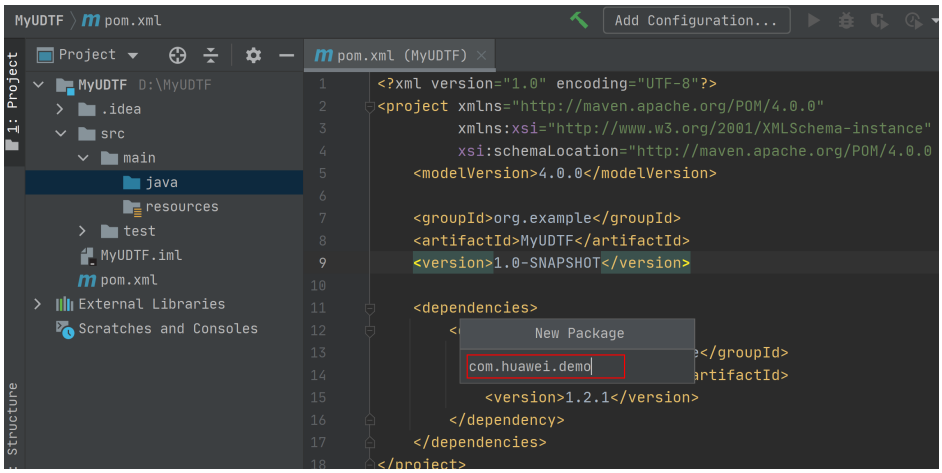
e. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package and a class file.

Figure 4-27 Creating a package and a class file



Set the package name as you need. In this example, set **Package** to **com.huawei.demo**. Then, press **Enter**.

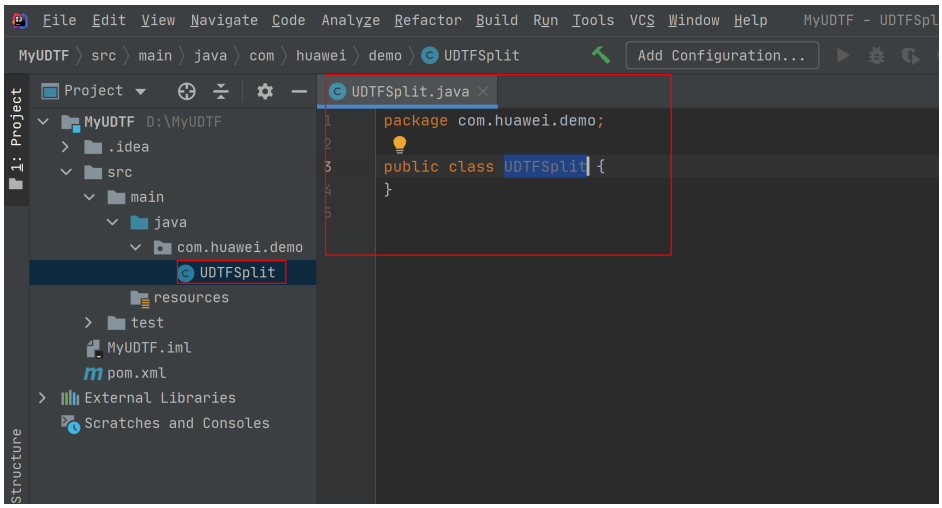
Figure 4-28 Customizing a package



 Copyright © Huawei Technologies Co., Ltd.

Create a Java class file in the package path. In this example, the Java Class file is **UDTFSplit**.

Figure 4-29 Creating a Java class file



2. Write the UDTF code. For sample code, see **Sample Code**.

   The UDTF class must inherit **org.apache.hadoop.hive.ql.udf.generic.GenericUDTF** to implement the **initialize**, **process**, and **close** methods.

   a. Call the **initialize** method in the UDTF. This method returns the information about the returned data rows of the UDTF, such as the number and type.

   b. Call the **process** method to process data. Each time **forward()** is called in the **process** method, a row is generated.

      If multiple columns are generated, you can put the values in an array and pass the array to the **forward()** function.

```
public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }
```

   c. Call the **close** method to clear methods that need to be closed.

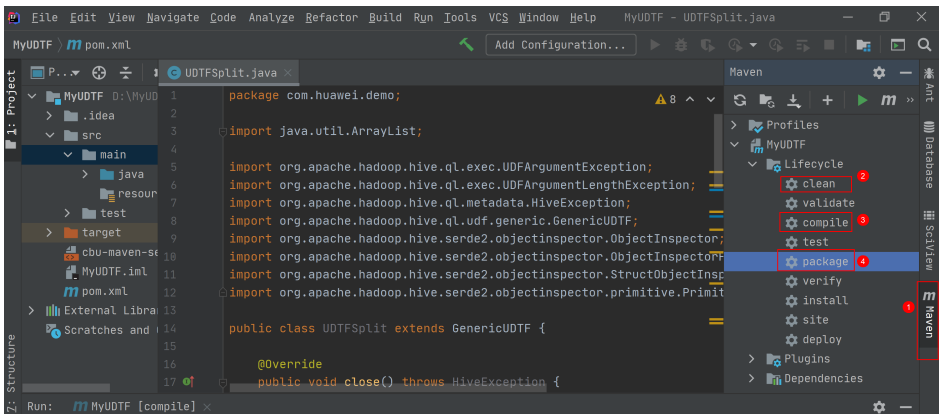
3. Use IntelliJ IDEA to compile the code and pack it into the JAR package.

   a. Click **Maven** in the tool bar on the right, and click **clean** and **compile** to compile the code.

 Copyright © Huawei Technologies Co., Ltd.

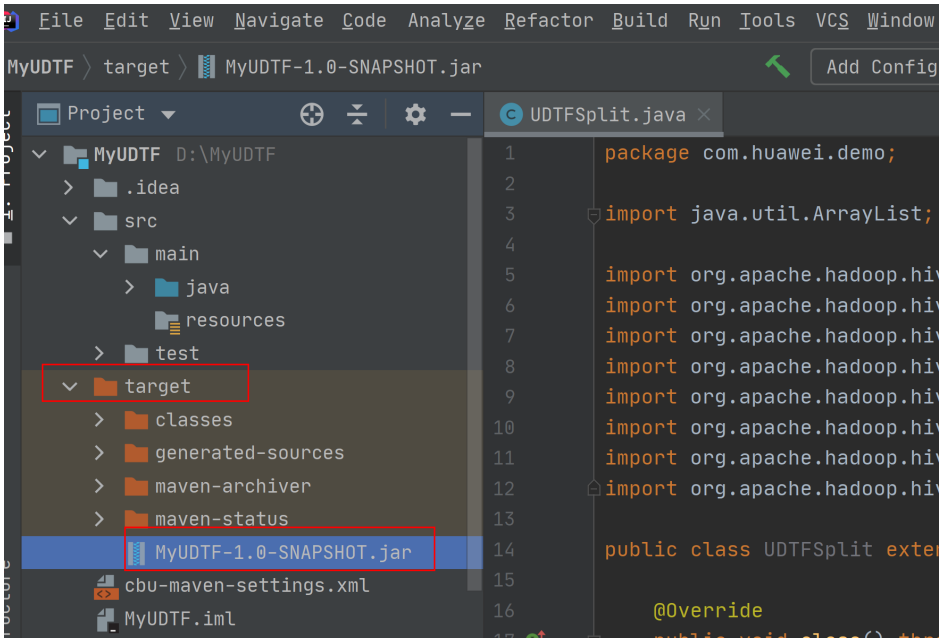
After the compilation is successful, click **package**.

Figure 4-30 Compiling and packaging



The generated JAR package is stored in the **target** directory. In this example, the generated JAR package is named **MyUDTF-1.0-SNAPSHOT.jar** and stored in the D:\MyUDTF\target directory.

Figure 4-31 Generating a JAR file



4. Log in to the OBS console and upload the JAR file to the OBS path.

   NOTE

   The region of the OBS bucket to which the JAR file is uploaded must be the same as the region of the AI DataLake workspace. Cross-region operations are not allowed.

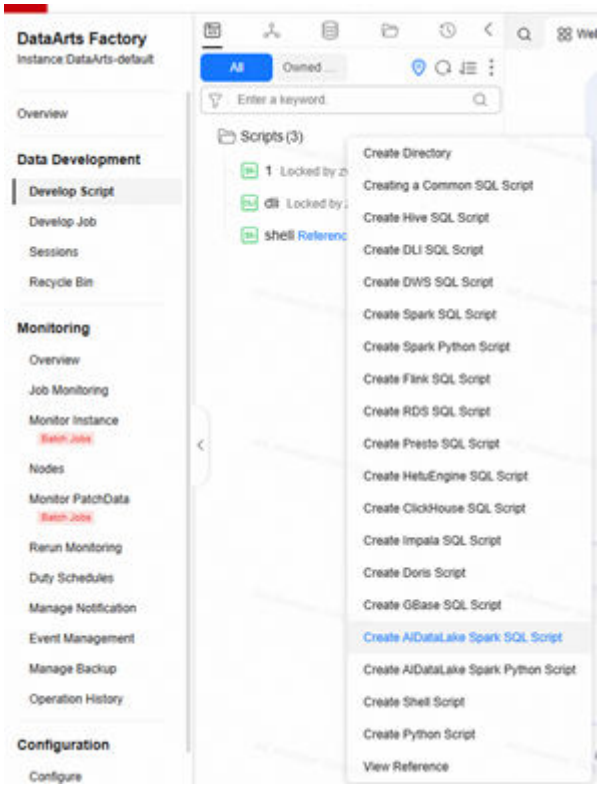
5. Create a UDTF.

   a. On the **AI DataLake console**, click **DataArts Studio** in the lower left corner.

   b. On the **Develop Script** page, create a Serverless Spark SQL script and select the endpoint, catalog, and database.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-32 Creating a Serverless Spark SQL script



c. In the SQL editing area, enter the following command for creating the UDTF:

```
CREATE FUNCTION mytestsplit AS 'com.huawei.demo.UDTFSplit' using jar 'obs://aidatalake-test-obs01/MyUDTF-1.0-SNAPSHOT.jar';
```

d. Click **Execute**.

6. Verify and use the UDTF.

   Use the UDTF in the SELECT statement as follows:

   ```
   select mytestsplit('abc:123\;efd:567\;utf:890');
   ```

7. (Optional) Delete the UDTF.

   If this function is no longer used, run the following statement to delete it:

   ```
   Drop FUNCTION mytestsplit;
   ```

Sample Code

The complete **UDTFSplit.java** code is as follows:

```
import java.util.ArrayList;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class UDTFSplit extends GenericUDTF {
```

 Copyright © Huawei Technologies Co., Ltd.

```
    @Override
    public void close() throws HiveException {
        // TODO Auto-generated method stub

    }

    @Override
    public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {
        if (args.length != 1) {
            throw new UDFArgumentLengthException("ExplodeMap takes only one argument");
        }
        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
            throw new UDFArgumentException("ExplodeMap takes string as a parameter");
        }

        ArrayList<String> fieldNames = new ArrayList<String>();
        ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();
        fieldNames.add("col1");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldNames.add("col2");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

}
```

4.4 Calling Remote UDFs in Spark SQL Jobs

4.4.1 Overview

Function Description

A remote user-defined function (UDF) refers to separating the execution logic of a UDF from the main computing engine. The main engine calls this service via standard interfaces such as HTTP, gRPC, or Thrift to complete the computation. By decoupling the computation logic from the main engine, remote UDF improves flexibility and reusability.

 Copyright © Huawei Technologies Co., Ltd.

This section explains the steps to develop a remote UDF and call it from a computing engine, using FunctionGraph as the execution platform for the remote UDF.

Learn more:

- **Comparison of UDF and Remote UDF**
- **Comparison of UDF, UDAF, and UDTF**

Solution Benefits

The advantages of remote UDFs include risk isolation, resource decoupling, and cost optimization.

- Risk isolation: Modifying UDFs in critical production environments can introduce execution risks.

  Remote UDFs offload the computing logic to FunctionGraph, maintaining service functionality while isolating risks through a serverless architecture.
- Resource decoupling: For resource-intensive tasks like large model inference, remote UDFs avoid long-term resource occupation in Spark clusters. This ensures stability, cost-effectiveness, and functionality by offloading heavy computations to FunctionGraph.
- Cost optimization: Remote UDFs turn the **default** queue into a pure data channel, removing risks caused by UDFs and offering more cost-effective and secure core data flow services to users.

Notes and Constraints

Remote UDFs only affect single-row data and are suitable for single-input, single-output scenarios.

Environment Preparations

Before developing and debugging a remote UDF using FunctionGraph, you must install IDEA, configure Maven, download required JAR files, and set up the FunctionGraph service.

Table 4-7 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required.
IntelliJ IDEA	IntelliJ IDEA 2019.1 or a compatible version is required.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.

 Copyright © Huawei Technologies Co., Ltd.

Preparation Item	Description
Dependency JAR file download	Download the dependency JAR file **spark-function-sdk_2.13-jar-with-dependencies.jar** from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/**.

Development Process

The Remote UDF development process is as follows:

Figure 4-33 Development process

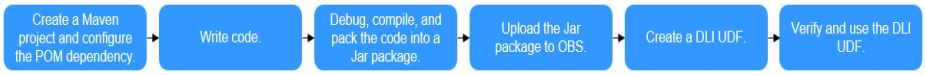


Table 4-8 Process description

No.	Step	Operation Interface	Description
1	Create a Maven project and configure the project structure.	IntelliJ IDEA	Write, debug, and compile the remote UDF code.
2	Write the remote UDF code.		
3	Debug, compile, and pack the code into a JAR package.		
4	Create a FunctionGraph function and upload the JAR file.	OBS console	Upload the generated JAR file to FunctionGraph.
5	Create a remote UDF.	DataArts Studio console	Create a UDF on the **Develop Script** page of the DataArts Studio console.

 Copyright © Huawei Technologies Co., Ltd.

No.	Step	Operation Interface	Description
6	Verify and use the remote UDF.	DataArts Studio console	On the **Develop Job** page of the DataArts Studio console, use the created UDF in a Spark SQL job.

Comparison of UDF and Remote UDF

- UDF: Ideal for high-performance scenarios where the development team is familiar with the query engine's language. It offers high execution efficiency but has lower flexibility and security.
- Remote UDF: Best for scenarios that require external service calls, multi-language development, or high flexibility in the development language. It provides better security and greater development flexibility.

Table 4-9 Comparison of UDF and remote UDF

Feature	UDF	Remote UDF
Execution method	Runs locally on the query engine.	Executes through a function hosting service.
Security	Instability in UDF code may affect the query engine process.	Execution failures only impact the UDF itself, not the overall job process.
Performance	High execution efficiency	Comparable to local UDFs in vectorized and batch modes.
Development flexibility	Highly coupled with query engine code, with limited flexibility.	Highly flexible and applicable to external services or libraries.
Development cost	Requires familiarity with the query engine, with high development cost.	Lower development cost due to reduced complexity.
Use case	High-performance scenarios where the team knows the query engine languages.	Scenarios needing flexible development languages or external service calls.

Comparison of UDF, UDAF, and UDTF

- UDF: A function that computes the sum of two numbers.
- UDAF: A function that computes an aggregate value, such as an average.

 Copyright © Huawei Technologies Co., Ltd.

- UDTF: A function that splits a string containing multiple values into multiple rows.

Table 4-10 Comparison of UDF, UDAF, and UDTF

Feature	User-Defined Function (UDF)	User-Defined Aggregate Function (UDAF)	User-Defined Table-Generating Function (UDTF)
Input parameters	One or more fields of a single row	Multiple fields from multiple rows	One or more fields of a single or multiple rows
Output result	Single scalar value	Single aggregate result	Multiple rows, each with multiple fields
Main use	Simple data transformation and computation (e.g., string manipulation, math)	Data aggregation (e.g., sum, average, count)	Data expansion and complex structure parsing (e.g., splitting strings, expanding arrays)

- **UDF example**:

  Creates a function that adds 1 to the input number.

  ```
  CREATE FUNCTION add_one AS 'com.example.AddOneUDF';
  SELECT add_one(age) FROM users;
  ```

  If the **users** table has a column **age** with a value of **25**, the function will output **26**.

- **UDAF example**:

  Calculates the maximum value of a column.

  ```
  CREATE FUNCTION max AS 'com.example.MaxUDAF';
  SELECT max(age) FROM users;
  ```

  If the **users** table has a column **age** with values **25**, **30**, and **35**, the function outputs **35**.

- **UDTF example**:

  Parses a JSON string into multiple rows.

  ```
  CREATE FUNCTION parse_json AS 'com.example.ParseJsonUDTF';
  SELECT parse_json(json_column) FROM users;
  ```

  If the **users** table has a column **json_column** with values **'{"name":"Alice","age":25},{"name":"Bob","age":30}'**, the function will output two rows, each containing **name** and **age** fields.

 Copyright © Huawei Technologies Co., Ltd.

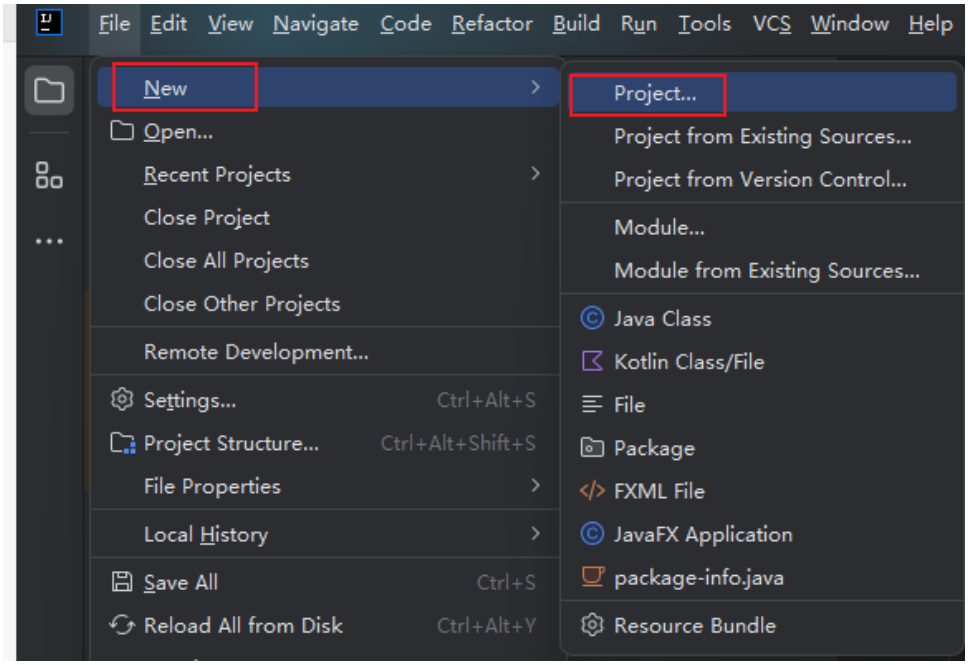
4.4.2 Creating a Maven Project and Configuring the Project Structure

Creating a Maven Project and Configuring the Project Structure

The following uses IntelliJ IDEA 2024.3.2.2 as an example.

1. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

   Figure 4-34 Creating a project

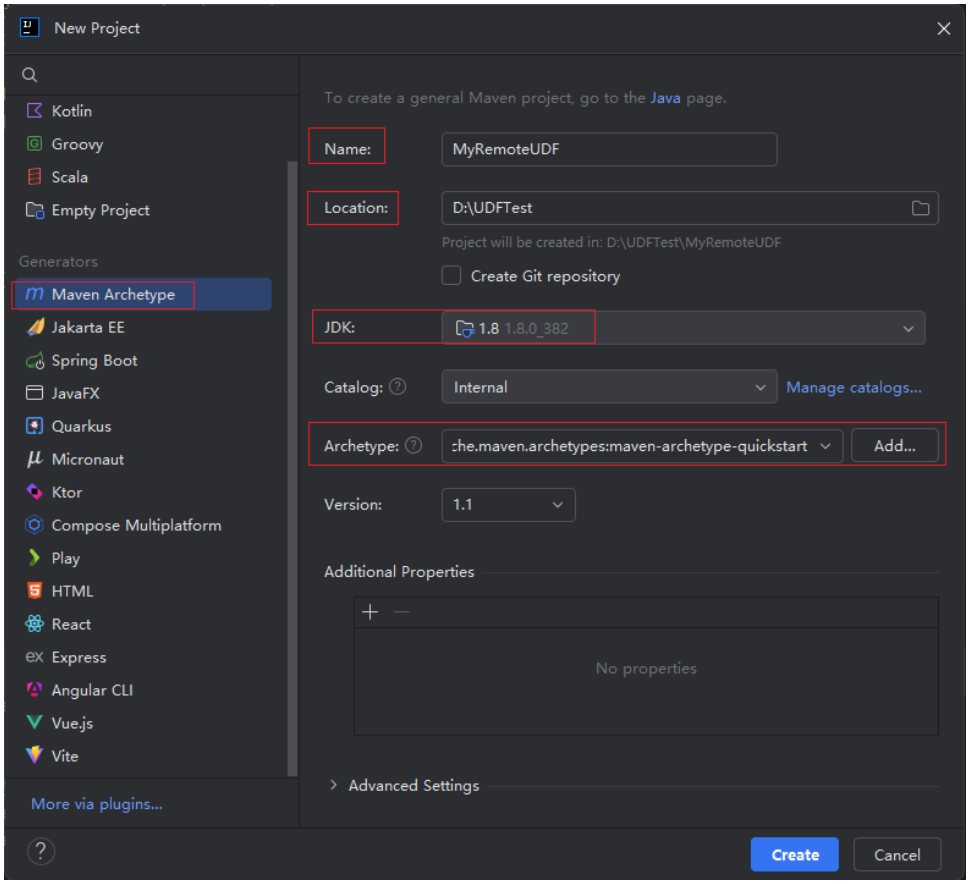
   

2. Choose **Maven Archetype** and configure the following parameters:
   - **Name**: Enter a project name. In this example, the project name is **MyRemoteUDF**.
   - **Location**: Select a path for storing the project. In this example, the project is saved in **D:\UDFTest**.
   - **JDK**: Select the JDK 1.8.
   - **Archetype**: Select **org.apache.maven.archetypes:maven-archetype-quickstart**.

   Click **Create** to create the project.

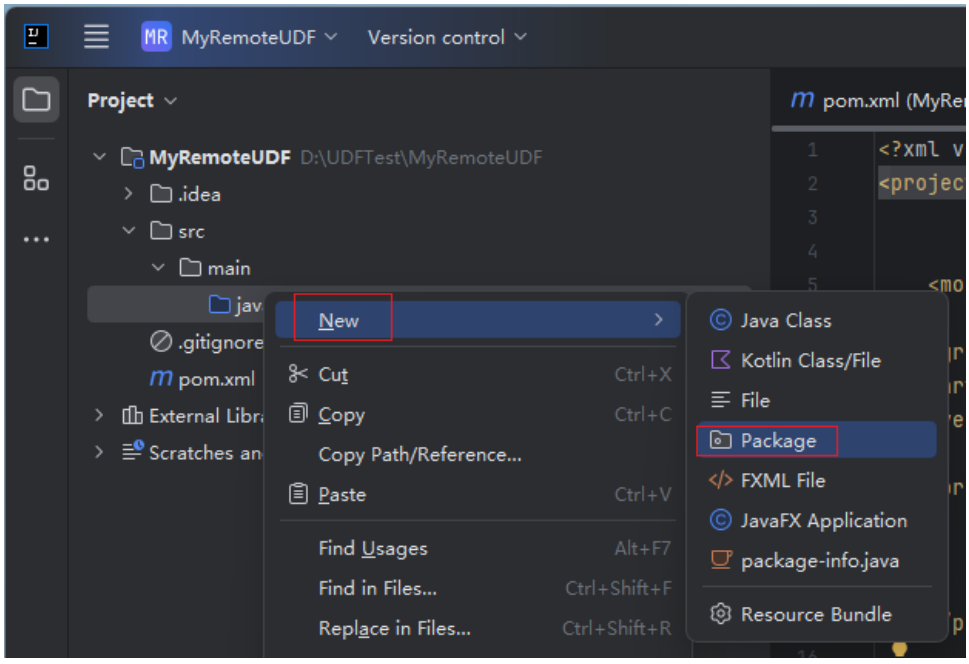
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-35 Creating a Maven project



3. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package.

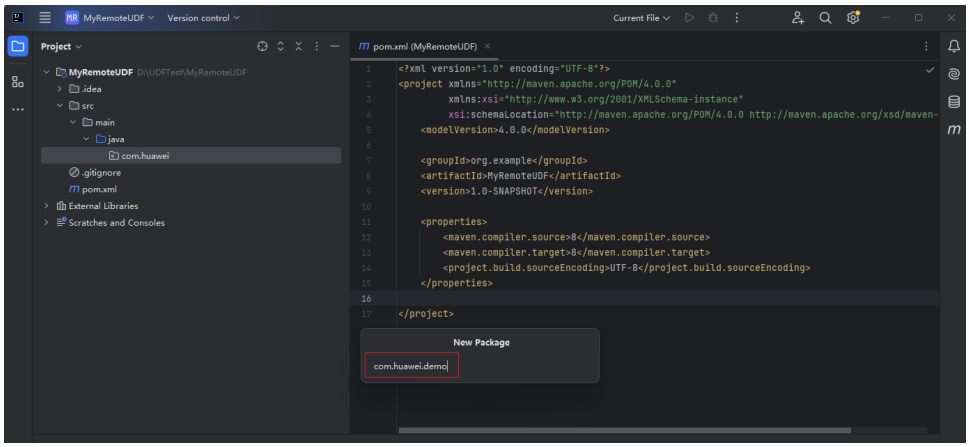
Figure 4-36 Creating a package



 Copyright © Huawei Technologies Co., Ltd.

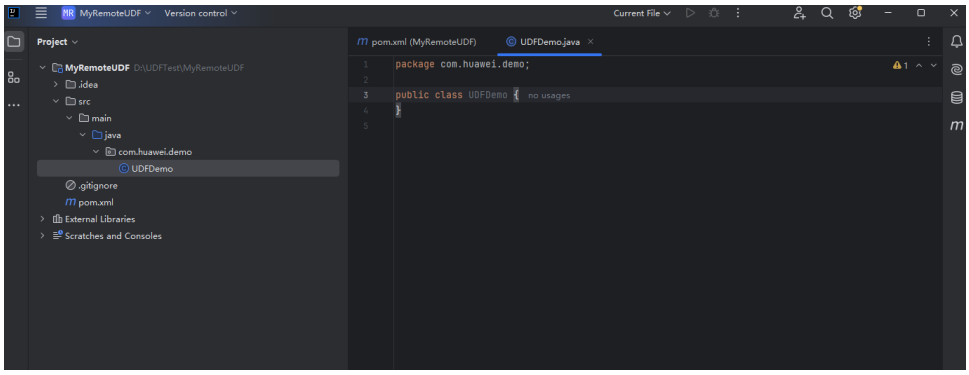
Set the package name as you need. In this example, the package name is **com.huawei.demo**. Then, press **Enter**.

Figure 4-37 Setting the package name



4. Create a Java Class file in the package path. In this example, the Java Class file is **UDFDemo**.

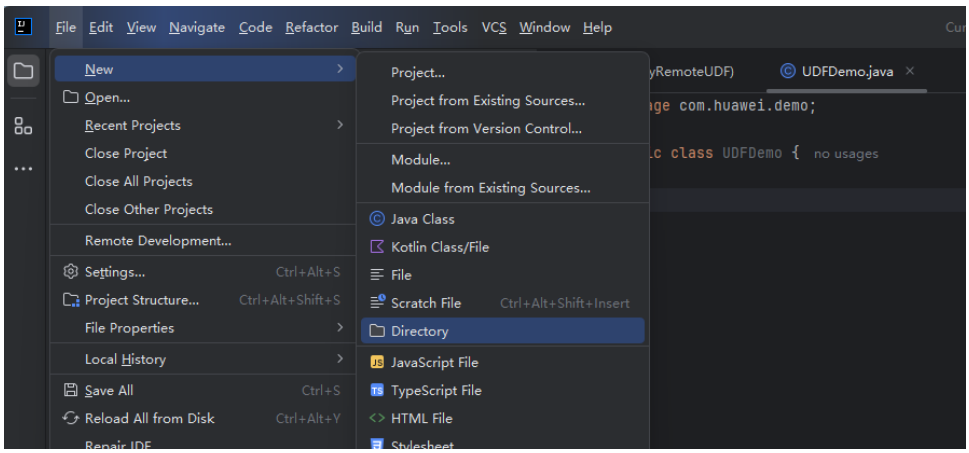
Figure 4-38 Creating a Java class file



5. Add the path for storing the JAR package.

   Click **File** > **New** > **Directory** to create a folder for storing the JAR package.

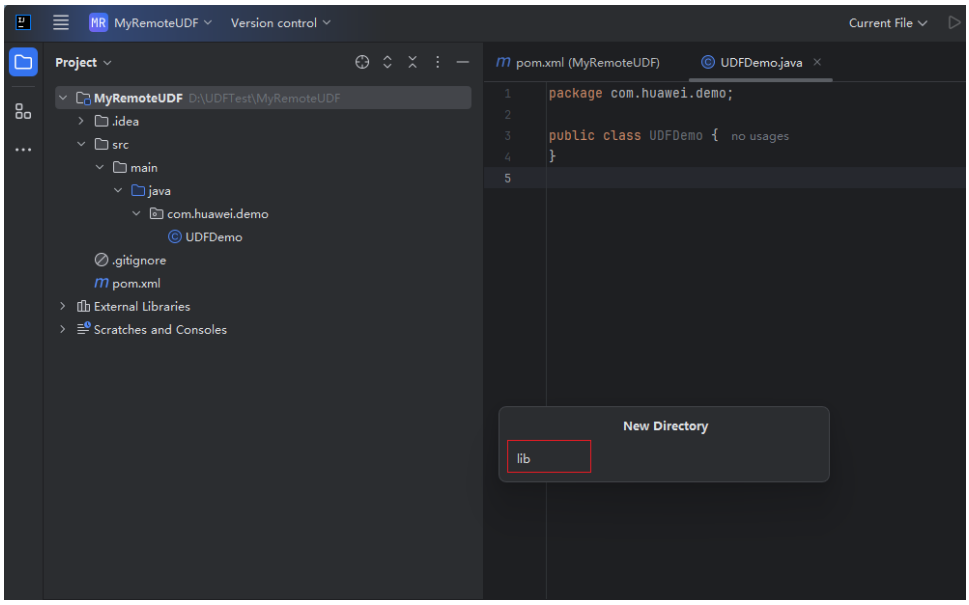
Figure 4-39 Adding the path for storing the JAR package



 Copyright © Huawei Technologies Co., Ltd.

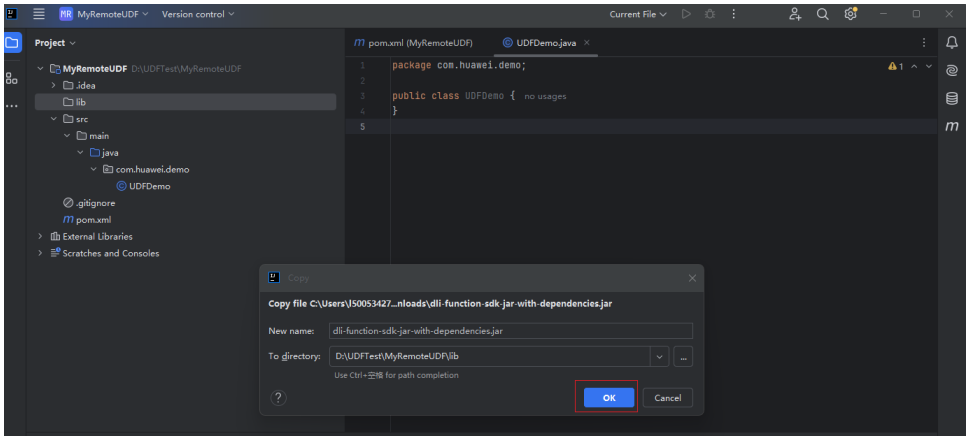
In this example, the folder name is **lib**. Press **Enter** to confirm the name.

Figure 4-40 Setting the folder name



6. Import the JAR package.
   a. Download the JAR package
      from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs**.
   b. After the download is complete, right-click the JAR package and choose **Copy** from the shortcut menu.
7. Locate the **lib** directory of the project, paste it to the created **lib** folder, and click **OK** in the dialog box that is displayed.

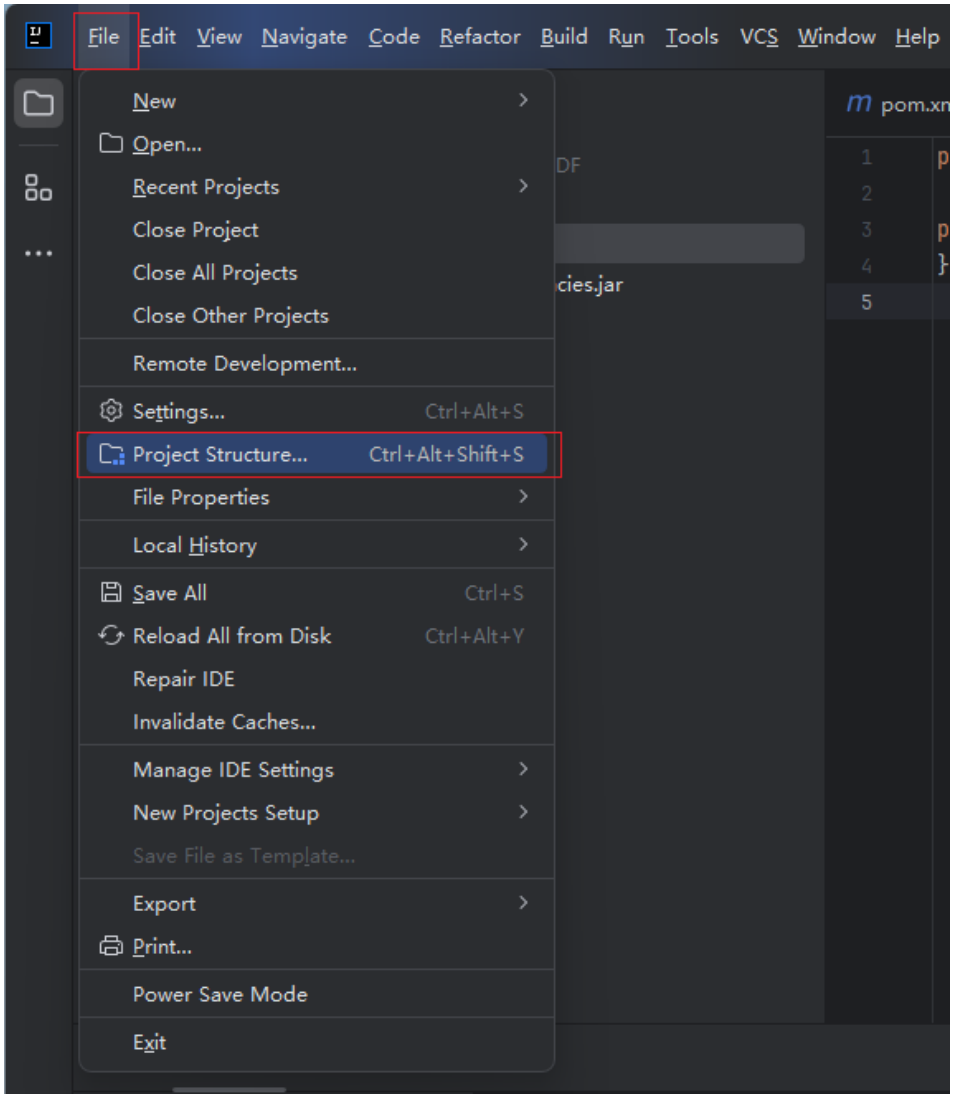
Figure 4-41 Pasting the JAR package to the lib folder



8. Configure **Project Structure**.
   a. Click **File** > **Project Structure**.

 Copyright © Huawei Technologies Co., Ltd.

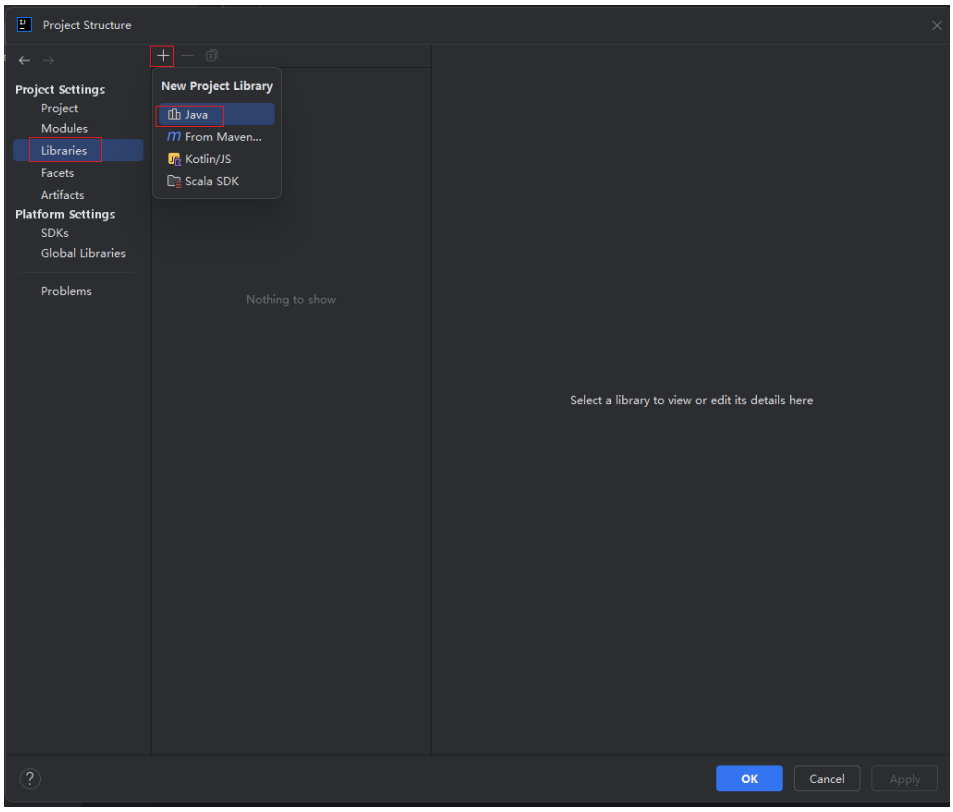
Figure 4-42 Clicking Project Structure



b. On the navigation pane, choose **Libraries**. Then click **+** and select **Java**.

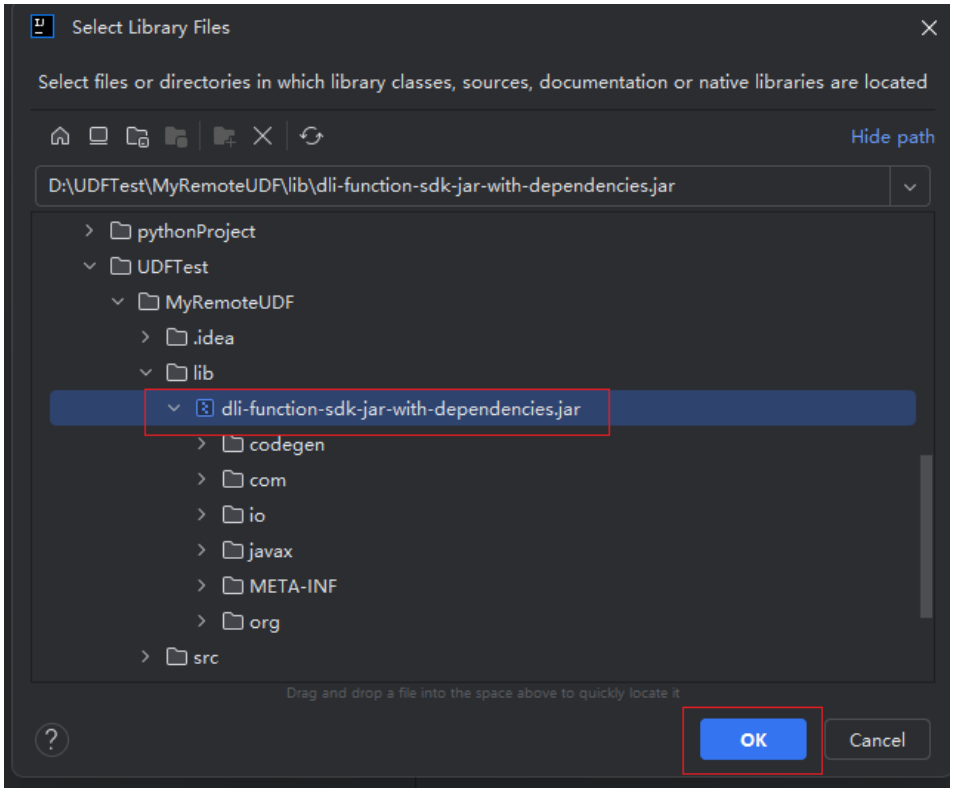
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-43 Clicking Java



c. Select the JAR package added to the **lib** folder and click **OK**.

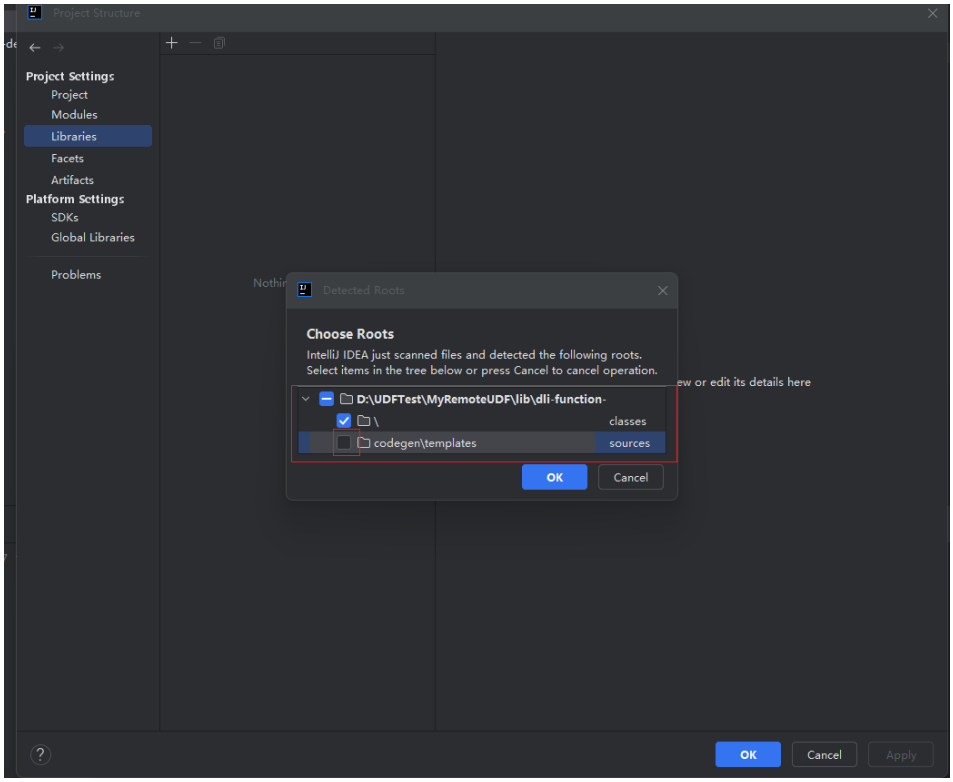
Figure 4-44 Selecting the JAR package added to the lib folder



 Copyright © Huawei Technologies Co., Ltd.

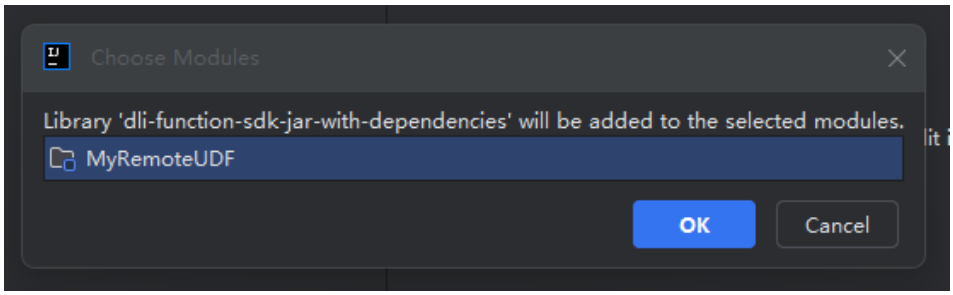
d. On the displayed **Choose Roots** dialog box, select only the first folder and click **OK**.

Figure 4-45 Choose Roots dialog box



e. Click **OK** as prompted.

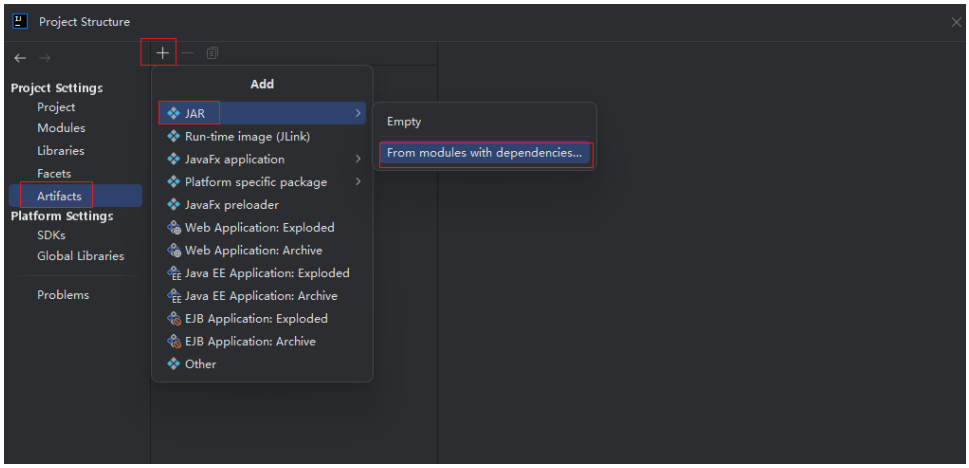
Figure 4-46 Clicking OK



9. Click **Artifacts**. On the displayed page, click **+** and then choose **JAR** > **From modules with dependencies**.

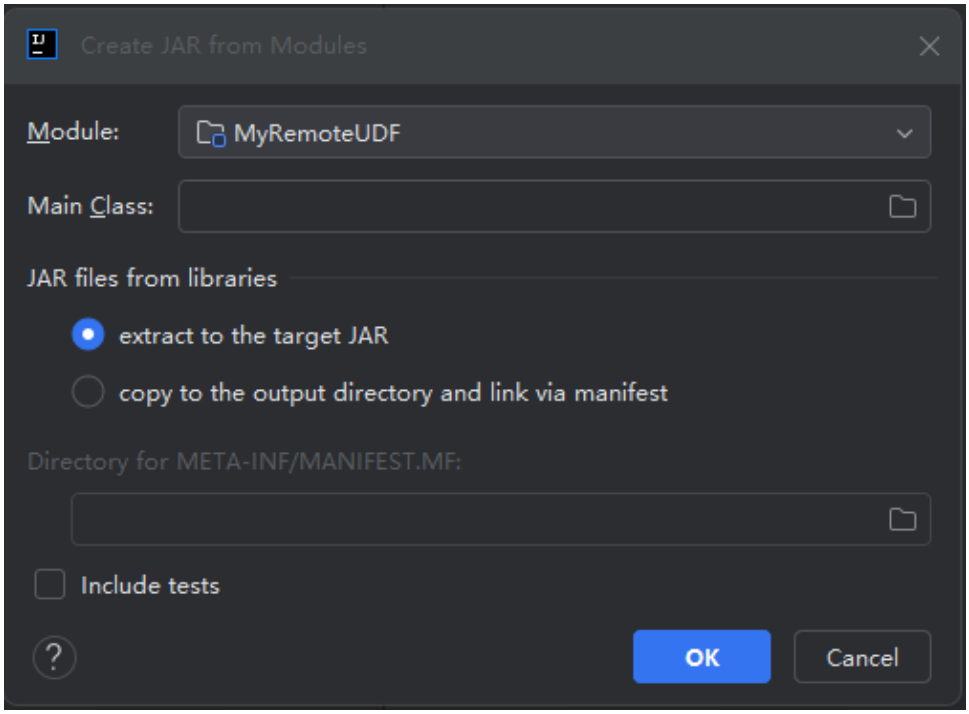
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-47 Choosing From modules with dependencies



10. In the **JAR files from libraries** area, enable the first option and click **OK**.

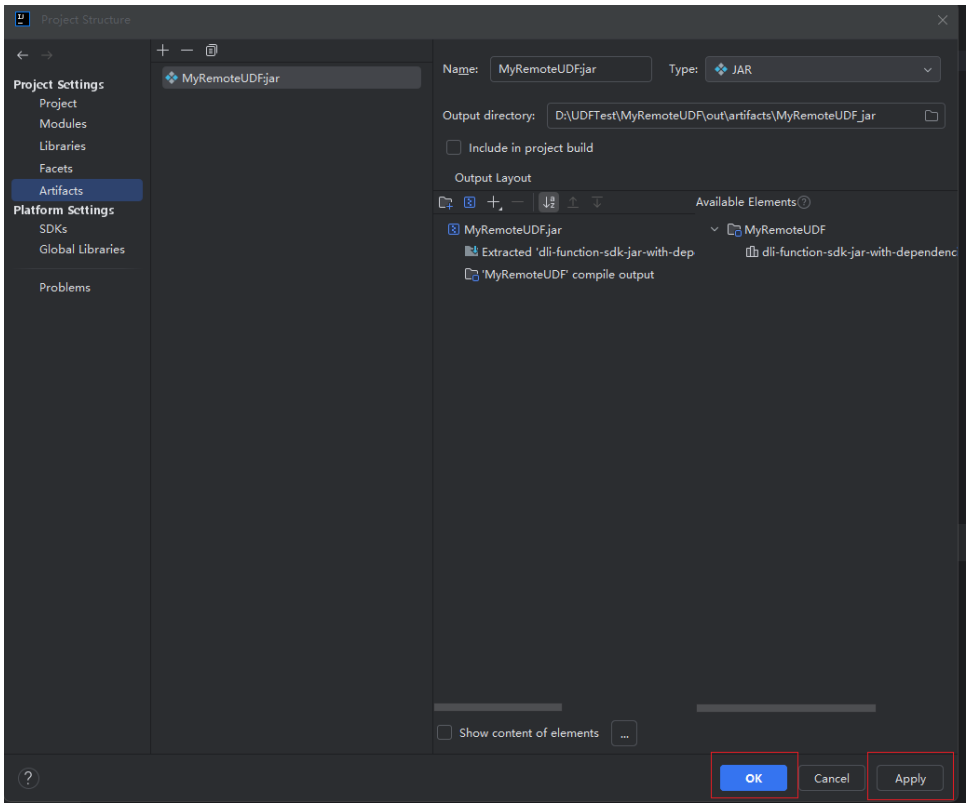
Figure 4-48 Clicking OK



11. Click **Apply** and then **OK**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-49 Configuring Artifacts



4.4.3 Writing the UDF Code and Exporting the JAR File

Step 1: Writing the UDF Code

Pay attention to the following requirements when you implement the UDF:

- The UDF must inherit UDFHandler.
- The @TypeResolve annotation is added to the method to indicate the input and output parameters of the function.

  Input and output parameters are separated by an arrow (->). For details about the input and output parameter types (**resolveType**) and the corresponding SQL and Java data types, see **Table 4-11**.

Table 4-11 resolveType

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG

 Copyright © Huawei Technologies Co., Ltd.

SQL type	Java type	resolveType
FloatType	Float.class	FLOAT
DoubleType	Double.class	DOUBLE
StringType	String.class	STRING
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	DECIMAL
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthInterval-Type	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- Add the @deterministic annotation.

  The @deterministic annotation indicates that the function is deterministic, that is, it always returns the same result for the same input.

  For details about how to implement the UDF, see the following sample code:

  – Description: Implements the addition of two integers and returns the result.

  – Example code:

```
package com.huawei.demo;

import com.huawei.spark.function.types.FunctionType;
import com.huawei.spark.function.types.TypeResolve;
import com.huawei.spark.function.handlers.UDFHandler;

@FunctionType(deterministic = true)
public class UDFDemo extends UDFHandler {
    @FunctionType(deterministic = false)
    @TypeResolve("INT, INT->INT")
    public Integer testIntegerTypeAdd(Integer col1, Integer col2) {
        return col1 + col2;
    }
}
```

  – Parameters:

    col1: The first integer parameter (of integer type)

    col2: The second integer parameter (of integer type)

 Copyright © Huawei Technologies Co., Ltd.

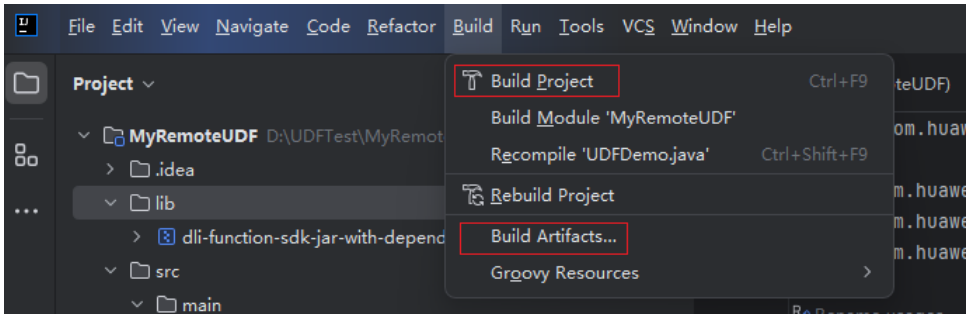
- Return value: The arithmetic sum of **col1** and **col2**

Step 2: Exporting the JAR Package

Use IntelliJ IDEA to compile the code and pack it into the JAR package.

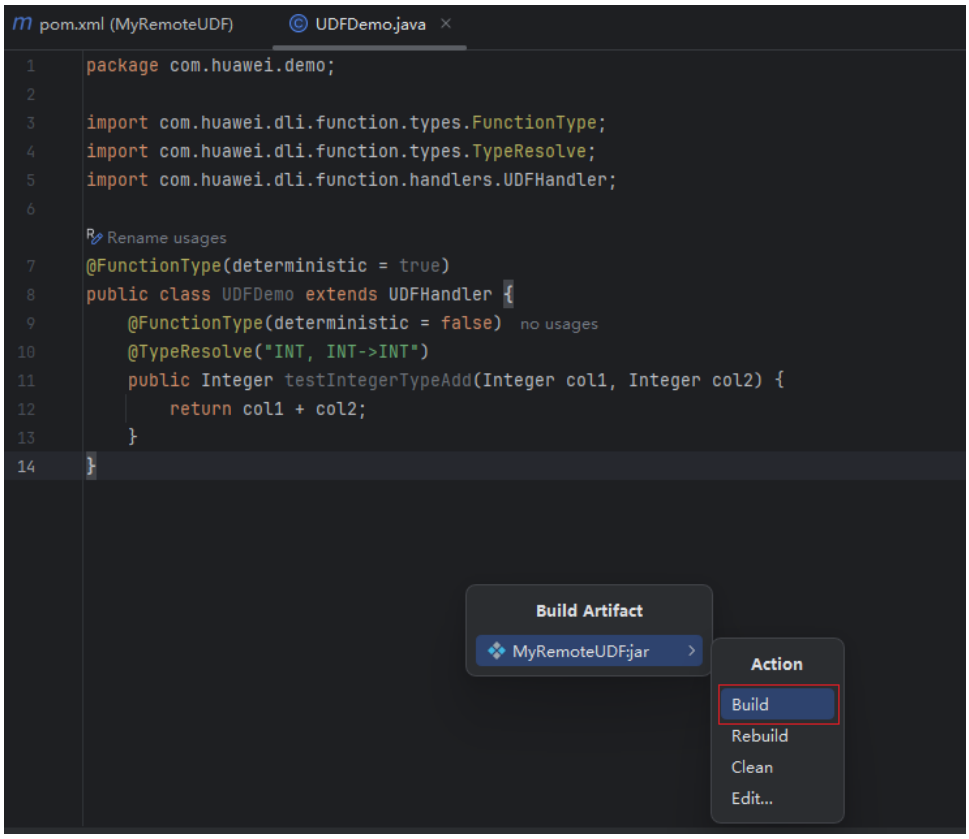
1. Click **Build**. On the displayed page, click **Build Project** and **Build Artifacts...** to compile and pack the code, respectively.

   Figure 4-50 Compiling and packing



2. In the displayed dialog box, click **Build**.

   Figure 4-51 Compiling and packing

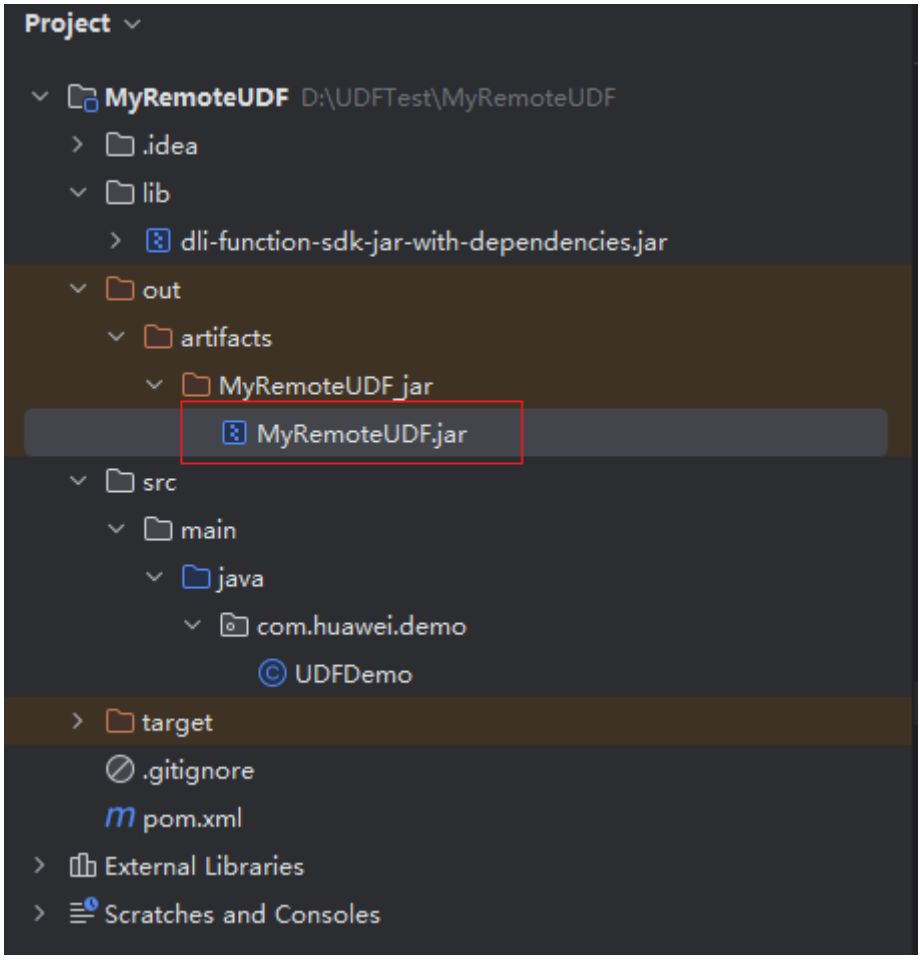


3. After the preceding steps are complete, locate the **out** folder in the project directory and expand it.

   The **MyRemoteUDF.jar** file is the JAR file to be uploaded to FunctionGraph.

 Copyright © Huawei Technologies Co., Ltd.

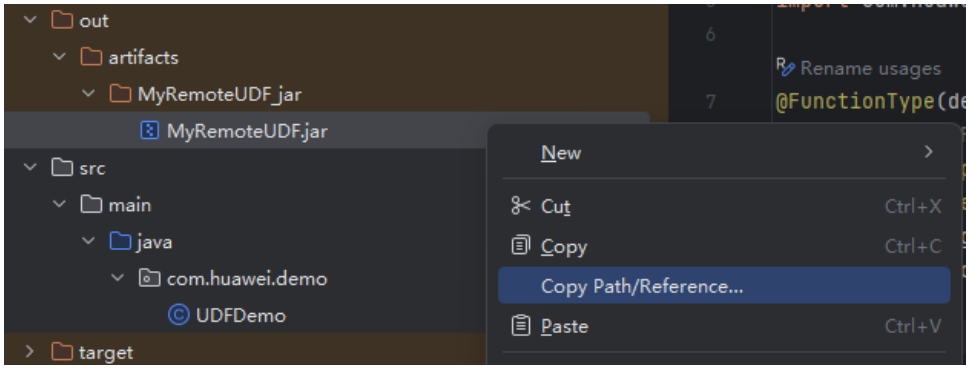
Figure 4-52 Viewing MyRemoteUDF.jar



4. Right-click **MyRemoteUDF.jar** and choose **Copy Path/Reference** from the shortcut menu to copy its path.

   In this example, the **MyRemoteUDF.jar** file is generated in the **D:\UDFTest\MyRemoteUDF\out\artifacts\MyRemoteUDF_jar** directory.

Figure 4-53 Copying the file path



4.4.4 Creating a Function with FunctionGraph

Prerequisites:

Learn about the **process of using FunctionGraph**.

 Copyright © Huawei Technologies Co., Ltd.

Making Preparations

Before creating a function with FunctionGraph, obtain the two configuration items.

Table 4-12 Configuration items

Item	Configuration Parameter
functiongraph Endpoint	spark.hadoop.fs.functiongraph.endpoint=functiongraph.XXYYZZ.huawei.com
Project ID	spark.hadoop.fs.user.projectId=xxxxxxxxxc9782fxxxxxxxxxxxxxxx

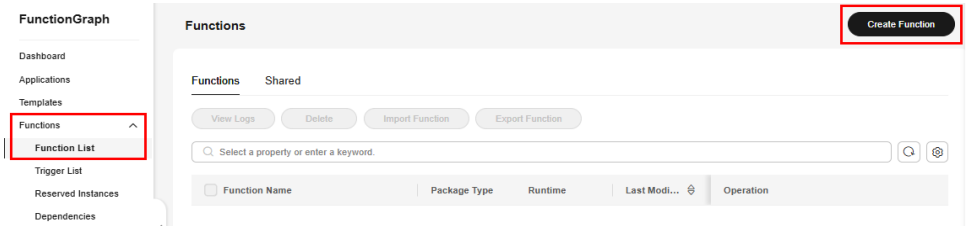
NOTE

When submitting a Spark SQL or Spark job, you need to set the preceding two configuration items.

Step 1: Creating a Function

1. Log in to the FunctionGraph console.
2. In the navigation pane on the left, choose **Functions** > **Function List**. On the displayed page, click **Create Function**.

   Figure 4-54 Creating a function

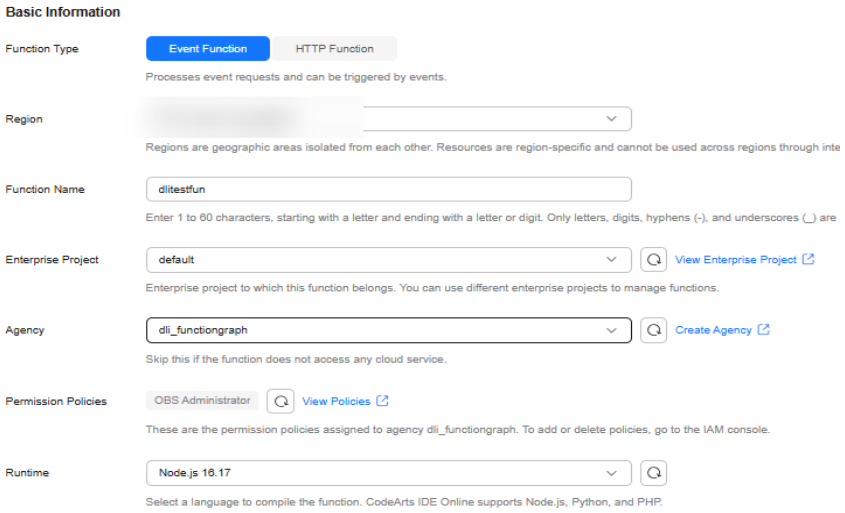
3. Click **Create from scratch** and configure required parameters.

   For details about the parameters, see **Creating an Event Function**.

   - **Function Type**: Select **Event Function**.
   - **Runtime**: Select Java 8.
   - **Agency**: Select an agency to authorize FunctionGraph to access other cloud services, and configure agency permissions for functions.
   - There are **constraints and restrictions** on the amount of data that can be processed by a FunctionGraph call at a time. If the function size exceeds 4 MB, you are advised to configure an agency to access OBS. For details, see **Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-55 Creating a function



Step 2: Configuring the Function

1. **Set the function handler.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the UDF handler in the **Handler** area.

      The UDF handler is in the format of **[*Package name*].[*Class name*].[*Function name*]**.

      - [*Package name*].[*Class name*] indicates the class where the UDF is located.
      - The function name is fixed to **handleRequest**.

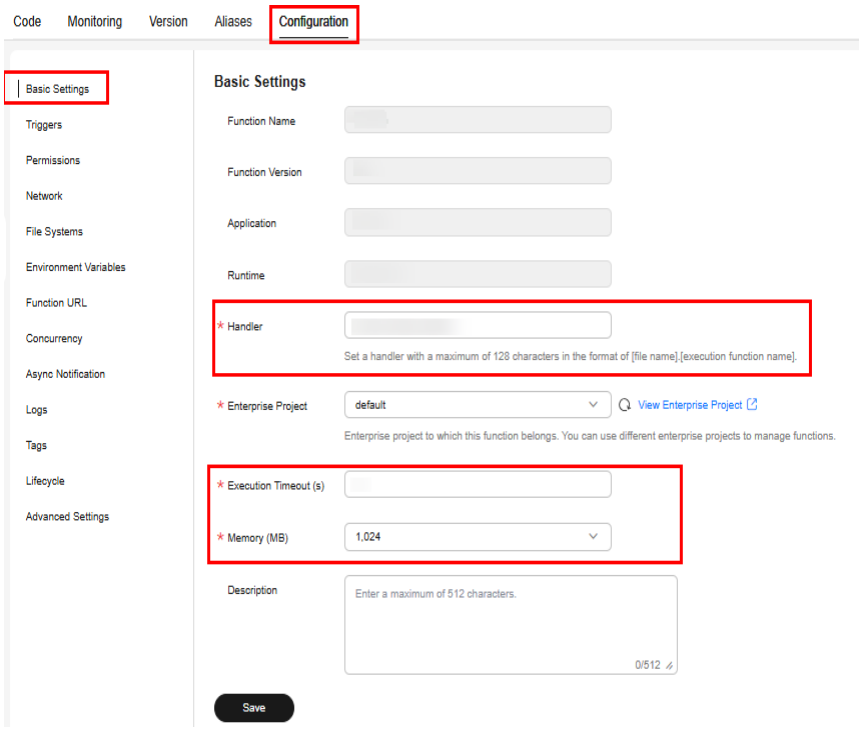
      In this example, it is set to **com.huawei.demo.UDFDemo.handleRequest**.
   c. After the setting is complete, click **Save**.
2. **Set the execution timeout interval and memory.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the execution time and memory.

      Configure the **execution timeout interval and memory size** based on the complexity of the UDF. The memory size determines the **billing**.

      In this example, the execution time is set to 15 seconds and the memory is set to 1024 MB.
   c. After the setting is complete, click **Save**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-56 Basic settings



3. **Configure the class isolation.**

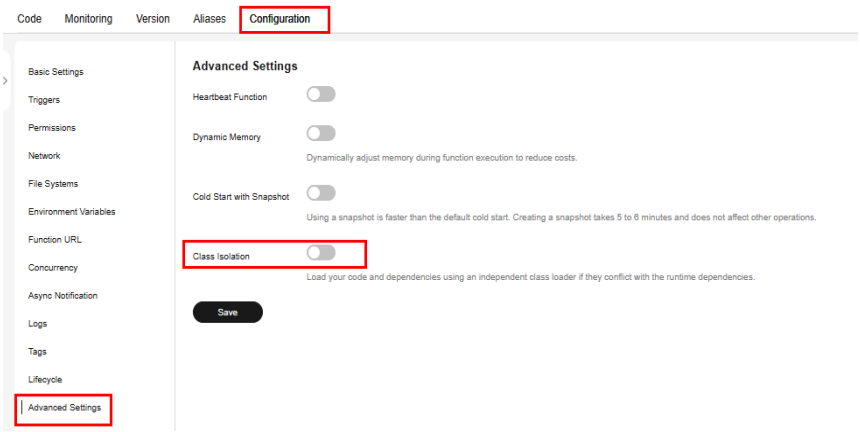
   a. After the function is created, click the function name in the list to go to its details page.

   b. Choose **Configuration** > **Advanced Settings** and configure the **Class Isolation** option.

      After class isolation is enabled, an independent class loader is used to load the uploaded code and its dependencies, resolving dependency and runtime conflicts. You can disable class isolation if there is no dependency conflict.

In this example, the class isolation is disabled. After the setting is complete, click **Save**.

Figure 4-57 Configuring class isolation

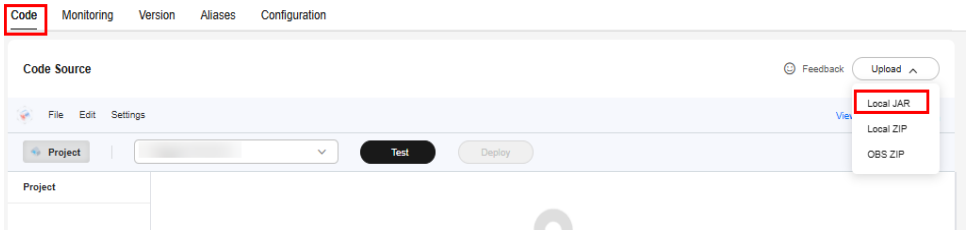


 Copyright © Huawei Technologies Co., Ltd.

Step 3: Uploading the JAR Package

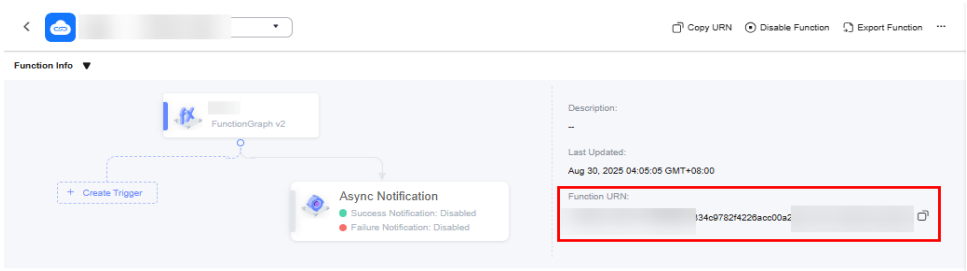
1. On the FunctionGraph console, click the target function name in the function list to go to its details page.
2. Click the **Code** tab, select **Local JAR** from the **Upload** drop-down list on the right, and upload the exported JAR package.

Figure 4-58 Uploading the JAR file



3. After the upload is successful, you can copy the Uniform Resource Name (URN) on the function details page. The URN uniquely identifies a function and will be used when the function is called.

Figure 4-59 Copying a URN

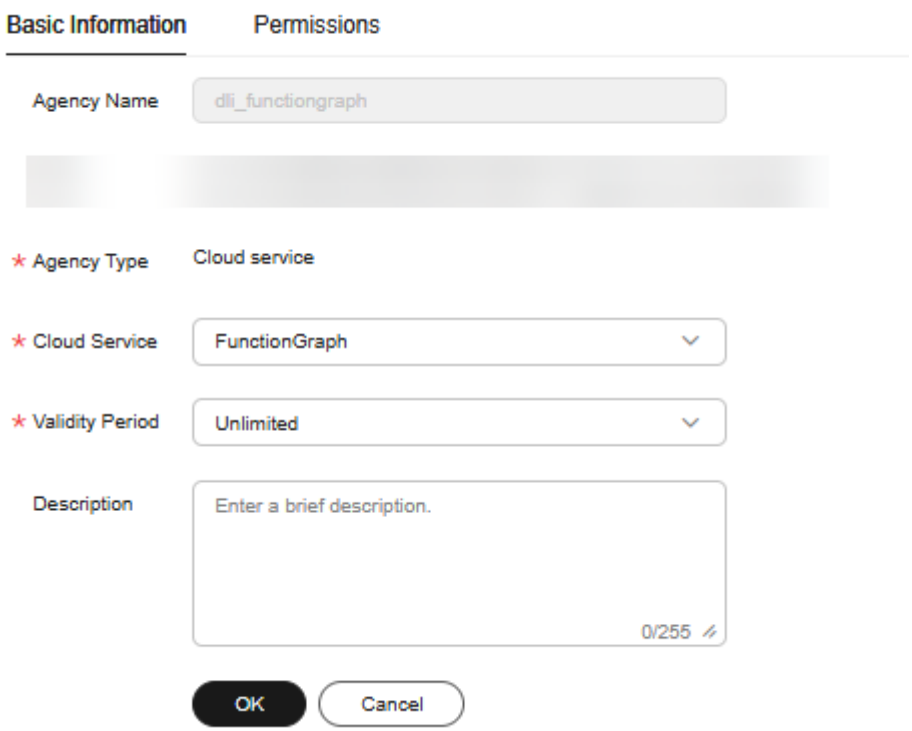


Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services

1. Log in to the management console.
2. In the upper right corner of the page, hover over the username and select **Identity and Access Management**.
3. In the navigation pane of the IAM console, choose **Agencies**.
4. On the displayed page, click **Create Agency**.
5. Set the following parameters:
   - **Agency Name**: Enter an agency name.
   - **Agency Type**: Select **Cloud service**.
   - **Cloud Service**: Select **FunctionGraph** from the drop-down list. This parameter is available only when you select **Cloud service** for **Agency Type**.
   - **Validity Period**: Select **Unlimited**.
   - **Description**: You can enter **Agency with OBS OperateAccess permissions**. This parameter is optional.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-60 Creating an agency



6. After configuring the basic information, click **OK**.
7. On the displayed page, search for the OBS Administrator permission in the search box on the right and select it. Then, click **Next**.
8. Click **Next**. On the **Select Scope** page, set the authorization scope.

   For details about authorization operations, see **Creating a User Group and Assigning Permissions**.

   - **All resources**: IAM users will be able to use all resources, including those in enterprise projects, region-specific projects, and global services under your account based on assigned permissions.
   - **Global services**: Global services are deployed in all regions, so you can access them seamlessly without having to switch between regions. However, global services cannot be authorized based on regional projects. They include services like Object Storage Service (OBS) and Content Delivery Network (CDN). After authorization, you can use the selected services based on your permissions.
   - **Region-specific projects**: IAM users will be able to use resources in the selected region-specific projects based on assigned permissions.
   - **Enterprise projects**: IAM users will be able to use resources in the selected enterprise projects based on assigned permissions. For example, an enterprise project may contain resources that are deployed in different regions. After you associate the enterprise project with the IAM users, they can access the resources in this enterprise project based on the assigned permissions.

   In this example, an OBS policy is selected. Therefore, select **Global services**.
9. Click **OK**.

 Copyright © Huawei Technologies Co., Ltd.

10. It takes 15 to 30 minutes for the authorization to be in effect.

    After the authorization takes effect, return to the function details page, select the corresponding agency. The authorized policy is displayed.

4.4.5 Using Remote UDFs

4.4.5.1 Creating a Function Using SQL

Description

Creates a remote UDF for use in Spark job development.

Syntax

```
CREATE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

or

```
CREATE OR REPLACE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

Requirements

None

Keywords

urn: On the FunctionGraph function list page, locate the function whose URN you want to copy and click **Copy URN** in the **Operation** column.

Example

Run the following command to create the UDFDemo function:

```
CREATE FUNCTION UDFDemo AS 'ExampleUDFHandler' USING FUNCTIONGRAPH 'urn:fss:cn-north-7:330e068af1334c9782f4226acc00a2e2:function:default:test:latest'
```

Select the corresponding database and queue, enter the SQL statement, and click **Execute**.

4.4.5.2 Calling a Function Using SQL

Description

Calls a specified function.

Syntax

```
SELECT function_name(column_name) FROM table_name;
```

Requirements

None

 Copyright © Huawei Technologies Co., Ltd.

Keywords

function_name: The name of the function created on FunctionGraph.

Example

Call the sample function **remote_udf** to return the average value of **real_stock_rate**.

```
SELECT remote_udf(real_stock_rate)  FROM dw_ad_estimate_real_stock_rate LIMIT 1000;
```

4.4.5.3 Deleting a Function Using SQL

Description

Deletes a specified function.

Syntax

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

Requirements

None

Keywords

- **TEMPORARY**: Indicates whether the function to be deleted is a temporary function.
- **IF EXISTS**: Prevents an error message from being displayed if the function to be deleted does not exist.

Example

```
DROP FUNCTION remote_udf;
```

4.4.5.4 Displaying All Functions

Description

Views all functions in the current project.

Syntax

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

Requirements

None

Keywords

LIKE: This qualifier is used only for compatibility and has no practical effect.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
SHOW FUNCTIONS;
```

4.4.5.5 Displaying Function Details

Description

Displays information about a specified function.

Syntax

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

Requirements

None

Keywords

EXTENDED: Displays extended usage information.

Example

```
DESCRIBE FUNCTION remote_udf;
```

4.5 Calling Remote UDAFs in Spark SQL Jobs

4.5.1 Overview

Function Description

Remote User-Defined Aggregate Functions (Remote UDAFs) apply to multiple-in-one-out service scenarios. That is, remote UDAFs aggregate multiple input records into one output value.

This section explains the steps to develop a remote UDAF and call it from a computing engine, using FunctionGraph as the execution platform for the remote UDAF.

Solution Benefits

The advantages of remote UDAFs include risk isolation, resource decoupling, and cost optimization.

- Risk isolation: Although technically feasible, executing UDAFs in critical production environments can introduce execution risks.

  Remote UDAFs offload the computing logic to FunctionGraph, maintaining service functionality while isolating risks through a serverless architecture.
- Resource decoupling: For resource-intensive tasks like large model inference, remote UDFs avoid long-term resource occupation in Spark clusters. This ensures stability, cost-effectiveness, and functionality by offloading heavy computations to FunctionGraph.

 Copyright © Huawei Technologies Co., Ltd.

- Cost optimization: Remote UDAFs turn the **default** queue into a pure data channel, removing risks caused by UDAFs and offering more cost-effective and secure core data flow services to users.

Environment Preparations

Before developing and debugging a remote UDAF using FunctionGraph, you must install IDEA, configure Maven, download required JAR files, and set up the FunctionGraph service.

Table 4-13 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required.
IntelliJ IDEA	IntelliJ IDEA 2019.1 or a compatible version is required.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.
Dependency JAR file download	Download the dependency JAR file **spark-function-sdk_2.13-jar-with-dependencies.jar** from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/**.

Comparison of UDAF and Remote UDAF

- UDAF: Ideal for high-performance scenarios where the development team is familiar with the query engine's language. It offers high execution efficiency but has lower flexibility and security.
- Remote UDAF: Best for scenarios that require external service calls, multi-language development, or high flexibility in the development language. It provides better security and greater development flexibility.

Table 4-14 Comparison of UDAF and remote UDAF

Feature	UDAF	Remote UDAF
Execution method	Runs locally on the query engine.	Executes through a function hosting service.
Security	Instability in UDAF code may affect the query engine process.	Execution failures only impact the UDAF itself, not the overall job process.
Performance	High execution efficiency	Comparable to local UDAFs in vectorized and batch modes.

 Copyright © Huawei Technologies Co., Ltd.

Feature	UDAF	Remote UDAF
Development flexibility	Highly coupled with query engine code, with limited flexibility.	Highly flexible and applicable to external services or libraries.
Development cost	Requires familiarity with the query engine, with high development cost.	Lower development cost due to reduced complexity.
Use case	High-performance scenarios where the team knows the query engine languages.	Scenarios needing flexible development languages or external service calls.

Comparison of UDF, UDAF, and UDTF

- UDF: A function that computes the sum of two numbers.
- UDAF: A function that computes an aggregate value, such as an average.
- UDTF: A function that splits a string containing multiple values into multiple rows.

Table 4-15 Comparison of UDF, UDAF, and UDTF

Feature	User-Defined Function (UDF)	User-Defined Aggregate Function (UDAF)	User-Defined Table-Generating Function (UDTF)
Input parameters	One or more fields of a single row	Multiple fields from multiple rows	One or more fields of a single or multiple rows
Output result	Single scalar value	Single aggregate result	Multiple rows, each with multiple fields
Main use	Simple data transformation and computation (e.g., string manipulation, math)	Data aggregation (e.g., sum, average, count)	Expanding data or parsing complex structures (e.g., splitting strings, flattening arrays)

- **UDF example**:

  Creates a function that adds 1 to the input number.

  ```
  CREATE FUNCTION add_one AS 'com.example.AddOneUDF';
  SELECT add_one(age) FROM users;
  ```

  If the **age** column in the **users** table is **25**, the output will be **26**.
- **UDAF example**:

  Calculates the maximum value of a column.

 Copyright © Huawei Technologies Co., Ltd.

```
CREATE FUNCTION max AS 'com.example.MaxUDAF';
SELECT max(age) FROM users;
```

If the **age** values in the **users** table are **25**, **30**, and **35**, the output will be **35**.

- **UDTF example**:

  Parses a JSON string into multiple rows.

  ```
  CREATE FUNCTION parse_json AS 'com.example.ParseJsonUDTF';
  SELECT parse_json(json_column) FROM users;
  ```

  If the **json_column** column in the **users** table contains **'{"name":"Alice","age":25},{"name":"Bob","age":30}'**, the output will be two rows, each containing **name** and **age** fields.

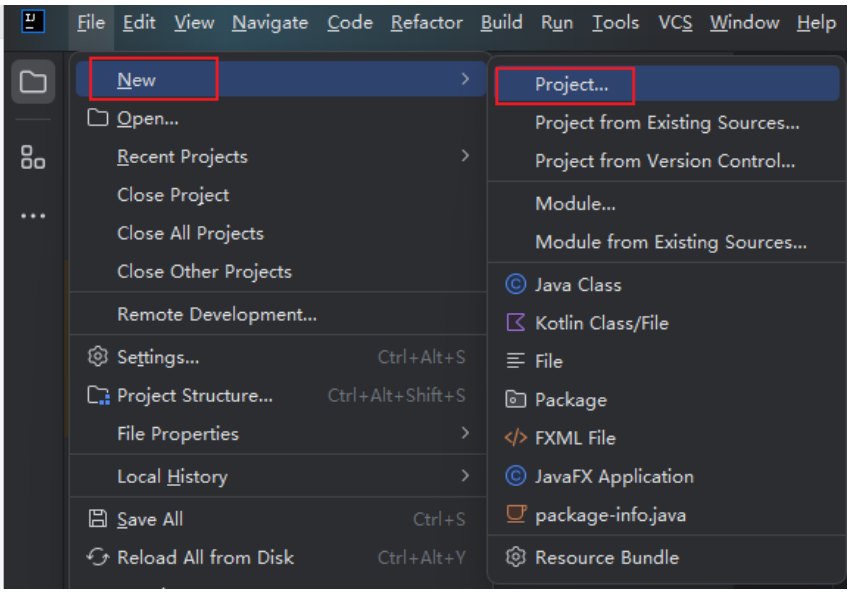
4.5.2 Creating a Maven Project and Configuring the Project Structure

Creating a Maven Project and Configuring the Project Structure

The following uses IntelliJ IDEA 2024.3.2.2 as an example.

1. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

   Figure 4-61 Creating a project

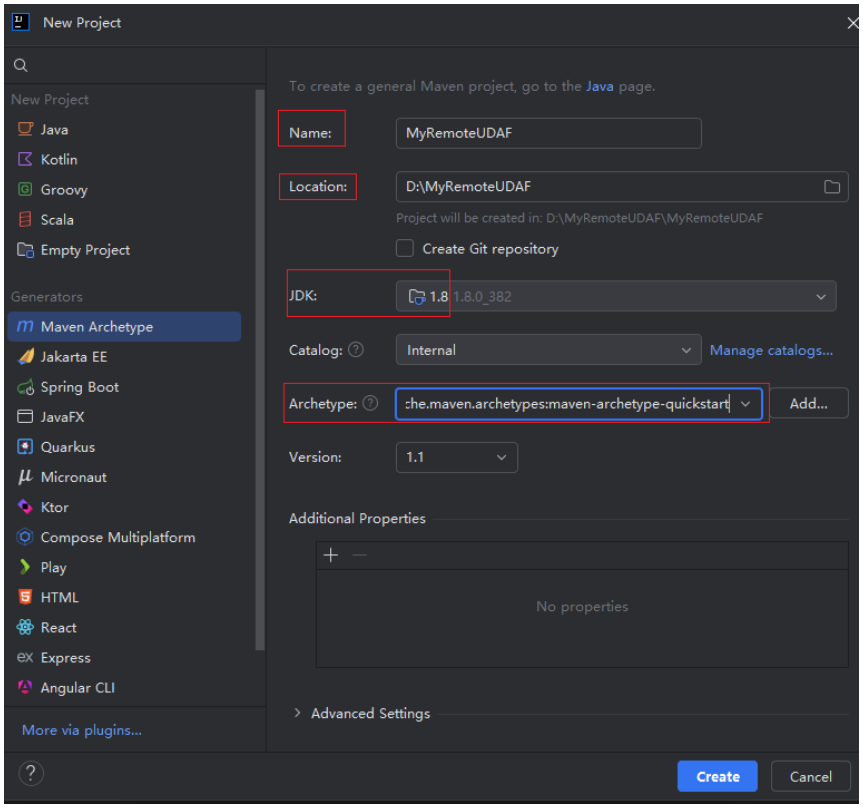
   

2. Choose **Maven Archetype** and configure the following parameters:
   - **Name**: Enter a project name. In this example, the project name is **MyRemoteUDAF**.
   - **Location**: Select a path for storing the project. In this example, the project is saved in **D:\UDAFTest**.
   - **JDK**: Select the JDK 1.8.
   - **Archetype**: Select **org.apache.maven.archetypes:maven-archetype-quickstart**.

   Click **Create** to create the project.

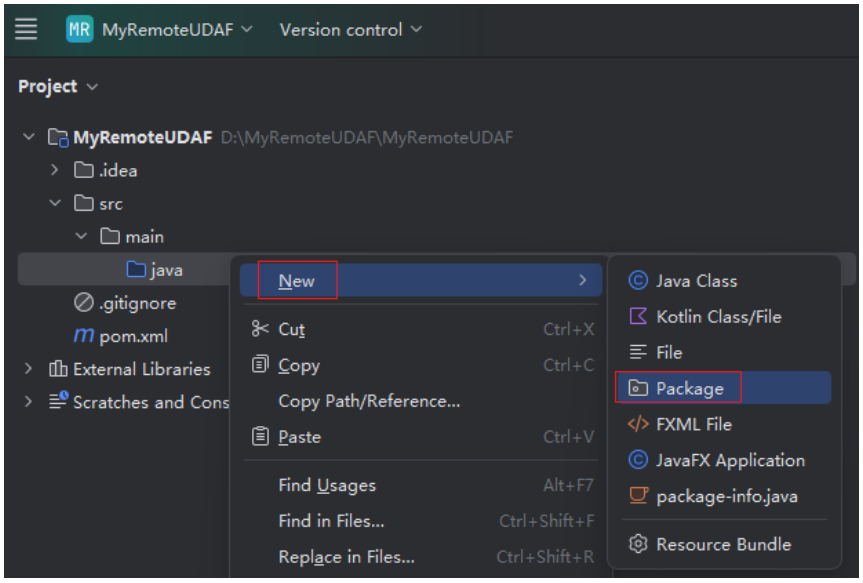
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-62 Creating a Maven project



3. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package.

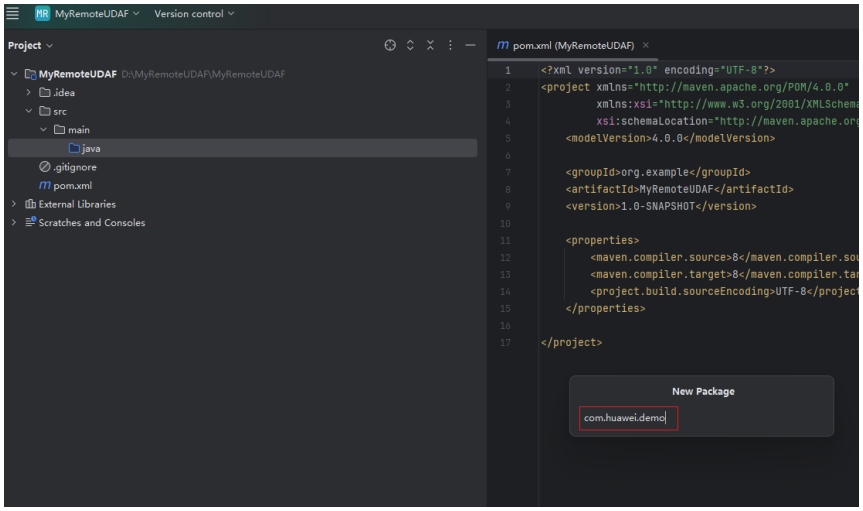
Figure 4-63 Creating a package



Set the package name as you need. In this example, the package name is **com.huawei.demo**. Then, press **Enter**.

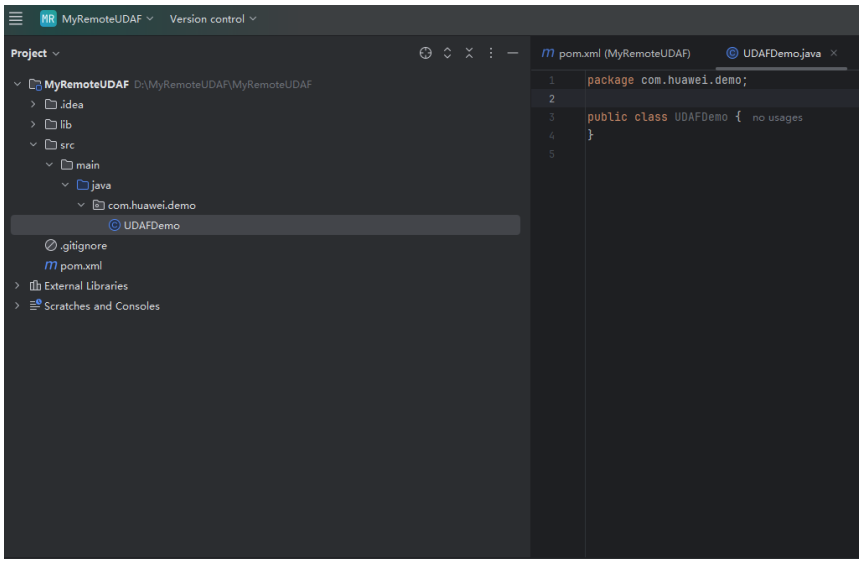
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-64 Setting the package name



4. Create a Java class file in the package path. In this example, the file name is **UDAFDemo**.

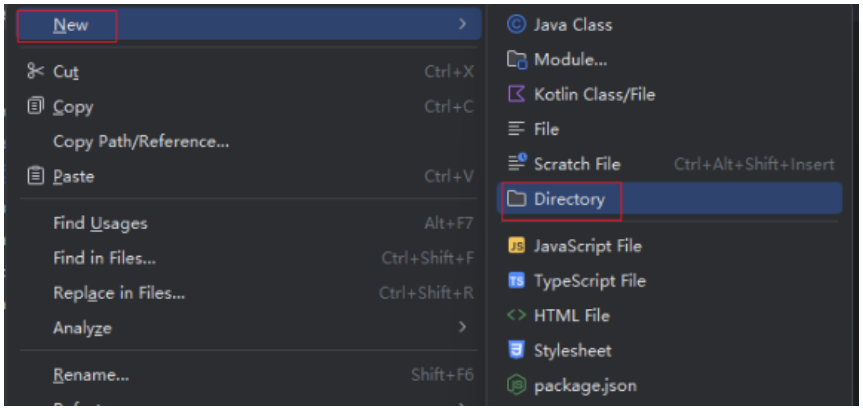
Figure 4-65 Creating a Java class file



5. Add the path for storing the JAR package.
   Click **File** > **New** > **Directory** to create a folder for storing the JAR package.

 Copyright © Huawei Technologies Co., Ltd.

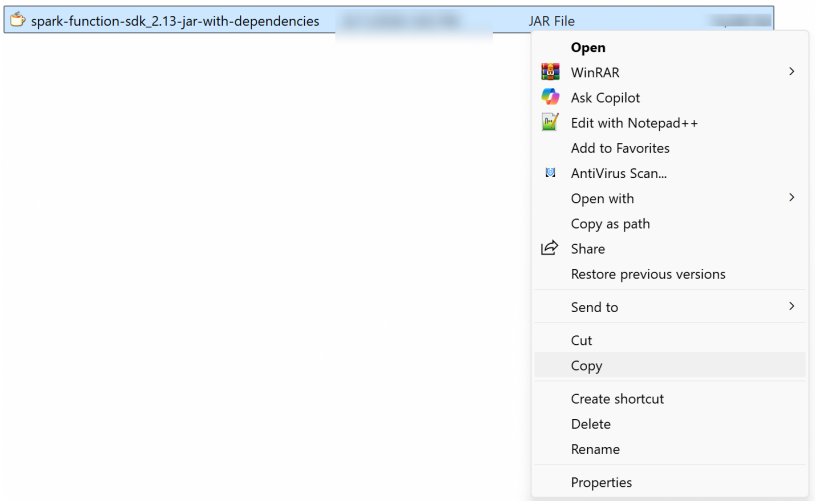
Figure 4-66 Adding the path for storing the JAR package



In this example, the folder name is **lib**. Press **Enter** to confirm the name.

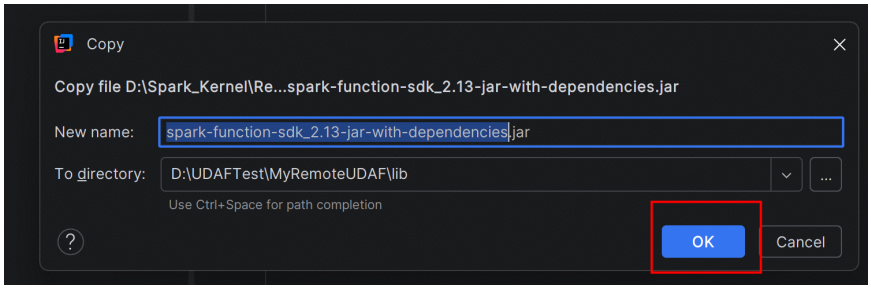
6. Import the JAR package.
   a. Download the JAR package
      from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs**.
   b. After the download is complete, right-click the JAR package and choose **Copy** from the shortcut menu.

      Figure 4-67 Copying the JAR package

7. Locate the **lib** directory of the project, paste it to the created **lib** folder, and click **OK** in the dialog box that is displayed.

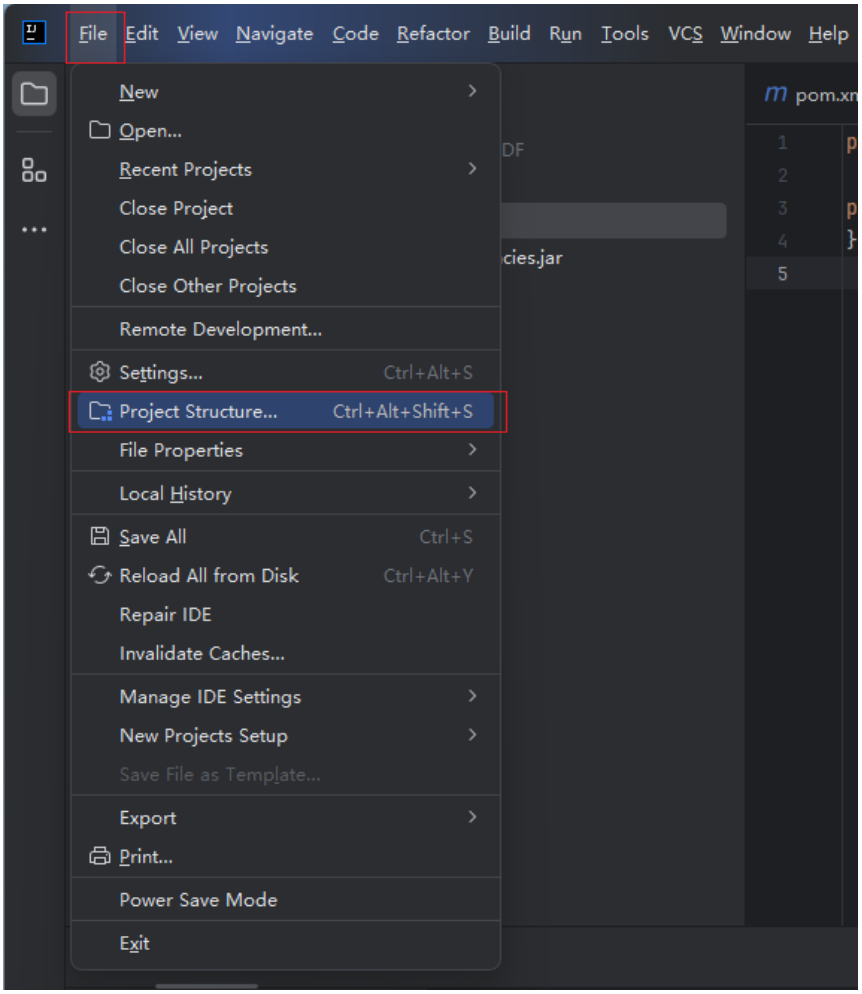
   Figure 4-68 Pasting the JAR package to the lib folder

 Copyright © Huawei Technologies Co., Ltd.

8. Configure **Project Structure**.
   a. Click **File** > **Project Structure**.

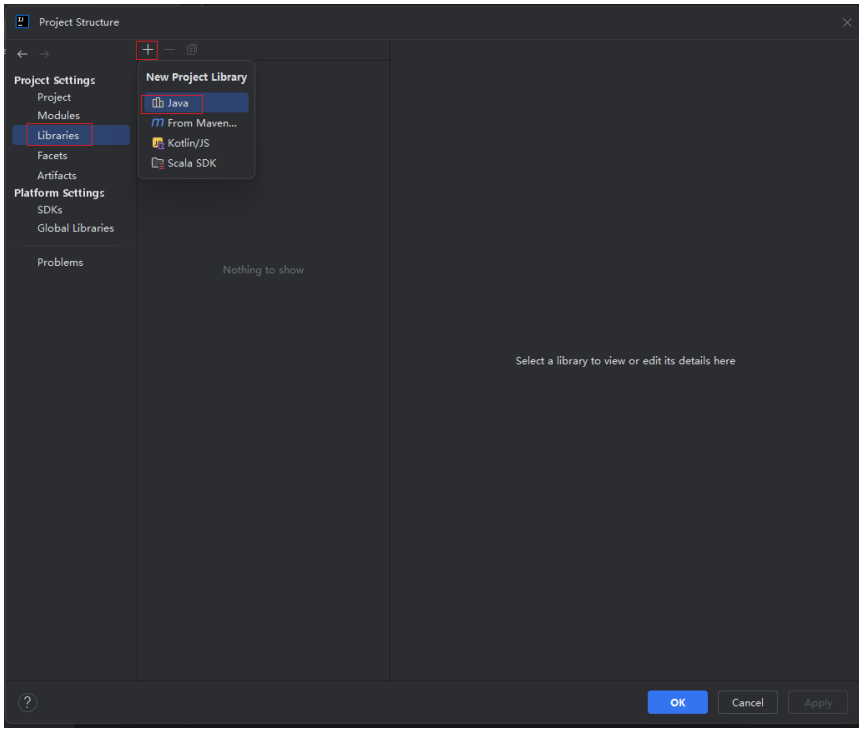
   Figure 4-69 Clicking Project Structure



   b. On the navigation pane, choose **Libraries**. Then click **+** and select **Java**.

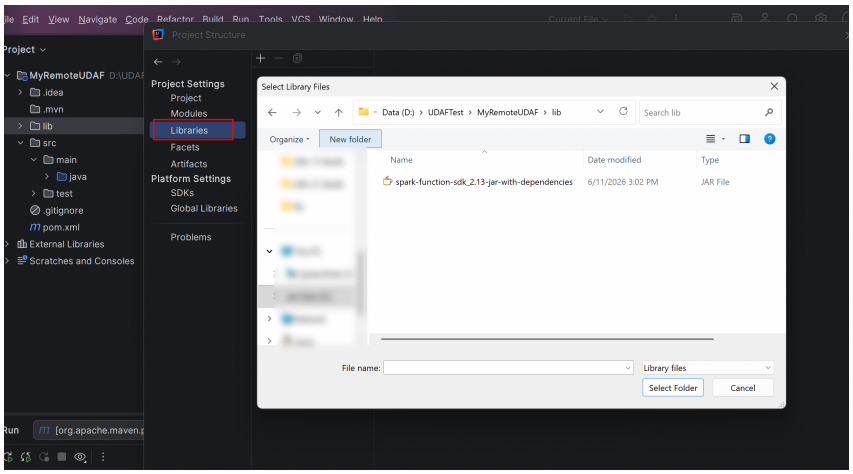
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-70 Clicking Java



c. Select the JAR package added to the **lib** folder and click **OK**.

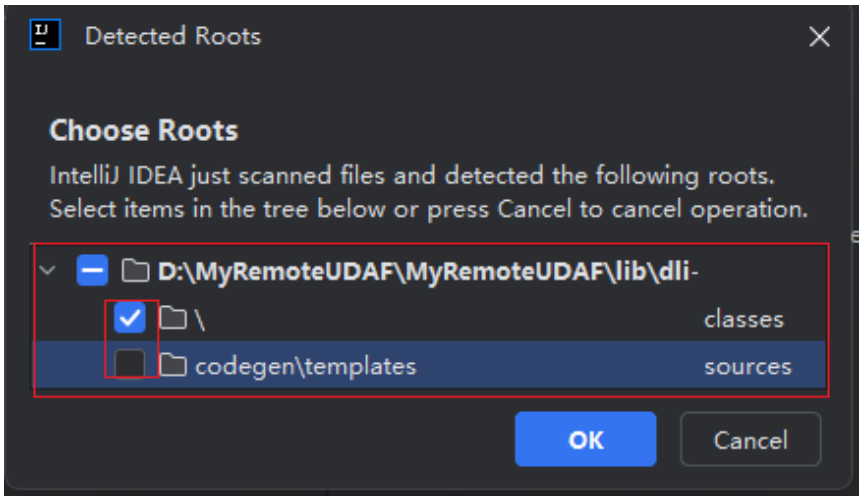
Figure 4-71 Selecting the JAR package added to the lib folder



d. On the displayed **Choose Roots** dialog box, select only the first folder and click **OK**.

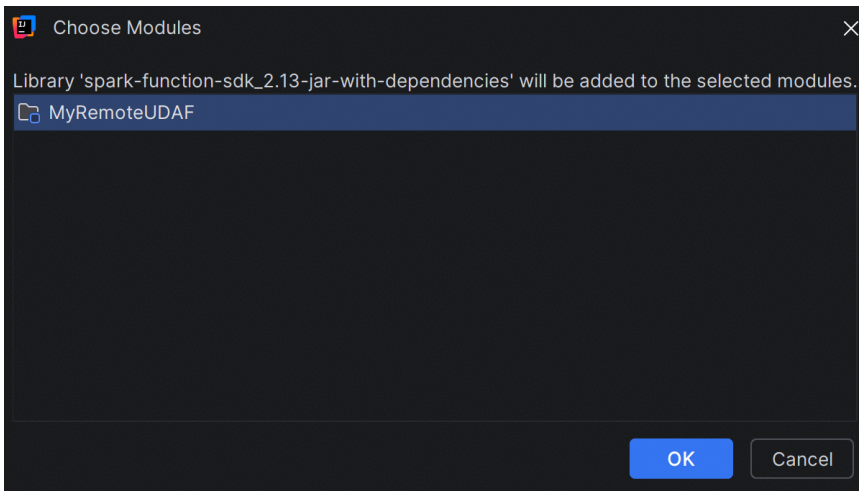
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-72 Choose Roots dialog box



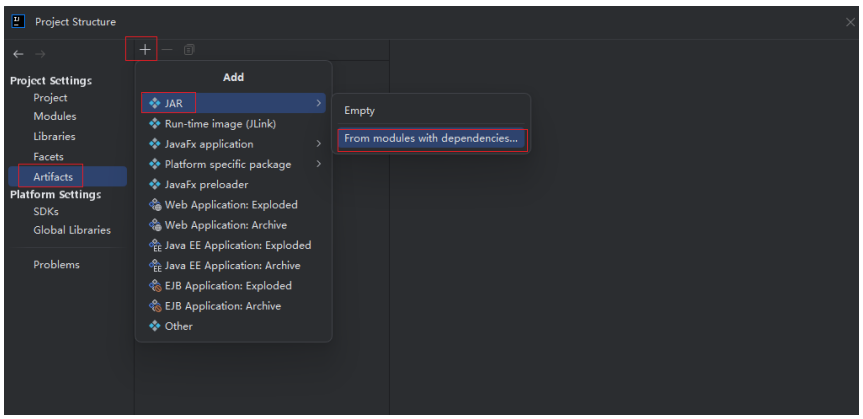
e. Click **OK** as prompted.

Figure 4-73 Clicking OK



9. Click **Artifacts**. On the displayed page, click **+** and then choose **JAR** > **From modules with dependencies**.

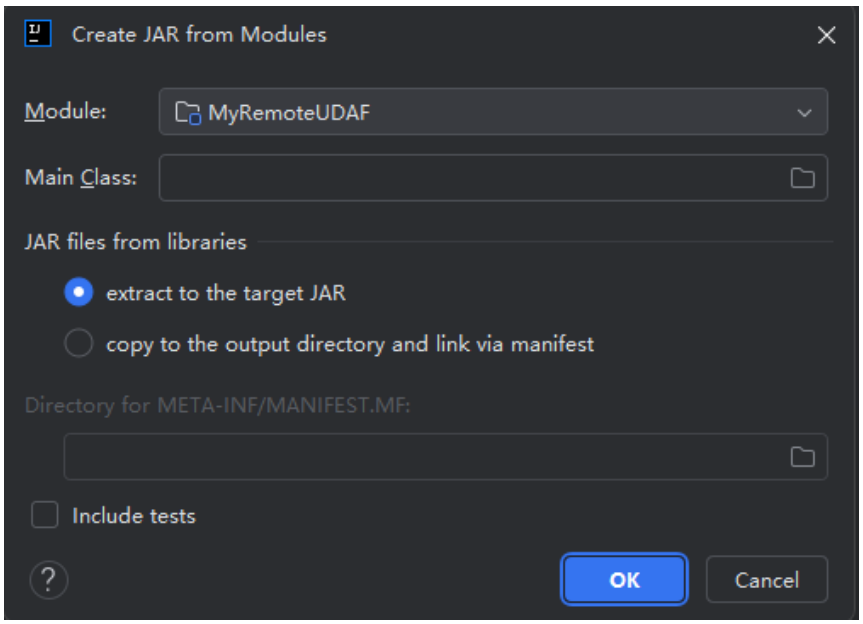
Figure 4-74 Choosing From modules with dependencies



 Copyright © Huawei Technologies Co., Ltd.

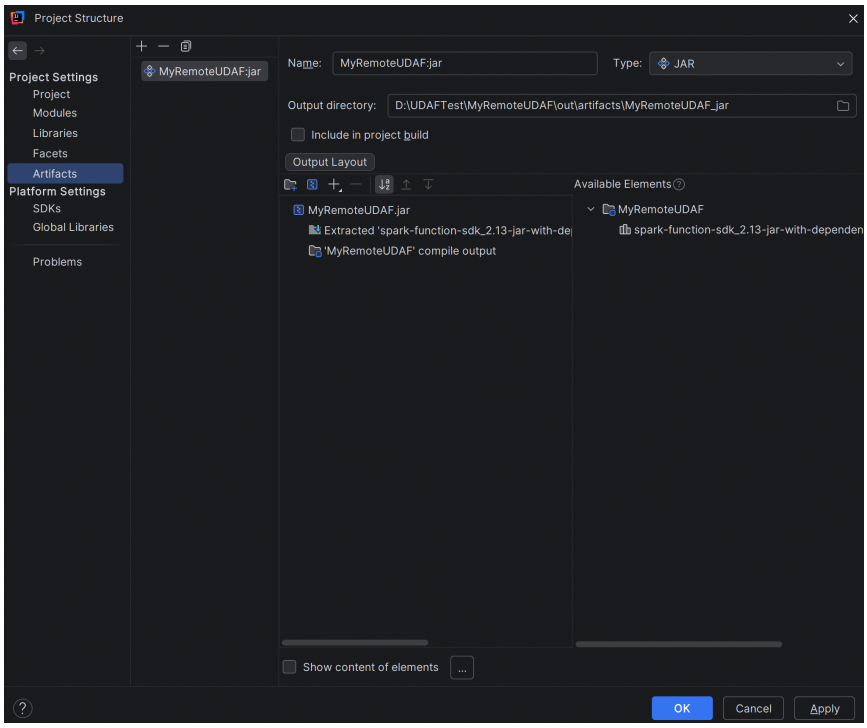
10. In the **JAR files from libraries** area, enable the first option and click **OK**.

Figure 4-75 Clicking OK



11. Click **Apply** and then **OK**.

Figure 4-76 Configuring Artifacts



4.5.3 Writing the UDAF Code and Exporting the JAR File

Step 1: Writing the UDAF Code

Pay attention to the following requirements when you implement the UDAF:

 Copyright © Huawei Technologies Co., Ltd.

- The UDAF must inherit the UDAFHandler class, write the UDAF function in the class, and implement the following methods:

```
public abstract Writable newBuffer();
public abstract void iterate(Writable buffer, Object[] args) throws FuncException;
public abstract void merge(Writable buffer, Writable partial) throws FuncException;
public abstract Object terminate(Writable buffer) throws FuncException;
```

- The @TypeResolve annotation is added to the method to indicate the input and output parameters of the function.

  Input and output parameters are separated by an arrow (->). For details about the input and output parameter types (**resolveType**) and the corresponding SQL and Java data types, see **Table 4-16**.

  Table 4-16 resolveType

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG
FloatType	Float.class	FLOAT
DoubleType	Double.class	DOUBLE
StringType	String.class	STRING
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	DECIMAL
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthInterval-Type	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- Add the @deterministic annotation.

 Copyright © Huawei Technologies Co., Ltd.

The @deterministic annotation indicates that the function is deterministic, that is, it always returns the same result for the same input.

For details about how to implement the UDAF, see the following sample code:

- Function description:

  This function is used to calculate the average value of data. The design process is as follows: Initialization > Iterative calculation > Result combination > Final calculation.

- Example code

```
package com.huawei.demo;

import com.huawei.spark.function.FuncException;
import com.huawei.spark.function.handlers.UDAFHandler;
import com.huawei.spark.function.requests.Writable;
import com.huawei.spark.function.types.TypeResolve;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

// get average
@TypeResolve("int->double")
public class UDAFDemo extends UDAFHandler {
    private static class Buffer implements Writable {
        private double sum = 0.d;

        private int count = 0;

        public Buffer() {
        }

        @Override
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeDouble(sum);
            out.writeInt(count);
        }

        @Override
        public void readExternal(ObjectInput in) throws IOException {
            sum = in.readDouble();
            count = in.readInt();
        }
    }

    public Writable newBuffer() {
        return new com.huawei.demo.UDAFDemo.Buffer();
    }

    public void iterate(Writable buffer, Object[] args) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        Integer arg = (Integer) args[0];
        buf.sum += arg;
        buf.count++;
    }

    public void merge(Writable buffer, Writable partial) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        com.huawei.demo.UDAFDemo.Buffer par = (com.huawei.demo.UDAFDemo.Buffer) partial;
        buf.sum += par.sum;
        buf.count += par.count;
    }

    public Object terminate(Writable buffer) throws FuncException {
        com.huawei.demo.UDAFDemo.Buffer buf = (com.huawei.demo.UDAFDemo.Buffer) buffer;
        if (buf.count == 0) {
            return 0;
        }
```

 Copyright © Huawei Technologies Co., Ltd.

```
        return buf.sum / buf.count;
    }
}
```

- Detailed description:
  - **newBuffer() initializes the buffer.**

    The Buffer class implements the Writable interface and is used to store intermediate results temporarily during aggregation. It supports serialization and deserialization.

    It contains two core variables:

    - **sum**: The cumulative total.
    - **count**: The cumulative count, which records the number of elements involved in the computation.

    It implements two serialization methods:

    - **writeExternal**: Writes the **sum** and **count** to the output stream.
    - **readExternal**: Reads **sum** and **count** from the input stream.
- **iterate(): Iterates the input data.**

  Purpose: Iterates over the input data and updates the intermediate state in the buffer.

  Parameters:

  - **buffer**: The current buffer.
  - **args**: An input parameter array, where **args[0]** is the integer to be calculated.

  Logic: Adds the input integer to **sum** and increments **count** by 1.
- **merge(): Merges intermediate results from distributed computations.**

  Purpose: Merges intermediate results from multiple subtasks in distributed computing scenarios.

  Parameters:

  - **buffer**: The main buffer, which is used to store the final merged result.
  - **partial**: The buffer of other subtasks, which is used to store the partial results to be merged.

  Logic: Adds the **sum** and **count** of each subtask to the main buffer.
- **terminate(): Calculates the final result.**

  Purpose: Calculates the final average value after all data is processed.

  Logic: Calculates the average by dividing the total **sum** by the total **count**. If there is no data (**count=0**), 0 is returned to avoid division by zero.

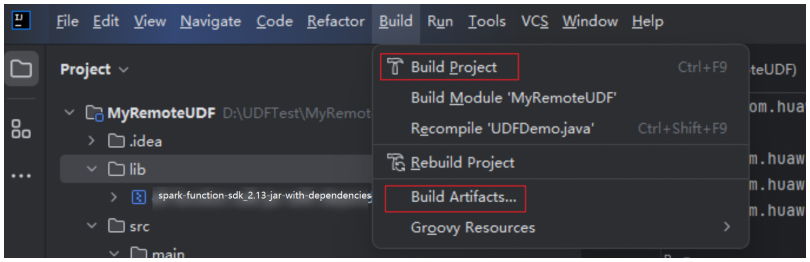
Step 2: Exporting the JAR Package

Use IntelliJ IDEA to compile the code and pack it into the JAR package.

1. Click **Build**. On the displayed page, click **Build Project** and **Build Artifacts...** to compile and pack the code, respectively.

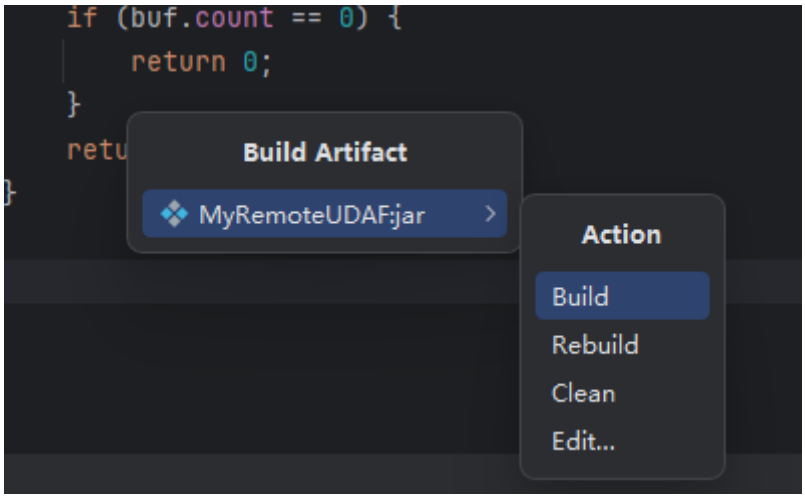
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-77 Compiling and packing



2. In the displayed dialog box, click **Build**.

Figure 4-78 Compiling and packing

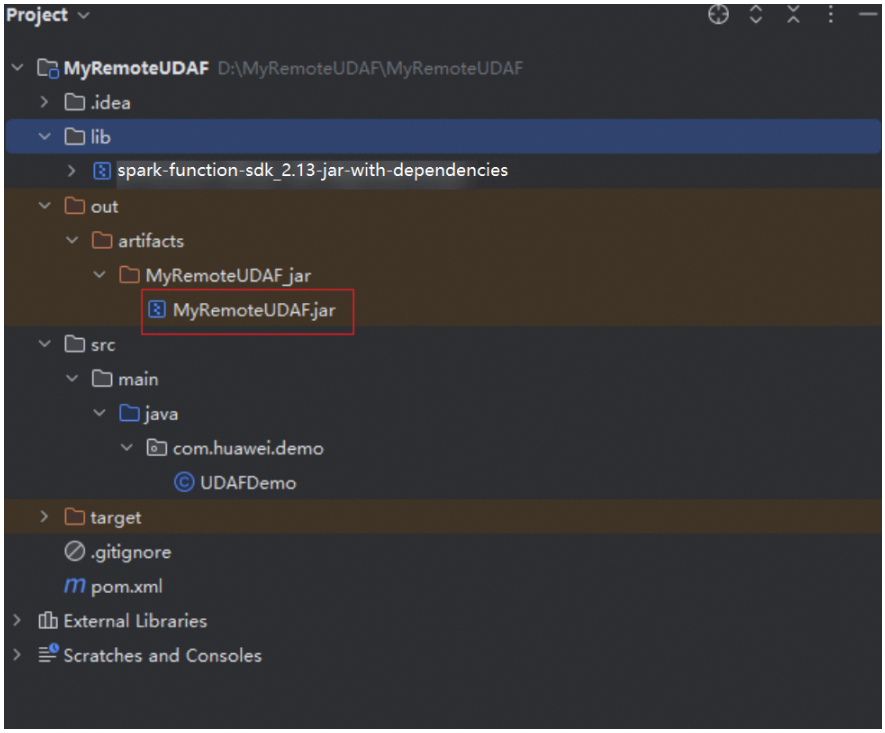


3. After the preceding steps are complete, locate the **out** folder in the project directory and expand it.

   The **MyRemoteUDAF.jar** file is the JAR file to be uploaded to FunctionGraph.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-79 Viewing MyRemoteUDAF.jar

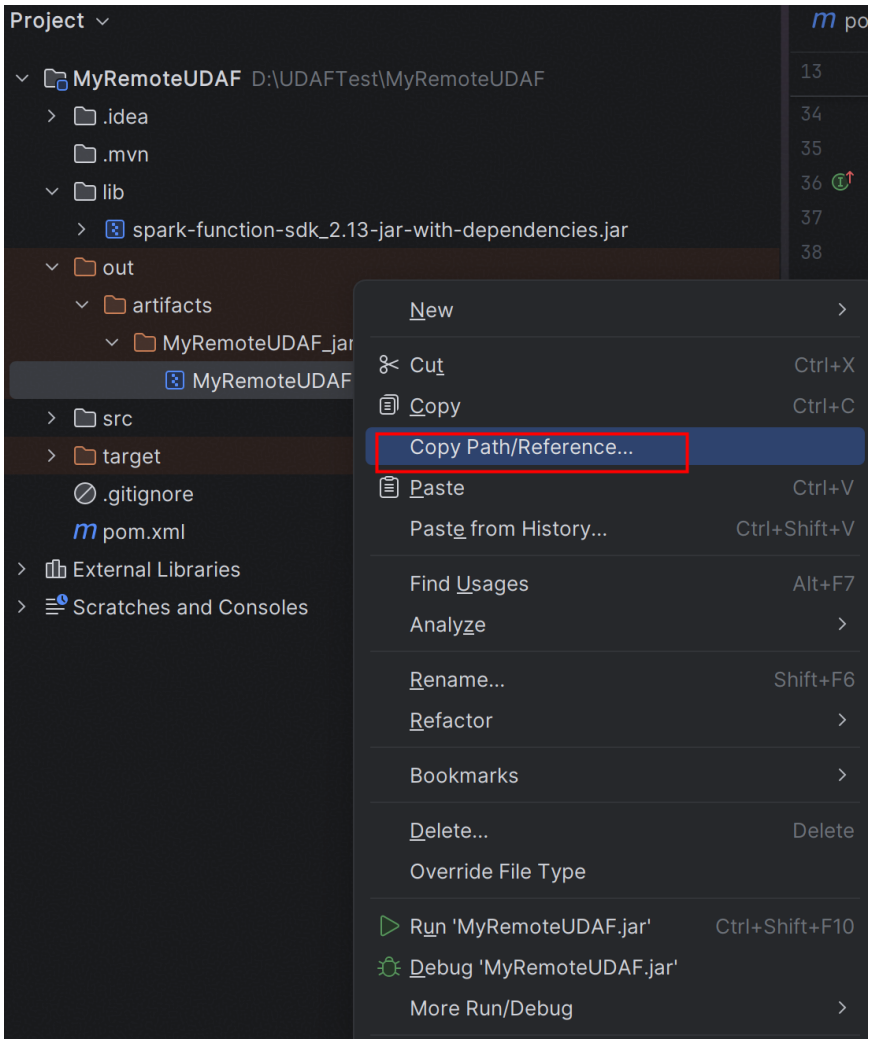


4. Right-click **MyRemoteUDAF.jar** and choose **Copy Path/Reference** from the shortcut menu to copy its path.

   In this example, the **MyRemoteUDAF.jar** file is generated in the **D:\UDAFTest\MyRemoteUDAF\out\artifacts\MyRemoteUDAF_jar** directory.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-80 Copying the file path



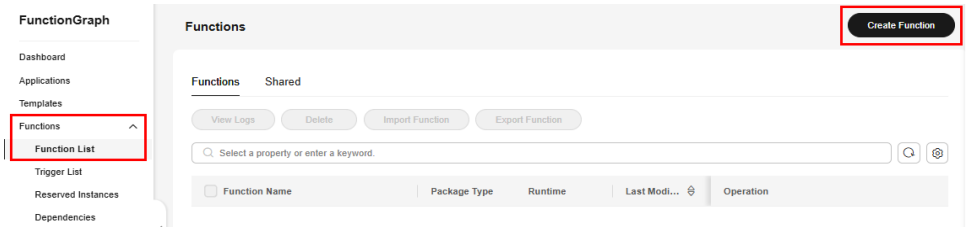
4.5.4 Creating a Function with FunctionGraph

Learn about the **process of using FunctionGraph**.

Step 1: Creating a Function

1. Log in to the FunctionGraph console.
2. In the navigation pane on the left, choose **Functions** > **Function List**. On the displayed page, click **Create Function**.

Figure 4-81 Creating a function



3. Click **Create from scratch** and configure required parameters.

 Copyright © Huawei Technologies Co., Ltd.

For details about the parameters, see **Creating an Event Function**.

- **Function Type**: Select **Event Function**.
- **Runtime**: Select Java 8.
- **Agency**: Select an agency to authorize FunctionGraph to access other cloud services, and configure agency permissions for functions.

  There are **constraints and restrictions** on the amount of data that can be processed by a FunctionGraph call at a time. If the function size exceeds 4 MB, you are advised to configure an agency to access OBS. For details, see **Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services**.

Step 2: Configuring the Function

1. **Set the function handler.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the UDAF handler in the **Handler** area.

      The UDAF handler is in the format of **[*Package name*].[*Class name*].[*Function name*]**.

      - [*Package name*].[*Class name*] indicates the class where the UDAF is located.
      - The function name is fixed to **handleRequest**.

      In this example, it is set to **com.huawei.demo.UDAFDemo.handleRequest**.
   c. After the setting is complete, click **Save**.
2. **Set the execution timeout interval and memory.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the execution time and memory.

      Configure the **execution timeout interval and memory size** based on the complexity of the UDAF. The memory size determines the **billing**.

      In this example, the execution time is set to 15 seconds and the memory is set to 1024 MB.
   c. After the setting is complete, click **Save**.
3. **Configure the class isolation.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Advanced Settings** and configure the **Class Isolation** option.

      After class isolation is enabled, an independent class loader is used to load the uploaded code and its dependencies, resolving dependency and runtime conflicts. You can disable class isolation if there is no dependency conflict.

 Copyright © Huawei Technologies Co., Ltd.

In this example, the class isolation is disabled. After the setting is complete, click **Save**.

Step 3: Uploading the JAR Package

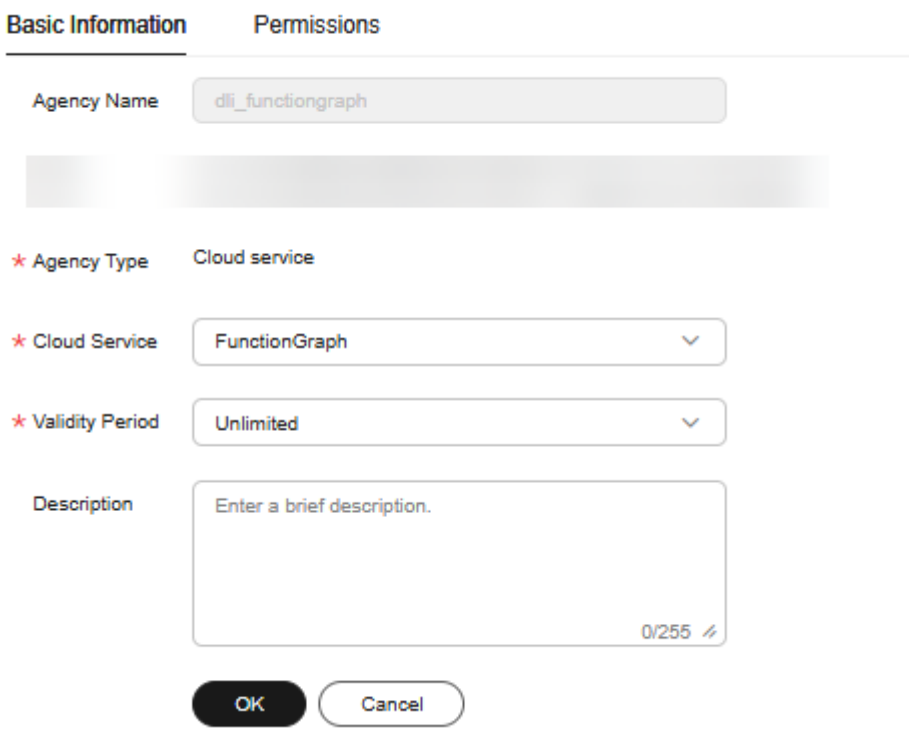
1. On the FunctionGraph console, click the target function name in the function list to go to its details page.
2. Click the **Code** tab, select **Local JAR** from the **Upload** drop-down list on the right, and upload the exported JAR package.
3. After the upload is successful, you can copy the Uniform Resource Name (URN) on the function details page. The URN uniquely identifies a function and will be used when the function is called.

Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services

1. Log in to the management console.
2. In the upper right corner of the page, hover over the username and select **Identity and Access Management**.
3. In the navigation pane of the IAM console, choose **Agencies**.
4. On the displayed page, click **Create Agency**.
5. Set the following parameters:
   - **Agency Name**: Enter an agency name.
   - **Agency Type**: Select **Cloud service**.
   - **Cloud Service**: This parameter is available only when you select **Cloud service** for **Agency Type**. Select **FunctionGraph** from the drop-down list.
   - **Validity Period**: Select **Unlimited**.
   - **Description**: You can enter **Agency with OBS OperateAccess permissions**. This parameter is optional.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-82 Creating an agency



6. After configuring the basic information, click **OK**.
7. On the displayed page, search for the OBS Administrator permission in the search box on the right and select it. Then, click **Next**.
8. Click **Next**. On the **Select Scope** page, set the authorization scope.

   For details about authorization operations, see **Creating a User Group and Assigning Permissions**.

   - **All resources**: IAM users will be able to use all resources, including those in enterprise projects, region-specific projects, and global services under your account based on assigned permissions.
   - **Global services**: Global services are deployed in all regions, so you can access them seamlessly without having to switch between regions. However, global services cannot be authorized based on regional projects. They include services like Object Storage Service (OBS) and Content Delivery Network (CDN). After authorization, you can use the selected services based on your permissions.
   - **Region-specific projects**: IAM users will be able to use resources in the selected region-specific projects based on assigned permissions.
   - **Enterprise projects**: IAM users will be able to use resources in the selected enterprise projects based on assigned permissions. For example, an enterprise project may contain resources that are deployed in different regions. After you associate the enterprise project with the IAM users, they can access the resources in this enterprise project based on the assigned permissions.

   In this example, an OBS policy is selected. Therefore, select **Global services**.
9. Click **OK**.

 Copyright © Huawei Technologies Co., Ltd.

10. It takes 15 to 30 minutes for the authorization to be in effect.

    After the authorization takes effect, return to the function details page, select the corresponding agency. The authorized policy is displayed.

4.5.5 Using Remote UDAFs

4.5.5.1 Creating a Function Using SQL

Description

Creates a Remote UDAF for use in Spark job development.

Syntax

```
CREATE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

or

```
CREATE OR REPLACE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

Requirements

None

Keywords

urn: On the FunctionGraph function list page, locate the function whose URN you want to copy and click **Copy URN** in the **Operation** column.

Example

Run the following command to create the UDAFDemo function:

```
CREATE FUNCTION UDAFDemo AS 'ExampleUDAFHandler' USING FUNCTIONGRAPH 'urn:fss:cn-north-7:330e068af1334c9782f4226acc00a2e2:function:default:test:latest'
```

Select the corresponding database and queue, enter the SQL statement, and click **Execute**.

4.5.5.2 Calling a Function Using SQL

Description

Calls a specified function.

Syntax

```
SELECT function_name(column_name) FROM table_name;
```

Requirements

None

 Copyright © Huawei Technologies Co., Ltd.

Keywords

function_name: The name of the function created on FunctionGraph.

Example

Call the sample function **remote_udaf** to return the average value of **real_stock_rate**.

```
SELECT remote_udaf(real_stock_rate)  FROM dw_ad_estimate_real_stock_rate LIMIT 1000;
```

4.5.5.3 Deleting a Function Using SQL

Description

Deletes a specified function.

Syntax

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

Requirements

None

Keywords

- **TEMPORARY**: Indicates whether the function to be deleted is a temporary function.
- **IF EXISTS**: Prevents an error message from being displayed if the function to be deleted does not exist.

Example

```
DROP FUNCTION remote_udaf;
```

4.5.5.4 Displaying All Functions

Description

Views all functions in the current project.

Syntax

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

Requirements

None

Keywords

LIKE: This qualifier is used only for compatibility and has no practical effect.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
SHOW FUNCTIONS;
```

4.5.5.5 Displaying Function Details

Description

Displays information about a specified function.

Syntax

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

Requirements

None

Keywords

EXTENDED: Displays extended usage information.

Example

```
DESCRIBE FUNCTION remote_udaf;
```

4.6 Calling Remote UDTFs in Spark SQL Jobs

4.6.1 Overview

Function Description

Remote User-Defined Table-Generating Functions (UDTFs) apply to one-in-multiple-out data operations. That is, remote UDTFs read a row of data and output multiple values as a table.

This section explains the steps to develop a remote UDTF and call it from a computing engine, using FunctionGraph as the execution platform for the remote UDTF.

Solution Benefits

The advantages of remote UDTFs include risk isolation, resource decoupling, and cost optimization.

- Risk isolation: Although technically feasible, executing UDTFs in critical production environments can introduce execution risks.

  Remote UDTFs offload the computing logic to FunctionGraph, maintaining service functionality while isolating risks through a serverless architecture.
- Resource decoupling: For resource-intensive tasks like large model inference, remote UDFs avoid long-term resource occupation in Spark clusters. This ensures stability, cost-effectiveness, and functionality by offloading heavy computations to FunctionGraph.

 Copyright © Huawei Technologies Co., Ltd.

- Cost optimization: Remote UDTFs turn the **default** queue into a pure data channel, removing risks caused by UDTFs and offering more cost-effective and secure core data flow services to users.

Environment Preparations

Before developing and debugging a remote UDTF using FunctionGraph, you must install IDEA, configure Maven, download required JAR files, and set up the FunctionGraph service.

Table 4-17 Development environment

Preparation Item	Description
Operating system (OS)	Windows 7 or a later version is required.
JDK	JDK 1.8 is required.
IntelliJ IDEA	IntelliJ IDEA 2019.1 or a compatible version is required.
Maven	Basic configurations of the development environment. Maven is used for project management throughout the lifecycle of software development.
Dependency JAR file download	Download the dependency JAR file **spark-function-sdk_2.13-jar-with-dependencies.jar** from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/**.

Comparison of UDTF and Remote UDTF

- UDTF: Ideal for high-performance scenarios where the development team is familiar with the query engine's language. It offers high execution efficiency but has lower flexibility and security.
- Remote UDTF: Best for scenarios that require external service calls, multi-language development, or high flexibility in the development language. It provides better security and greater development flexibility.

Table 4-18 Comparison of UDTF and remote UDTF

Feature	UDTF	Remote UDTF
Execution method	Runs locally on the query engine.	Executes through a function hosting service.
Security	Instability in UDTF code may affect the query engine process.	Execution failures only impact the UDTF itself, not the overall job process.
Performance	High execution efficiency	Comparable to local UDTFs in vectorized and batch modes.

 Copyright © Huawei Technologies Co., Ltd.

Feature	UDTF	Remote UDTF
Development flexibility	Highly coupled with query engine code, with limited flexibility.	Highly flexible and applicable to external services or libraries.
Development cost	Requires familiarity with the query engine, with high development cost.	Lower development cost due to reduced complexity.
Use case	High-performance scenarios where the team knows the query engine languages.	Scenarios needing flexible development languages or external service calls.

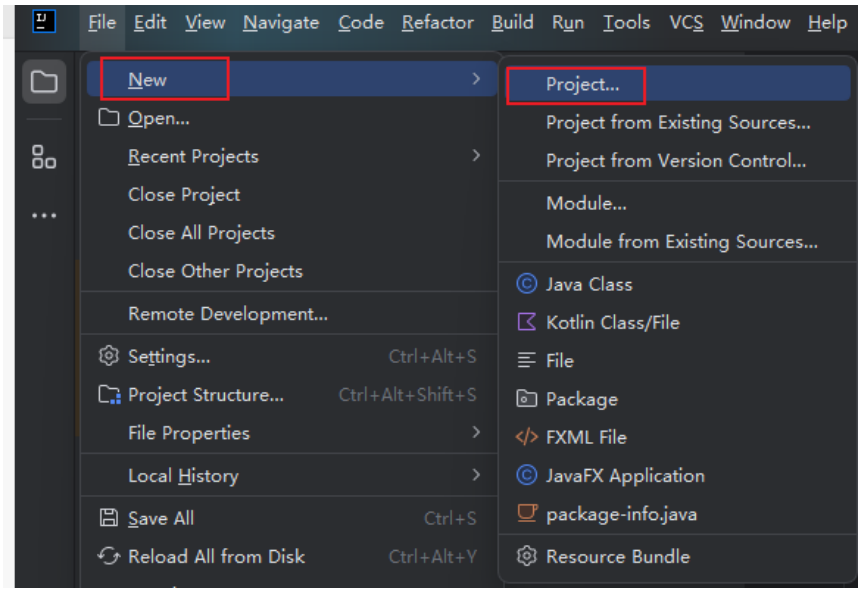
4.6.2 Creating a Maven Project and Configuring the Project Structure

Creating a Maven Project and Configuring the Project Structure

The following uses IntelliJ IDEA 2024.3.2.2 as an example.

1. Start IntelliJ IDEA and choose **File** > **New** > **Project**.

   Figure 4-83 Creating a project



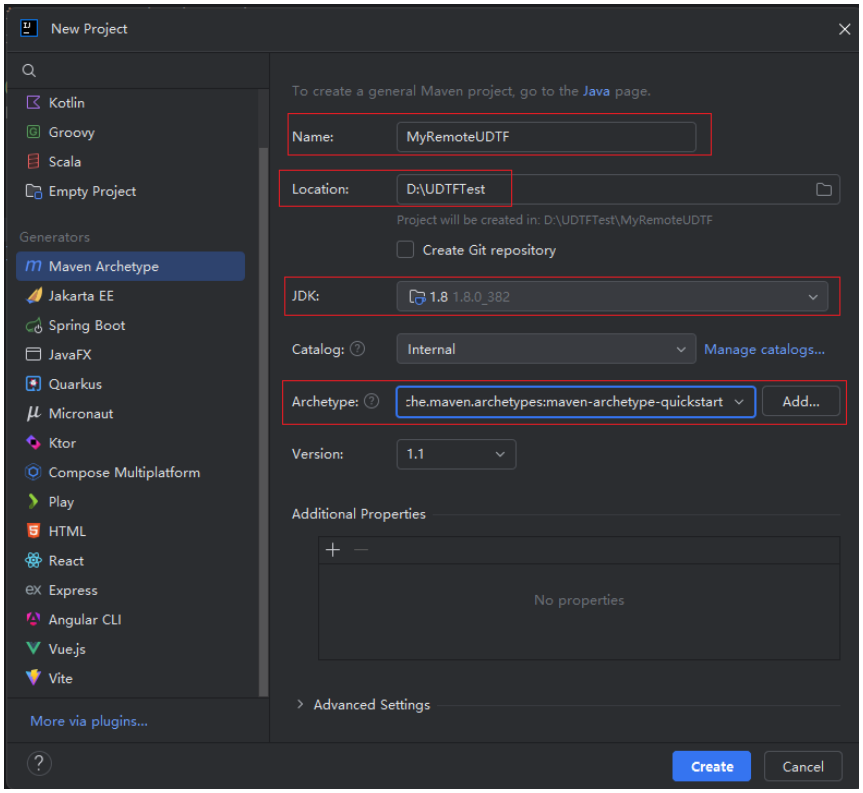
2. Choose **Maven Archetype** and configure the following parameters:
   - **Name**: Enter a project name. In this example, the project name is **MyRemoteUDTF**.
   - **Location**: Select a path for storing the project. In this example, the project is saved in **D:\UDTFTest**.
   - **JDK**: Select the JDK 1.8.

 Copyright © Huawei Technologies Co., Ltd.

– **Archetype**: Select **org.apache.maven.archetypes:maven-archetype-quickstart**.

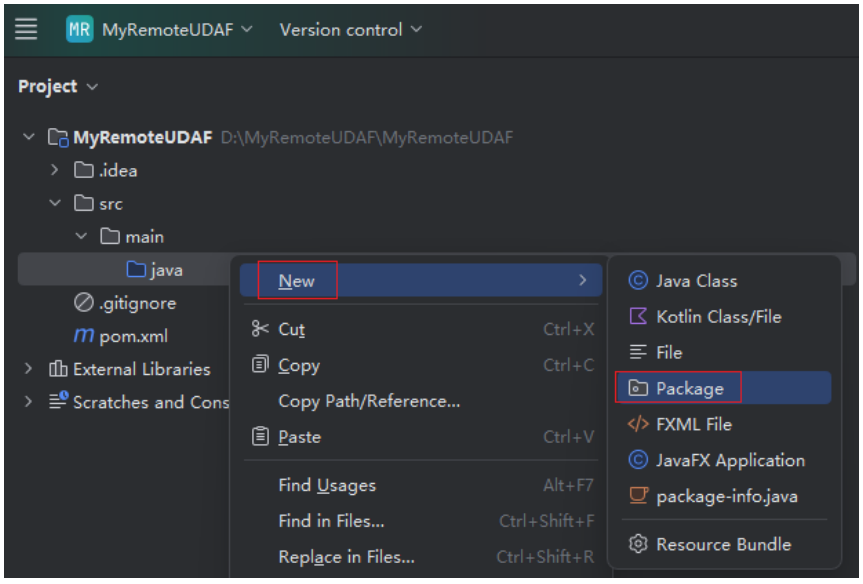
Click **Create** to create the project.

Figure 4-84 Creating a Maven project



3. Choose **src** > **main** and right-click the **java** folder. Choose **New** > **Package** to create a package.

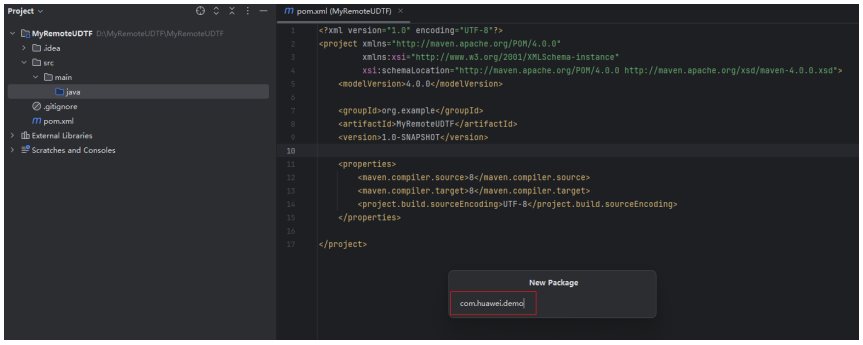
Figure 4-85 Creating a package



 Copyright © Huawei Technologies Co., Ltd.

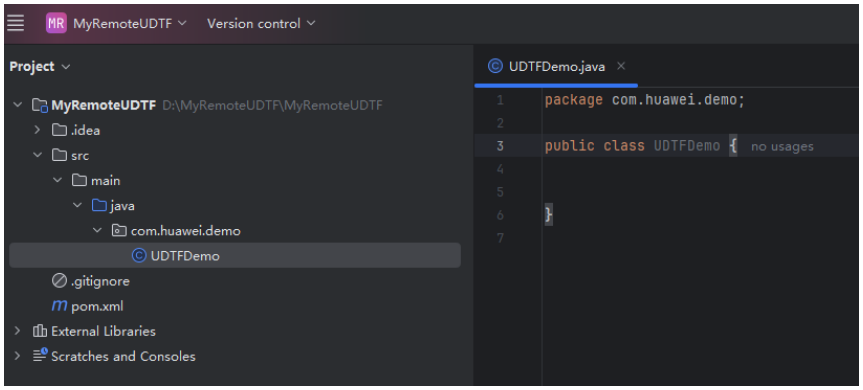
Set the package name as you need. In this example, the package name is **com.huawei.demo**. Then, press **Enter**.

Figure 4-86 Setting the package name



4. Create a Java class file in the package path. In this example, the file name is **UDTFDemo**.

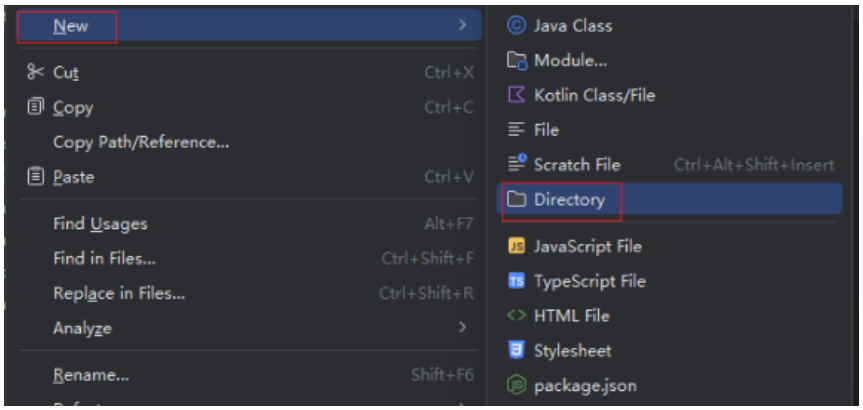
Figure 4-87 Creating a Java class file



5. Add the path for storing the JAR package.

   Click **File** > **New** > **Directory** to create a folder for storing the JAR package. In this example, the folder name is **lib**. Press **Enter** to confirm the name.

Figure 4-88 Adding the path for storing the JAR package



6. Import the JAR package.

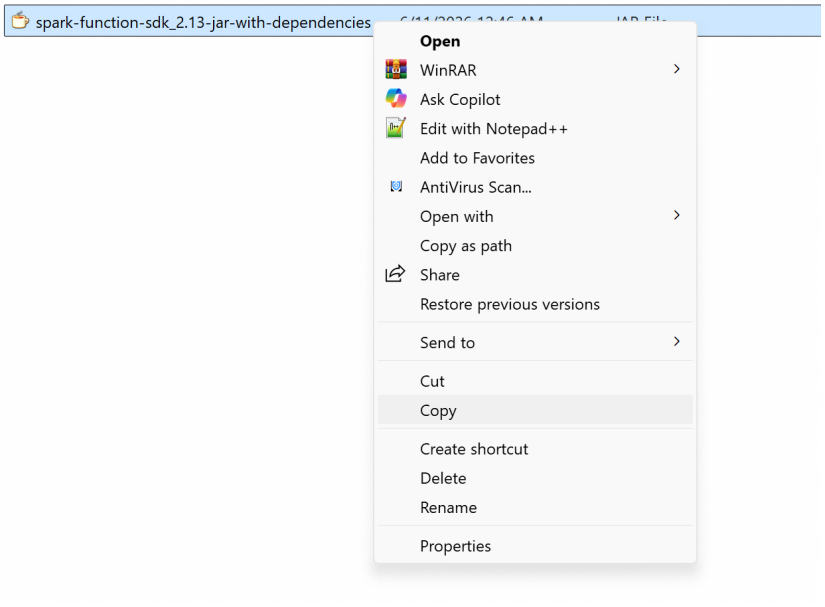
   a. Download the JAR package

 Copyright © Huawei Technologies Co., Ltd.

from **https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/com/huawei/mrs**.

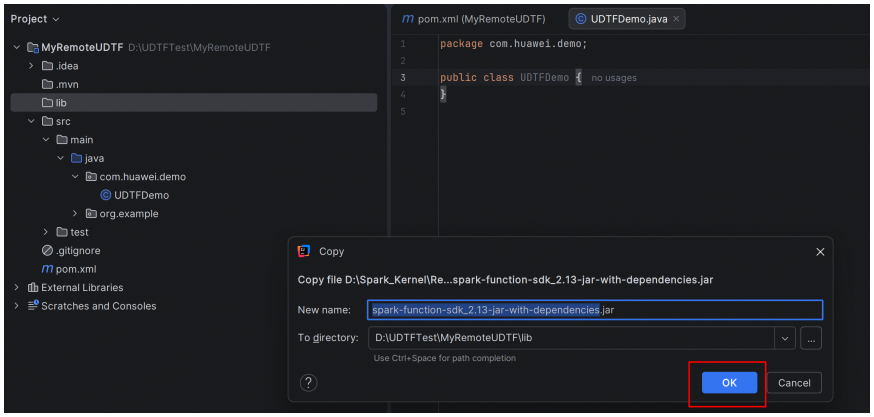
b. After the download is complete, right-click the JAR package and choose **Copy** from the shortcut menu.

Figure 4-89 Copying the JAR package



7. Locate the **lib** directory of the project, paste it to the created **lib** folder, and click **OK** in the dialog box that is displayed.

Figure 4-90 Pasting the JAR package to the lib folder

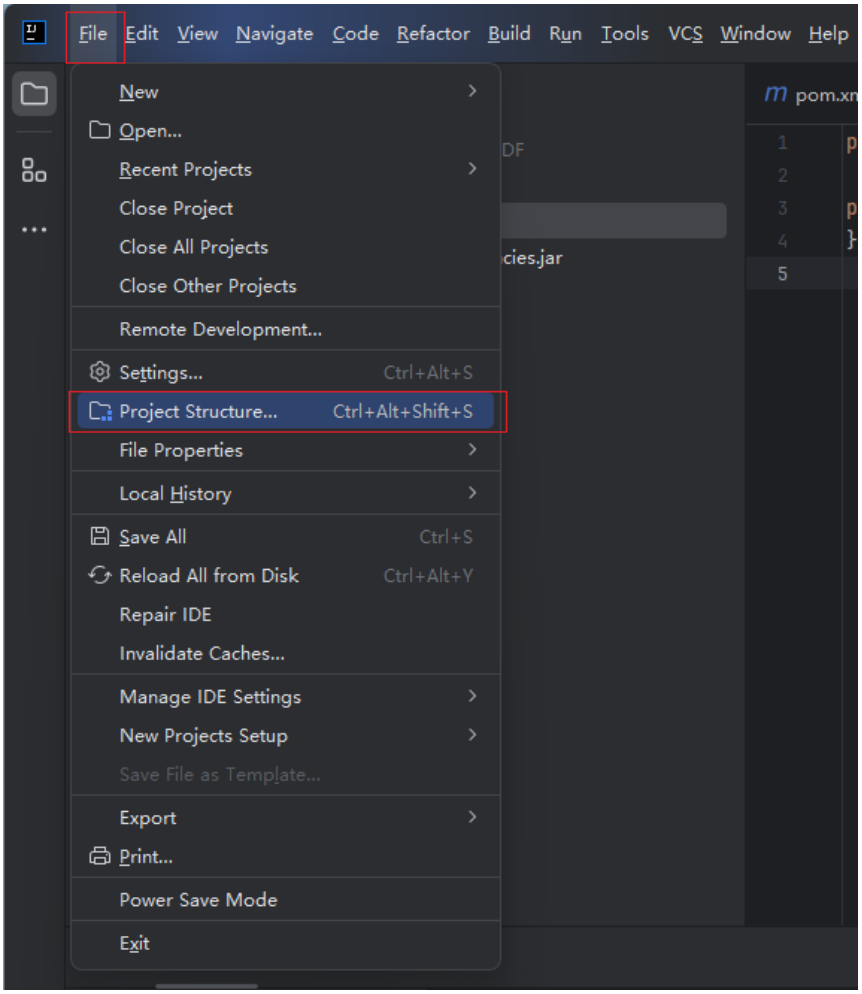


8. Configure **Project Structure**.

   a. Click **File** > **Project Structure**.

 Copyright © Huawei Technologies Co., Ltd.

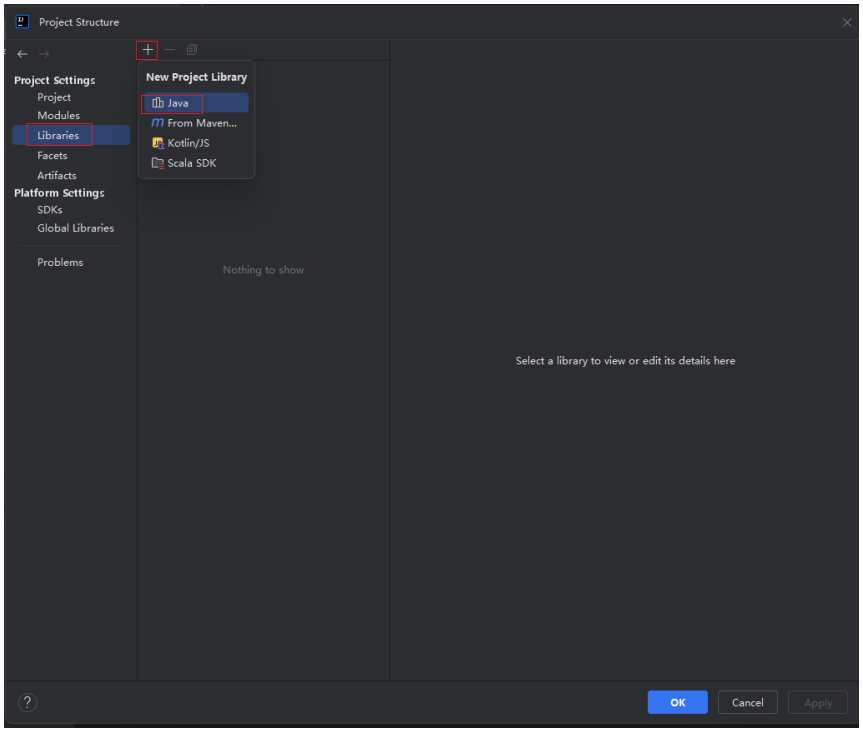
Figure 4-91 Clicking Project Structure



b. On the navigation pane, choose **Libraries**. Then click **+** and select **Java**.

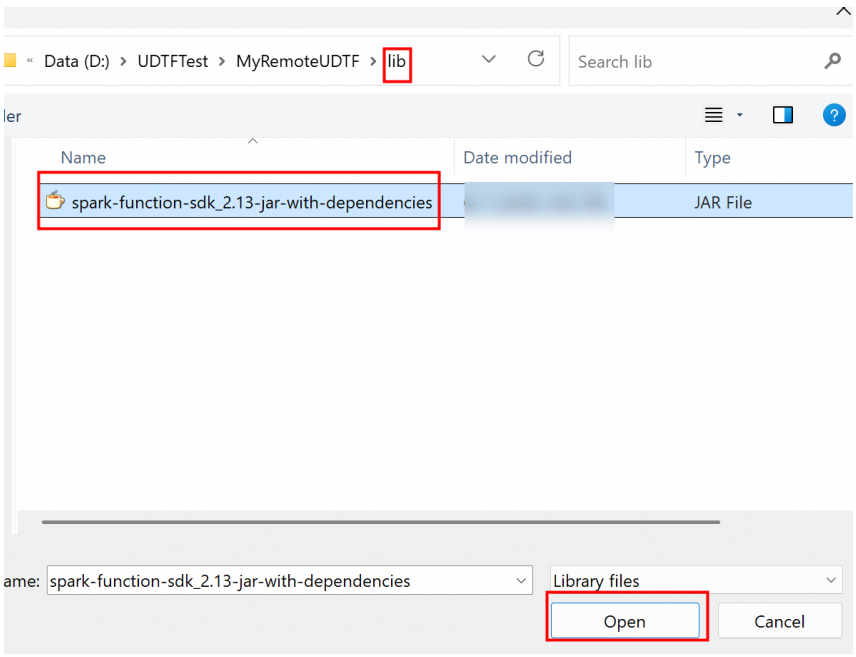
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-92 Clicking Java



c. Select the JAR package added to the **lib** folder and click **OK**.

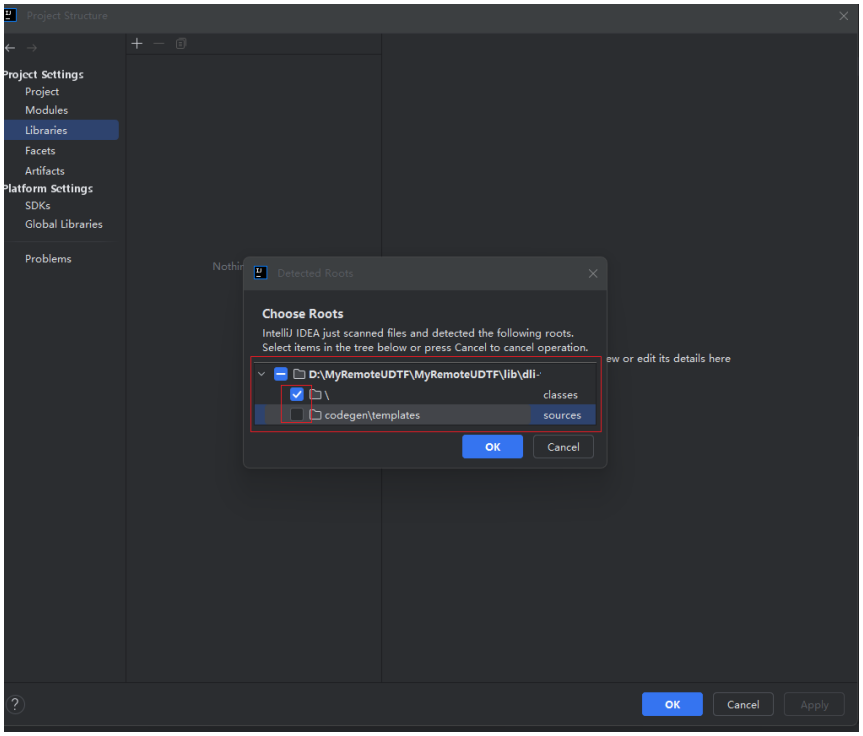
Figure 4-93 Selecting the JAR package added to the lib folder



d. On the displayed **Choose Roots** dialog box, select only the first folder and click **OK**.

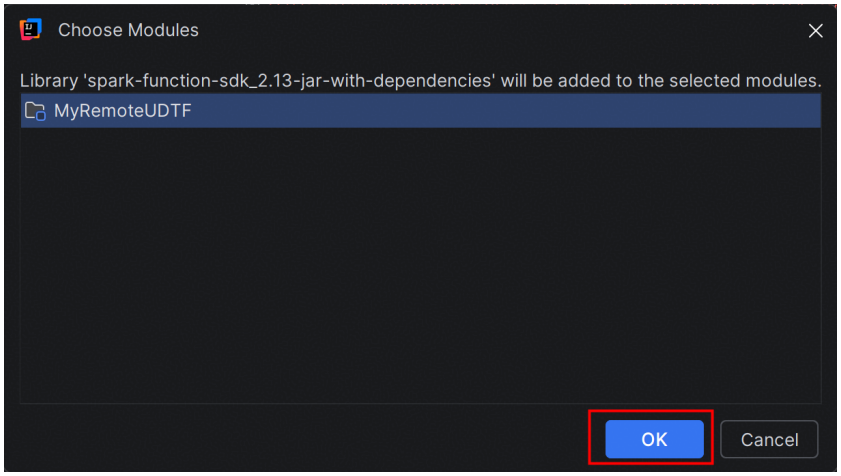
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-94 Choose Roots dialog box



e. Click **OK** as prompted.

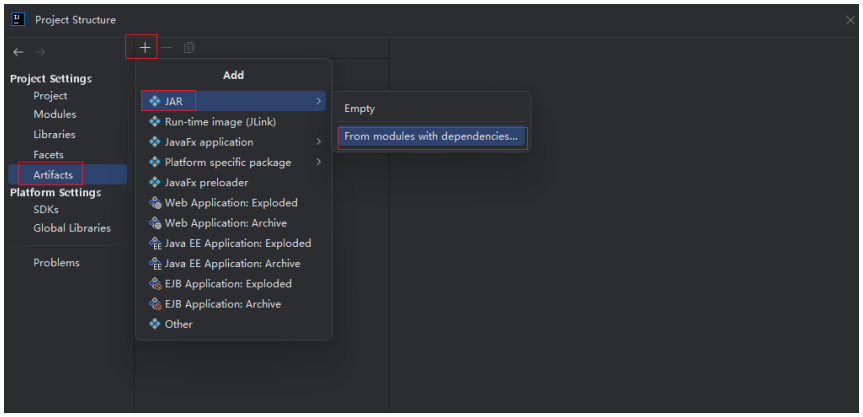
Figure 4-95 Clicking OK



9. Click **Artifacts**. On the displayed page, click **+** and then choose **JAR** > **From modules with dependencies**.

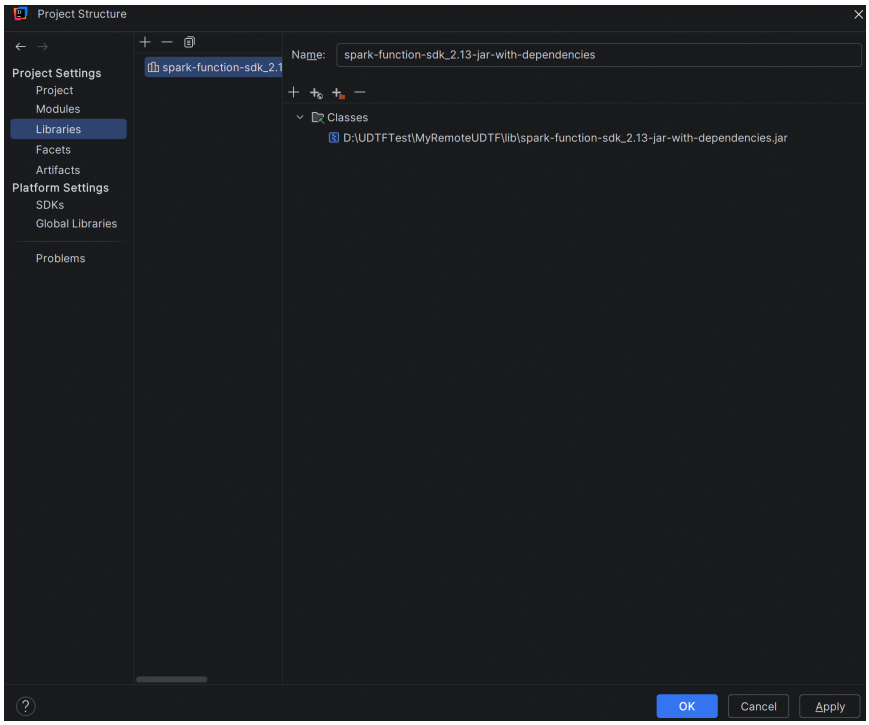
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-96 Choosing From modules with dependencies



Then, the following page is displayed.

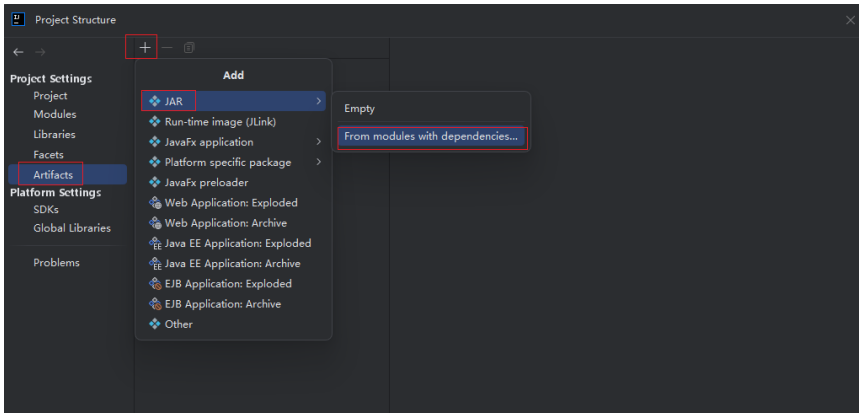
Figure 4-97 Page displayed after the addition is complete



10. Click **Artifacts**. On the displayed page, click **+** and then choose **JAR** > **From modules with dependencies**.

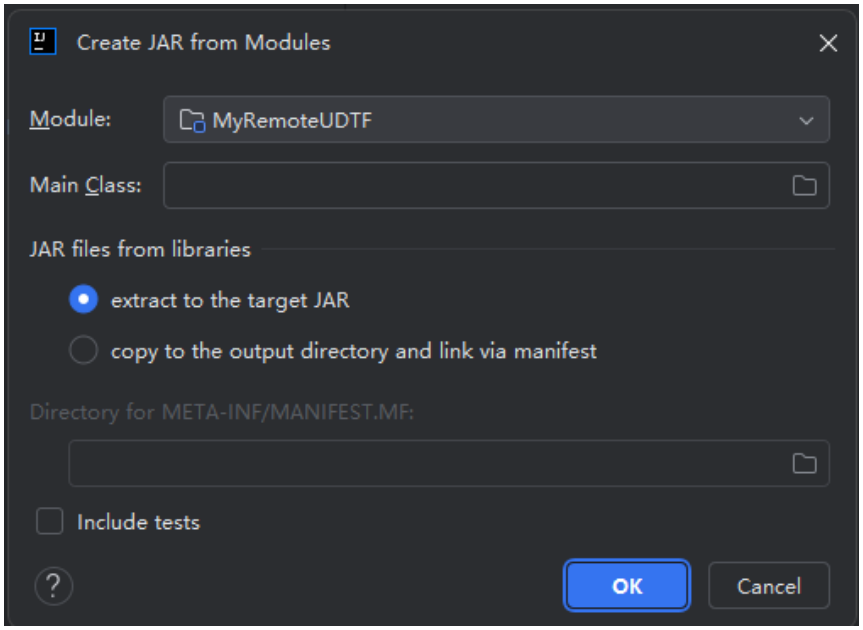
 Copyright © Huawei Technologies Co., Ltd.

Figure 4-98 Clicking Artifacts



11. In the **JAR files from libraries** area, enable the first option and click **OK**.

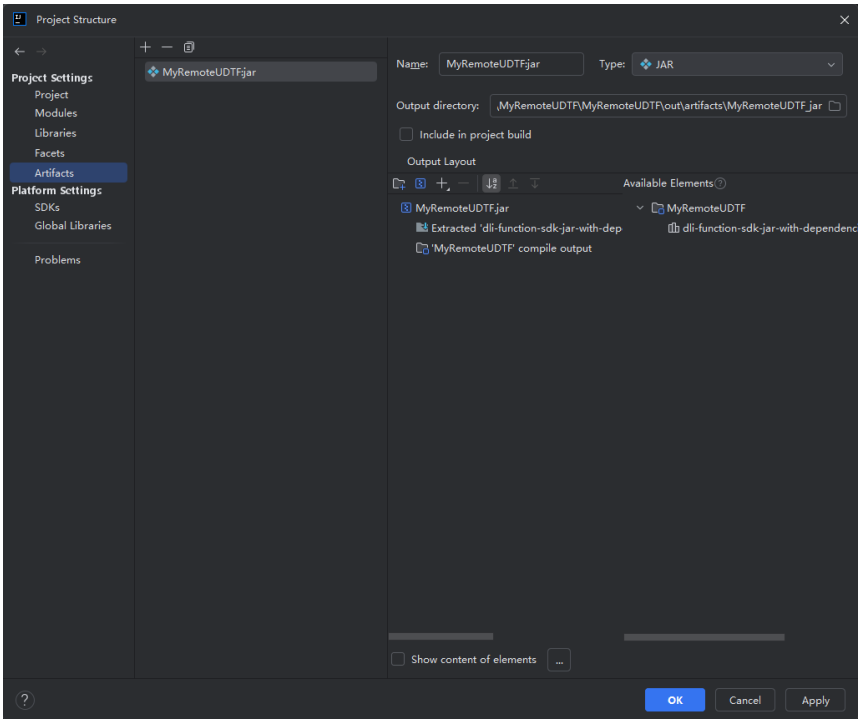
Figure 4-99 Clicking OK



12. Click **Apply** and then **OK**.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-100 Configuring Artifacts



4.6.3 Writing the UDTF Code and Exporting the JAR File

Step 1: Writing the UDTF Code

Pay attention to the following requirements when you implement the UDTF:

- The UDTF must inherit UDTFHandler.
- The @TypeResolve annotation is added to the method to indicate the input and output parameters of the function.

  Input and output parameters are separated by an arrow (->). For details about the input and output parameter types (**resolveType**) and the corresponding SQL and Java data types, see **Table 4-19**.

Table 4-19 resolveType

SQL type	Java type	resolveType
BooleanType	Boolean.class	BOOLEAN
ByteType	Byte.class	BYTE
ShortType	Short.class	SHORT
IntegerType	Integer.class	INT
LongType	Long.class	LONG
FloatType	Float.class	FLOAT
DoubleType	Double.class	DOUBLE

 Copyright © Huawei Technologies Co., Ltd.

SQL type	Java type	resolveType
StringType	String.class	STRING
BinaryType	byte[].class	BINARY
DecimalType.Fixed(precision, scale)	BigDecimal.class	DECIMAL
DateType	LocalDate.class	DATE
TimestampType	ZonedDateTime.class	TIMESTAMP
TimestampNTZType	LocalDateTime.class	TIMESTAMP_NTZ
YearMonthInterval-Type	Period.class	INTERVAL_YEAR_MONTH
DayTimeIntervalType	Duration.class	INTERVAL_DAY_TIME
List	List.class	LIST
Map	Map.class	MAP
Struct	Map.class	STRUCT

- Add the @deterministic annotation.

  The @deterministic annotation is a Spark scheduling parameter that indicates which a function is deterministic. A deterministic function always returns the same result for the same input. This annotation can be applied at both the class level and the function level, with the function-level annotation taking precedence over the class-level annotation.

  For details about how to implement the UDTF, see the following sample code:

  – Function description:

    The function splits a string and generates a result set. It takes two string parameters: **body** and **split**.

    The main functionality is to split a **body** string using the delimiter specified by **split**, and combine each result substring with a fixed version tag (**version**, with the value **tag**).

    The function returns a two-dimensional array of objects, where each element is an array containing a split substring and a version tag.

  – Example code

```
package com.huawei.demo;

import com.huawei.spark.function.handlers.UDTFHandler;
import com.huawei.spark.function.types.FunctionType;
import com.huawei.spark.function.types.TypeResolve;


public class UDTFDemo extends UDTFHandler {
    @FunctionType(deterministic = true)
    @TypeResolve("string,string->string,string")
    public Object[][] splitString(Object[] args) {
        String body = (String) args[0];
```

 Copyright © Huawei Technologies Co., Ltd.

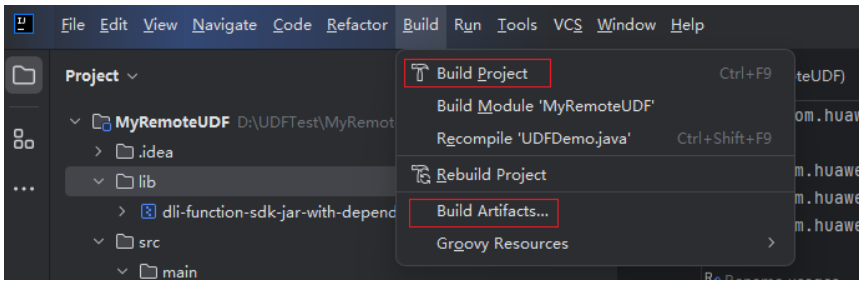
```
        String split = (String) args[1];
        String[] value = body.split(split);
        Object[][] result = new Object[value.length][];
        String version = "tag";
        for (int i = 0; i < value.length; i++) {
            result[i] = new Object[] {value[i], version};
        }
        return result;
    }
}
```

Step 2: Exporting the JAR Package

Use IntelliJ IDEA to compile the code and pack it into the JAR package.

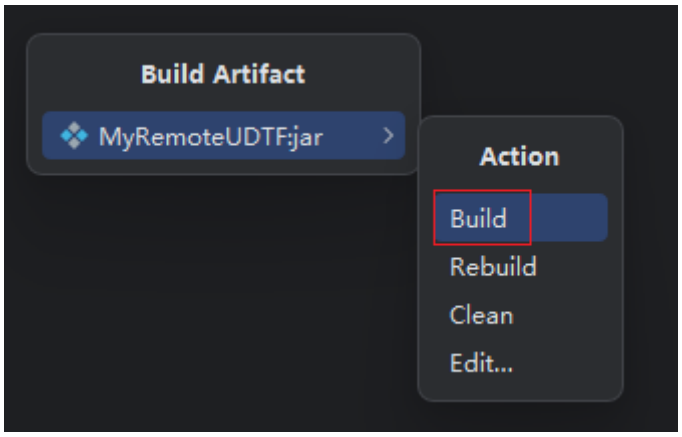
1. Click **Build**. On the displayed page, click **Build Project** and **Build Artifacts...** to compile and pack the code, respectively.

   Figure 4-101 Compiling and packing

2. In the displayed dialog box, click **Build**.

   Figure 4-102 Compiling and packing

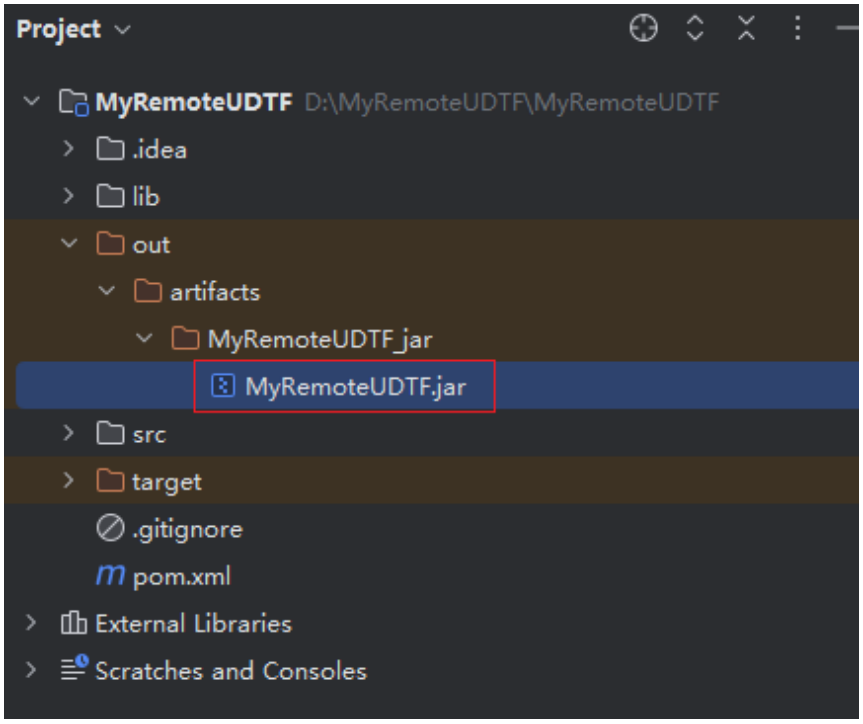
   

3. After the preceding steps are complete, locate the **out** folder in the project directory and expand it.

   The **MyRemoteUDTF.jar** file is the JAR file to be uploaded to FunctionGraph.

 Copyright © Huawei Technologies Co., Ltd.

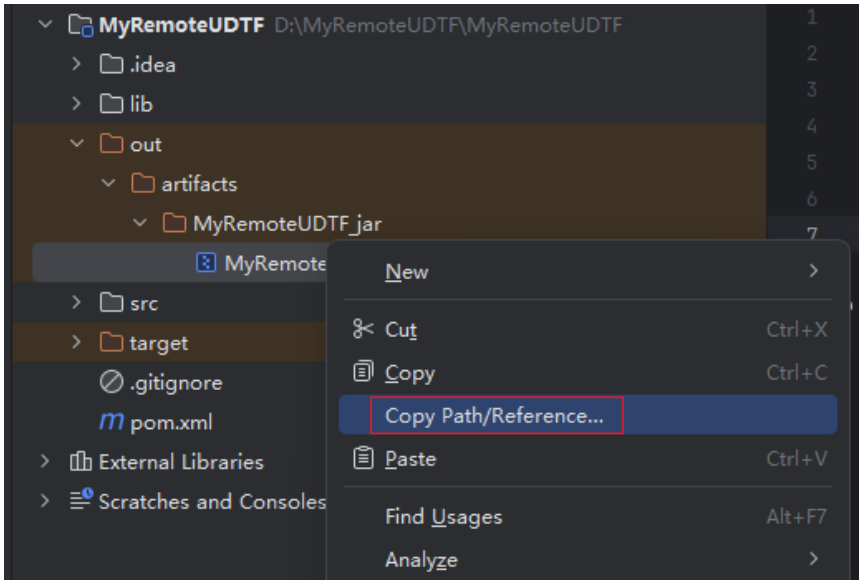
Figure 4-103 Viewing MyRemoteUDTF.jar



4. Right-click **MyRemoteUDTF.jar** and choose **Copy Path/Reference** from the shortcut menu to copy its path.

   In this example, the **MyRemoteUDTF.jar** file is generated in the **D:\UDTFTest\MyRemoteUDTF\out\artifacts\MyRemoteUDTF_jar** directory.

Figure 4-104 Copying the file path



4.6.4 Creating a Function with FunctionGraph

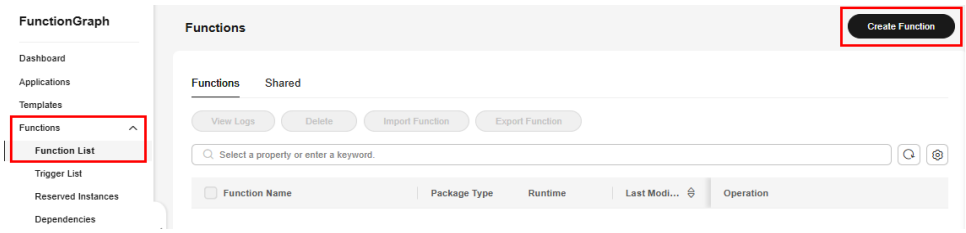
Learn about the **process of using FunctionGraph**.

 Copyright © Huawei Technologies Co., Ltd.

Step 1: Creating a Function

1. Log in to the FunctionGraph console.
2. In the navigation pane on the left, choose **Functions** > **Function List**. On the displayed page, click **Create Function**.

   Figure 4-105 Creating a function



3. Click **Create from scratch** and configure required parameters.

   For details about the parameters, see **Creating an Event Function**.

   - **Function Type**: Select **Event Function**.
   - **Runtime**: Select Java 8.
   - **Agency**: Select an agency to authorize FunctionGraph to access other cloud services, and configure agency permissions for functions.

     There are **constraints and restrictions** on the amount of data that can be processed by a FunctionGraph call at a time. If the function size exceeds 4 MB, you are advised to configure an agency to access OBS. For details, see **Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services**.

Step 2: Configuring the Function

1. **Set the function handler.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the UDTF handler in the **Handler** area.

      The UDTF handler is in the format of **[*Package name*].[*Class name*].[*Function name*]**.

      - [*Package name*].[*Class name*] indicates the class where the UDTF is located.
      - The function name is fixed to **handleRequest**.

      In this example, it is set to **com.huawei.demo.UDTFDemo.handleRequest**.
   c. After the setting is complete, click **Save**.
2. **Set the execution timeout interval and memory.**
   a. After the function is created, click the function name in the list to go to its details page.
   b. Choose **Configuration** > **Basic Settings** and set the execution time and memory.

 Copyright © Huawei Technologies Co., Ltd.

Configure the **execution timeout interval and memory size** based on the complexity of the UDTF. The memory size determines the **billing**.

In this example, the execution time is set to 15 seconds and the memory is set to 1024 MB.

c. After the setting is complete, click **Save**.

3. **Configure the class isolation.**

a. After the function is created, click the function name in the list to go to its details page.

b. Choose **Configuration** > **Advanced Settings** and configure the **Class Isolation** option.

After class isolation is enabled, an independent class loader is used to load the uploaded code and its dependencies, resolving dependency and runtime conflicts. You can disable class isolation if there is no dependency conflict.

In this example, the class isolation is disabled. After the setting is complete, click **Save**.

Step 3: Uploading the JAR Package

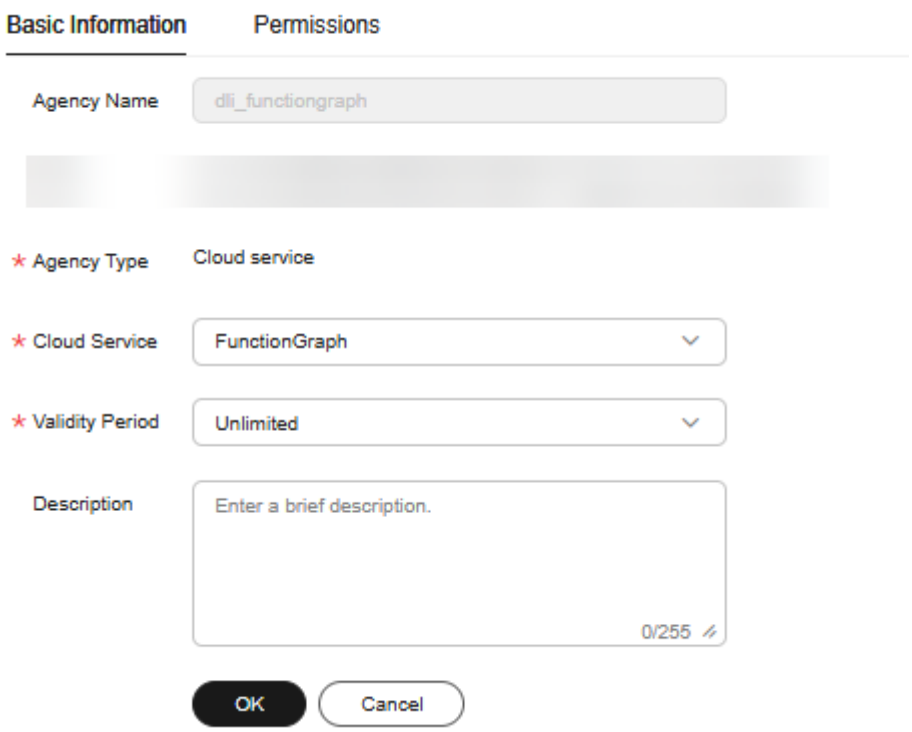
1. On the FunctionGraph console, click the target function name in the function list to go to its details page.
2. Click the **Code** tab, select **Local JAR** from the **Upload** drop-down list on the right, and upload the exported JAR package.
3. After the upload is successful, you can copy the Uniform Resource Name (URN) on the function details page. The URN uniquely identifies a function and will be used when the function is called.

Related Operation: Configuring an Agency for FunctionGraph to Access Other Cloud Services

1. Log in to the management console.
2. In the upper right corner of the page, hover over the username and select **Identity and Access Management**.
3. In the navigation pane of the IAM console, choose **Agencies**.
4. On the displayed page, click **Create Agency**.
5. Set the following parameters:
   - **Agency Name**: Enter an agency name.
   - **Agency Type**: Select **Cloud service**.
   - **Cloud Service**: Select **FunctionGraph** from the drop-down list. This parameter is available only when you select **Cloud service** for **Agency Type**.
   - **Validity Period**: Select **Unlimited**.
   - **Description**: You can enter **Agency with OBS OperateAccess permissions**. This parameter is optional.

 Copyright © Huawei Technologies Co., Ltd.

Figure 4-106 Creating an agency



6. After configuring the basic information, click **OK**.
7. On the displayed page, search for the OBS Administrator permission in the search box on the right and select it. Then, click **Next**.
8. Click **Next**. On the **Select Scope** page, set the authorization scope.

   For details about authorization operations, see **Creating a User Group and Assigning Permissions**.

   - **All resources**: IAM users will be able to use all resources, including those in enterprise projects, region-specific projects, and global services under your account based on assigned permissions.
   - **Global services**: Global services are deployed in all regions, so you can access them seamlessly without having to switch between regions. However, global services cannot be authorized based on regional projects. They include services like Object Storage Service (OBS) and Content Delivery Network (CDN). After authorization, you can use the selected services based on your permissions.
   - **Region-specific projects**: IAM users will be able to use resources in the selected region-specific projects based on assigned permissions.
   - **Enterprise projects**: IAM users will be able to use resources in the selected enterprise projects based on assigned permissions. For example, an enterprise project may contain resources that are deployed in different regions. After you associate the enterprise project with the IAM users, they can access the resources in this enterprise project based on the assigned permissions.

   In this example, an OBS policy is selected. Therefore, select **Global services**.
9. Click **OK**.

 Copyright © Huawei Technologies Co., Ltd.

10. It takes 15 to 30 minutes for the authorization to be in effect.

    After the authorization takes effect, return to the function details page, select the corresponding agency. The authorized policy is displayed.

4.6.5 Using Remote UDTFs

4.6.5.1 Creating a Function Using SQL

Description

Creates a Remote UDTF for use in Spark job development.

Syntax

```
CREATE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

or

```
CREATE OR REPLACE FUNCTION [db_name.]function_name AS class_name USING FUNCTIONGRAPH 'urn'
```

Requirements

None

Keywords

urn: On the FunctionGraph function list page, locate the function whose URN you want to copy and click **Copy URN** in the **Operation** column.

Example

Run the following command to create the **remote_udtf** function:

```
CREATE FUNCTION remote_udtf AS 'splitString' USING FUNCTIONGRAPH 'urn:fss:cn-north-7:330e068af1334c9782f4226acc00a2e2:function:default:test:latest'
```

Select the corresponding database and queue, enter the SQL statement, and click **Execute**.

4.6.5.2 Calling a Function Using SQL

Description

Calls a specified function.

Syntax

```
SELECT column_list, column_tmp  FROM table_name LATERAL VIEW function_name(column_list, ',') t AS column_tmp;
```

Requirements

None

 Copyright © Huawei Technologies Co., Ltd.

Keywords

function_name: The name of the function created on FunctionGraph.

Example

Call the sample function **remote_udtf**.

```
SELECT remote_udtf("a_b_c", '_') ;
```

Select the corresponding database and queue, enter the SQL statement, and click **Execute**.

4.6.5.3 Deleting a Function Using SQL

Description

Deletes a specified function.

Syntax

```
DROP [TEMPORARY] FUNCTION [IF EXISTS] [db_name.] function_name;
```

Requirements

None

Keywords

- **TEMPORARY**: Indicates whether the function to be deleted is a temporary function.
- **IF EXISTS**: Prevents an error message from being displayed if the function to be deleted does not exist.

Example

```
DROP FUNCTION remote_udtf;
```

4.6.5.4 Displaying All Functions

Description

Views all functions in the current project.

Syntax

```
SHOW [USER|SYSTEM|ALL] FUNCTIONS ([LIKE] regex | [db_name.] function_name);
```

Requirements

None

Keywords

LIKE: This qualifier is used only for compatibility and has no practical effect.

 Copyright © Huawei Technologies Co., Ltd.

Example

```
SHOW FUNCTIONS;
```

Execution result:

4.6.5.5 Displaying Function Details

Description

Displays information about a specified function.

Syntax

```
DESCRIBE FUNCTION [EXTENDED] [db_name.] function_name;
```

Requirements

UDTFs cannot be nested.

Keywords

EXTENDED: Displays extended usage information.

Example

```
DESCRIBE FUNCTION remote_udtf;
```

Execution result:

 Copyright © Huawei Technologies Co., Ltd.