AI DataLake

SDK Reference

Issue 01

Date 2026-11-09



HUAWEI

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 AI DataLake SDK....1
2 Engine-specific SDKs....4
2.1 Python SDKs....4
2.1.1 Python SDK Overview....4
2.1.2 Python Environment Configuration....4
2.1.3 AK/SK Obtaining....5
2.1.4 Catalog SDK....5
2.1.4.1 Catalog SDK Overview....6
2.1.4.2 Obtaining and Installing the Catalog SDK....7
2.1.4.3 Initializing the Client....7
2.1.4.4 Running the Example with AuraJobV2....9
2.1.4.5 Creating a UDF....11
2.1.4.6 Creating a UDTF....14
2.1.4.7 Creating a UDAF....16
2.1.4.8 Querying Function Details....19
2.1.4.9 Listing Functions....20
2.1.4.10 Deleting Functions....21
2.1.4.11 Registering a Data-Juicer Operator....22
2.1.4.12 Common Parameters....25
2.1.4.12.1 signature....25
2.1.4.12.2 runtime_env....27
2.1.4.13 Exception Classes....29
2.1.5 Aura DataFrame SDK (Python)....30
2.1.5.1 Aura DataFrame SDK Overview....30
2.1.5.2 Obtaining and Installing the Aura DataFrame SDK....33
2.1.5.3 Getting Started with the Aura DataFrame SDK....35
2.1.5.4 Submitting a Job Using the Aura DataFrame SDK....38
2.1.5.4.1 Establishing a Connection with the AuraJobV2 Endpoint....38
2.1.5.4.2 Configuring the UDF Deployment Workspace....40
2.1.5.4.3 Creating a Table and Writing Data to the Table....41
2.1.5.4.4 Establishing a Connection and Querying Data from a Specified Table....43
2.1.5.4.5 Data Processing and Analysis....44
2.1.5.4.6 Using Scalar UDFs to Process Data....46

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.4.7 Using Class UDFs to Process Data 48
2.1.5.4.8 Using UDAFs for Aggregation Statistics 51
2.1.5.4.9 Directly Using a Registered Scalar UDF 53
2.1.5.4.10 Directly Using a Registered UDAF 55
2.1.5.4.11 Speech Recognition Case Study 56
2.1.5.4.12 Video Frame Description and Summarization Example 60
2.1.5.5 Aura DataFrame SDK Summary 67
2.1.5.6 Session APIs 71
2.1.5.7 Dataset APIs 73
2.1.5.8 Table APIs 94
2.1.5.9 I/O APIs 101
2.1.5.10 UDF APIs 106
2.1.5.11 AI APIs 125
2.1.5.12 UDFs 128
2.1.5.12.1 UDF Overview 128
2.1.5.12.2 Development Notes 133
2.1.5.12.3 Data Type Mapping 134
2.1.5.12.4 Environment Variable Configuration 134
2.1.5.12.5 Scalar UDF 138
2.1.5.12.6 Class UDF 140
2.1.5.12.7 Vectorized UDF 143
2.1.5.12.8 UDTF 145
2.1.5.12.9 UDAF 147
2.1.5.12.10 WITH ARGUMENTS Syntax 151
2.1.5.12.11 Adaptive Parallelism 152
2.1.5.12.12 UDF Error Handling and Retry 153
2.1.5.12.13 UDF Actor Placement Policy 161
2.1.5.12.14 Python UDF Performance Tuning 165
2.1.5.13 Data Types 167
2.1.5.13.1 Images 167
2.1.5.13.2 Videos 168
2.1.5.13.3 Audio 169
2.1.5.13.4 Vectors 171
2.1.6 Spark SDK (Python) 171
2.1.6.1 Obtaining and Initializing the Spark SDK 172
2.1.6.2 Submitting a Spark SQL Job 174
2.1.6.3 Submitting a Spark Job 176
2.1.6.4 Connecting to the Database and Querying Data Using Spark 179
2.2 Java SDKs 181
2.2.1 Configuring the Java Environment 181
2.2.2 Spark SDK (Java) 182
2.2.2.1 Obtaining and Initializing the Spark SDK 182

 Copyright © Huawei Technologies Co., Ltd.

2.2.2.2 Submitting a Spark SQL Job..... 185
2.2.2.3 Submitting a Spark Job..... 187
2.2.2.4 Querying the Metadata of the Spark Databases and Tables..... 191

 Copyright © Huawei Technologies Co., Ltd.

1 AI DataLake SDK

Introduction to AI DataLake SDK

AI DataLake provides various types of software development kits (SDKs) to simplify the integration development with the AI DataLake service. You can use SDKs to perform operations such as job submission, data querying, and custom function development without needing to manage the underlying details of REST API calls.

AI DataLake SDKs are categorized by language into **Python SDK** and **Java SDK**. These SDKs cover various job scenarios for computing engines such as Aura, Ray, and Spark.

This document classifies SDKs by language. You can directly locate the appropriate SDK based on your business's technical stack.

For details about the privacy data processing policies of SDKs, see **Huawei Cloud SDK Privacy Statement**.

SDK Languages

Language	SDK	Applicable Engines	Applicable Scenarios	Help Documents
Python	UDF SDK	Aura (AuraJobV2)	UDF/UDAF/UDTF registration and management; multi-engine reuse	**Catalog SDK**
Python	Aura DataFrame SDK	Aura (AuraJobV2)	Python/Notebook interactive data processing; multi-modal data development	**Aura DataFrame SDK (Python)**
Python	Spark SDK (Python)	Spark (Spark SQL, Spark Job)	Spark SQL queries; Spark batch job submission	**Spark SDK (Python)**

 Copyright © Huawei Technologies Co., Ltd.

Language	SDK	Applicable Engines	Applicable Scenarios	Help Documents
Java	Spark SDK (Java)	Spark (Spark SQL, Spark Job)	Spark SQL queries; Spark batch job submission; database and table metadata queries	**Spark SDK (Java)**

SDK Architecture

The AI DataLake SDK system consists of **engine-specific SDKs**, and **cross-engine shared SDKs** (three layers).

- **Engine-specific SDKs**: Self-developed SDKs for each engine (Aura DataFrame SDK, Ray SDK, and Spark SDK).
- **Cross-engine shared SDKs**: Used to register UDFs with LakeFormation and reuse them across multiple engines (UDF SDK).

AI DataLake SDK for UDFs

The UDF SDK provided by AI DataLake is designed for cross-engine reuse of UDFs. You can write a function once using the UDF SDK and register it with the LakeFormation metadata service. This allows the function to be recognized and called by multiple engines, avoiding redundant development across different engines.

Type	Description
Applicable engines	Currently, Aura is supported.
Applicable scenarios	One-time development and multi-engine reuse of UDFs/UDAFs/UDTFs
Implementation mechanism	Function code and metadata are uniformly stored in LakeFormation. Each engine discovers and calls the code and metadata through the metadata service.
Core capabilities	Function registration (**create_udf**/**create_udtf**/**create_udaf**), query, deletion, and Data-Juicer operator adaptation

Engine-specific SDKs

Engine-specific SDKs for Aura, Ray, and Spark are encapsulated based on general-purpose SDKs or REST APIs, making them more user-friendly and simplifying the development process.

 Copyright © Huawei Technologies Co., Ltd.

SDK	Language	Applicable Endpoint Type	Applicable Scenarios	Core Features	SDK Reference
Aura DataFrame SDK	Python	AuraJobV2	Python / Notebook interactive data processing; multi-modal data development	Pandas-like DataFrame APIs, with results directly returned as Pandas DataFrames. Supports multi-modal data types (images, videos, audio, and vectors). Supports a full range of Python UDF capabilities (scalar UDF, class UDF, vectorized UDF, UDTF, and UDAF). Supports operator-level resource specification configuration, affinity scheduling policies, and performance monitoring.	**Aura DataFrame SDK Overview**
Spark SDK	Java, Python	Spark SQL, Spark Job	Spark SQL queries and Spark batch job submission	The Java version provides three types of clients: Spark SQL, Spark Job, and metadata query. The Python version provides a unified client and Builder for chain job construction, supporting job types such as JAR and SQL scripting.	**Spark SDK (Python)**

More Connections

In addition to SDKs, AI DataLake provides the following connection methods. The difference between SDKs and these connection methods is that SDKs provide encapsulated programming APIs, which are suitable for application integration development. The following methods are used for specific tools or protocol scenarios:

Connection Method	Applicable Engines	Applicable Scenarios
Spark Connect	Spark	Notebook interactive development and PySpark script integration
REST API	All engines	Automated job submission and application integration
DataArts Studio	Aura	Visualized job orchestration and production environment scheduling

Copyright © Huawei Technologies Co., Ltd.

2 Engine-specific SDKs

2.1 Python SDKs

2.1.1 Python SDK Overview

AI DataLake provides multiple types of SDKs for Python developers to simplify the integration development with AI DataLake. You can use Python SDKs to perform operations such as job submission, data querying, and custom function development without needing to manage the underlying details of REST API calls.

Table 2-1 AI DataLake Python SDKs

SDK Type	Applicable Engines	Applicable Scenarios
UDF SDK	Aura (AuraJobV2)	One-time development and multi-engine reuse of UDFs/ UDAFs/UDTFs
Aura DataFrame SDK	Aura (AuraJobV2)	Python/Notebook interactive data processing; multi-modal data development
Spark SDK-Python	Spark (Spark SQL, Spark Job)	Spark SQL queries and Spark batch job submission

2.1.2 Python Environment Configuration

Scenarios

Before installing and using Python SDKs, make sure you have completed the basic configuration of your development environment.

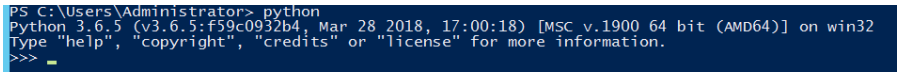
It is recommended to use Python version 2.7.10 or later, or Python 3.4.0 or later. You must also configure the Visual C++ compilation environment, such as Visual C ++ build tools or Visual Studio.

 Copyright © Huawei Technologies Co., Ltd.

Procedure

1. Download Python from **Python's official website** and install it.
2. Check whether the configurations succeed. Specifically, at the **Start** menu, enter **cmd** and press **Enter** to bring up the command prompt window. Enter **python** and press **Enter**. If the version information is displayed, as shown in **Figure 2-1**, Python is successfully installed and configured.

   Figure 2-1 Checking the configuration

   NOTE

   If you encounter an error similar to "error: Microsoft Visual C++ xx.x is required. Get it with Build Tools for Visual Studio" when installing the Python application package, the C++ compiler may not been installed. In this case, install the Visual Studio compiler of the required version. For some OSs, restart the system after Visual Studio is installed.

2.1.3 AK/SK Obtaining

The Python SDKs of AI DataLake use the Access Key ID/Secret Access Key (AK/SK) authentication to access services. You must obtain your access keys before using the SDKs.

Notes and Constraints

- Hard-coded or plaintext AK/SK is risky. You are advised to store them in the configuration files or environment variables.
- When using a temporary AK/SK pair, you must also configure a security token. Since temporary credentials have a validity period, they are recommended only for testing purposes.

Procedure

1. Log in to the **AI DataLake console**.
2. Hover over the username in the upper right corner and choose **My Credentials** from the drop-down list.
3. Click the **Access Keys** tab.
4. Click **Add Access Key**.
5. Enter an access key description (optional) and click **OK**.
6. Enter the verification code or password for identity authentication.
7. Click **OK** to generate and download the access key file (**.csv**).
8. Open the **.csv** file to obtain the access key ID (AK) and secret access key (SK) of the user.

2.1.4 Catalog SDK

 Copyright © Huawei Technologies Co., Ltd.

2.1.4.1 Catalog SDK Overview

This section describes the Python SDKs provided by the Catalog module of AI DataLake, including how to obtain and use the SDK.

Introduction to Catalog SDKs

SDK is short for Software Development Kit. It is a collection of software development tools provided for developers, including a series of APIs, tools, sample code, and documents. It helps developers quickly and efficiently develop various applications.

The AI DataLake Catalog Python SDK encapsulates REST APIs of LakeFormation and Object Storage Service (OBS) to simplify the registration, management, and development of user-defined functions (UDFs). It is designed for scenarios where UDF management and usage are separated. The Python UDFs registered using this SDK can be used by the Aura engine in AI DataLake. The SDK identifies the caller's identity using access keys, and provides capabilities such as automatic signing, type inference, UDF packaging and uploading, and metadata registration. This allows you to submit and manage scalar UDFs, UDTFs, UDAFs, and Data-Juicer operators through a unified client.

The core entry of the SDK is **AIDatalake_Catalog_Client**, which encapsulates the following capabilities:

- LakeFormation REST API calls (Function CRUD operations)
- OBS upload/deletion of UDF packages
- Type inference based on annotations of callable objects
- Packaging UDFs into ZIP files and uploading them

Features

Table 2-2 SDK capabilities

Capability	Method	Description
Registering scalar UDFs	create_udf	Registers scalar functions to LakeFormation, processing input columns row by row.
Registering UDTFs	create_udtf	Registers table functions to LakeFormation, allowing multiple rows and columns to be returned at once.
Registering UDAFs	create_udaf	Registers aggregate functions to LakeFormation, aggregating data across multiple rows.
Registering Data-Juicer operators	create_datajuicer_operator	Automatically adapts and registers the Filter/Mapper operators of Data-Juicer as scalar UDFs.

 Copyright © Huawei Technologies Co., Ltd.

Capability	Method	Description
Querying functions	get_function	Retrieves function metadata from LakeFormation by name.
Listing functions	list_function	Lists all functions in a specified database.
Deleting functions	delete_function	Deletes function records from LakeFormation and clears their OBS packages.

2.1.4.2 Obtaining and Installing the Catalog SDK

The SDK is released in the PyPI repository as a Python package. You can run the **pip** command to install the SDK. During the installation, the third-party libraries required by the SDK are automatically downloaded. This section describes how to obtain and install the SDK.

Prerequisites

Python 3.10 or 3.11 has been installed. For details about the installation, see **Python Environment Configuration**.

Procedure

Step 1 Obtain the SDK installation package from technical support.

Step 2 Install the SDK.

1. Download and decompress the SDK installation package.
2. Open the CLI and go to the directory where the package is decompressed.
3. Run the following command to install the SDK. During the installation, the third-party dependencies are automatically downloaded.

```
pip install huawei_aidatalake_catalog-0.1.3-py3-none-any.whl
```

NOTE

The SDK dependencies include **esdk-obs-python** (OBS), LakeFormation REST client, **cloudpickle**, **pyarrow**, and **pandas**. These dependencies are automatically obtained during the SDK installation.

----End

2.1.4.3 Initializing the Client

The client **AIDatalake_Catalog_Client** is the core entry of the SDK. During initialization, you need to pass in authentication credentials (AK/SK), endpoints for LakeFormation and OBS, and other relevant information. Once initialized, you can use this client to create, query, and delete functions.

 Copyright © Huawei Technologies Co., Ltd.

Prerequisites

- A LakeFormation instance has been created, with a catalog and a database available in the instance.
- An OBS parallel file system has been created.

Sample Code for AK/SK Authentication

- Sample code

```
from aidatalake_catalog.client import AIDatalake_Catalog_Client

client = AIDatalake_Catalog_Client(
    ak="your-ak",
    sk="your-sk",
    security_token=None,                    # Used for temporary credentials
    obs_endpoint="obs.cn-north-4.myhuaweicloud.com",
    lakeformation_endpoint="https://lakeformation.cn-north-4.myhuaweicloud.com",
    lakeformation_instanceid="your-instance-id",
    catalog_name="your-catalog-name",
    default_database="your_default_db",
    obs_path="your-bucket/udf/",            # bucket_name/dir/subdir
    project_id="your-project-id",
    ignore_ssl_verification=False,
)
```

- Parameter description

Parameter	Mandatory (Yes/No)	Type	Description
ak	Yes	str	Access keys (AK/SK). For details about how to obtain the AK/SK, see **AK/SK Obtaining**.
sk	Yes	str	
security_token	No	str | None	Security token of the temporary credentials. If the long-term AK/SK credential is used, set this parameter to **None**.
obs_endpoint	Yes	str	OBS endpoint, for example, **obs.cn-north-4.myhuaweicloud.com**.
lakeformation_endpoint	Yes	str	URL of the LakeFormation API endpoint.
lakeformation_instanceid	Yes	str	LakeFormation instance ID (used for IAM authentication).
catalog_name	Yes	str	LakeFormation catalog name (three-layer structure: instance/catalog/database).
default_database	Yes	str	Default database name for function operations.
obs_path	Yes	str	OBS path, in the format of **bucket_name/dir/subdir**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Description
project_id	Yes	str	Project ID (used for LakeFormation API authentication).
ignore_ssl_verification	No	bool	Whether to disable SSL certificate verification. The default value is **False**. This parameter can be set to **True** in the self-signed certificate scenario.

NOTE

- Hard coding AKs and SKs or storing them in code in plaintext poses significant security risks. You are advised to store them in encrypted form in configuration files or environment variables and decrypt them when needed to ensure security.
- This example stores AK and SK in environment variables. Before running this example, set the environment variables **xxx_SDK_AK** and **xxx_SDK_SK** in the local environment.

2.1.4.4 Running the Example with AuraJobV2

This example demonstrates how to register a UDF with LakeFormation using the SDK, and then use the AuraJobV2 engine to read data columns from a specified database table and perform calculations using the UDF.

Prerequisites

- An OBS parallel file system has been created to store the UDF package uploaded by the SDK. For details, see **Creating a Parallel File System**.
- A LakeFormation instance has been created, with a catalog and a database available in the instance. For details, see **Creating a DataArts Lake Formation Instance and Planning Metadata**.
- A data table has been registered in LakeFormation. (In this example, the table contains at least the **id** and **value** columns.)
- An AuraJobV2 endpoint has been created, and information such as the endpoint name and workspace ID has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- You have obtained the AK/SK information and have the permission to access the OBS parallel file system and LakeFormation.
- The Aura DataFrame SDK has been obtained and installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.

Example

```
# 1. Import dependencies.
import os
```

 Copyright © Huawei Technologies Co., Ltd.

```
from aidatalake_catalog import Parameter, Signature
from aidatalake_catalog.client import AIDatalake_Catalog_Client
from aura_frame.multimodal import ai_lake

# 2. Define a UDF with type annotations.
class Multiplier:
    """Class-defined UDF: Multiplies by a configurable factor (default: 2)."""
    def __init__(self, factor: int = 2):
        self._factor = factor
    def __call__(self, x: int) -> int:
        return x * self._factor

# 3. Define the function signature. (Optional. If not passed, the SDK automatically infers it based on the
type annotations.)
sig = Signature(
    parameters=[Parameter(name="x", annotation=int)],
    return_annotation=int,
)
# 4. Initialize the Catalog client. (You are advised to read the AK/SK from environment variables to avoid
hardcoding.)
client = AIDatalake_Catalog_Client(
    ak=os.getenv("XXX_SDK_AK"),
    sk=os.getenv("XXX_SDK_SK"),
    security_token=None,                    # Used for temporary credentials
    obs_endpoint="obs.cn-north-4.myhuaweicloud.com",
    lakeformation_endpoint="https://lakeformation.cn-north-4.myhuaweicloud.com",
    lakeformation_instanceid="your-instance-id",
    catalog_name="your-catalog-name",
    default_database="your_default_db",
    obs_path="your-bucket/udf/",            # bucket_name/dir/subdir
    project_id="your-project-id",
    ignore_ssl_verification=False,
)

# 5. Register the UDF.
func = client.create_udf(
    Multiplier,
    name="class_multiplier_def",
    signature=sig,
    description="Class-defined UDF: Multiplies by a factor (default: 2)."
)

# 6. Use the Aura engine to execute the function.
"""
|  id  | value|
|------|------|
|  1   |  10  |
|  2   |  20  |
|  3   |  30  |
"""
con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

ds = con.load_dataset("table_1")  # table_1 must be registered with LakeFormation in advance.
ds = ds.map(fn=f'{database}.class_multiplier_def', on=['value'], as_col='mul_result')
df = ds.execute().sort_values('id').reset_index(drop=True)

print(f"  {'ID':<5} {'Original value':<10} {'mul_result':<15}")
print("-" * 35)
for _, r in df.iterrows():
    print(f"{int(r['id']):<5} {int(r['value']):<10} {int(r['mul_result']):<15}")
```

 Copyright © Huawei Technologies Co., Ltd.

Execution result:

```
 ID  Original value  mul_result
-----------------------------------
 1   10        20
 2   20        40
 3   30        60
```

2.1.4.5 Creating a UDF

This section describes how to register a UDF (which processes input columns row by row and returns a single value) with LakeFormation. The registration process is as follows: Type inference -> Packaging -> Uploading to OBS -> Writing metadata to LakeFormation. If the registration in LakeFormation fails, the uploaded OBS package will be automatically rolled back.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- The UDF callable object (function, function class, or class instance) has been prepared and annotated with type annotations.

Function Signature

```
client.create_udf(
    udf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
udf	Yes	Any (location parameter)	-	UDF callable object (function or callable class).

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
database	No	str | None	default_database	Name of the target database. If this parameter is set to **None**, the default database on the client is used.
name	Yes	str	-	Function name. The value must start with a letter and contain 3 to 128 characters, including only letters, digits, and underscores (_).
signature	No	Signature | None	None	Explicit function signature. If this parameter is set to **None**, it is automatically inferred from UDF annotations. For more information about the signature, see **signature**.
imports	No	str | Iterable[str] | None	None	List of file/directory paths or URLs to be included in the UDF package.
runtime_env	No	str | Mapping | None	None	Runtime environment configuration (such as **python_version** and PyPI dependencies). The value can be a JSON string or dictionary. For more information about **runtime_env**, see **runtime_env**.
description	No	str | None	None	Description of the function.
pretty_name	No	str | None	None	Display name. If the parameter is set to **None**, the value of **name** is used.
is_strict	No	bool	True	Whether the function is strict (whether NULL is returned when the input is NULL).
volatility	No	str	VOLATILE	Function volatility. The options are **IMMUTABLE**, **STABLE**, and **VOLATILE**.
is_batch	No	bool	False	Whether to enable the batch processing mode.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
batch_size	No	int | None	None	Size of each batch. This parameter is recommended when **is_batch** is set to **True**. The value must be a positive integer.
cpu_arch	No	str | None	None	CPU architecture. The value can be **arm** or **x86**.
cpus	No	float | None	None	Number of CPU cores. The minimum value is **0.33**.
memory	No	int | None	None	Memory size, in MB. The minimum value is **1**.
device_type	No	str | None	None	Device type, for example, NPU or GPU.
devices	No	float | None	None	Number of devices. The value must be a positive number.

Return Value

Returns a **Function** object containing the **version_id** assigned by LakeFormation, **resource_uris** (OBS package addresses), and parameters (metadata such as the signature and runtime environment).

Sample Code

NOTE

- If no signature is provided, the SDK infers the signature from the UDF's type annotations using **infer_signature**. For class-based UDFs, the **__call__** method is used.
- For class-based UDFs, the constructor definition (ConstructorDef) is extracted and the class name is recorded to help the Aura engine locate the class.
- For class instance UDFs, the instance is serialized via **cloudpickle**, and the class name is derived from the instance type.

```
# 1. Define a UDF with type annotations.
def to_celsius(fahrenheit: float) -> float:
    return (fahrenheit - 32.0) * 5.0 / 9.0

# 2. Register with LakeFormation.
func = client.create_udf(
    udf=to_celsius,
    name="to_celsius",
    description="Convert Fahrenheit to Celsius",
    runtime_env={"python_version": "3.11", "pypi": []},
)

print(f"Registered: {func.function_name}, version={func.version_id}")
```

 Copyright © Huawei Technologies Co., Ltd.

2.1.4.6 Creating a UDTF

This section describes how to register a UDTF with LakeFormation. A UDTF can receive input and return results with multiple rows and columns.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- The UDF callable object (function, function class, or class instance) has been prepared and annotated with type annotations.
- OBS and LakeFormation have been enabled, and the corresponding endpoints, instance IDs, and project IDs have been obtained.

Function Signature

```
client.create_udtf(
    udtf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
udtf	Yes	Any (location parameter)	-	UDTF callable object (function or callable class).
database	No	str | None	default_database	Name of the target database. If this parameter is set to **None**, the default database on the client is used.
name	Yes	str	-	Function name. The value must start with a letter and contain 3 to 128 characters, including only letters, digits, and underscores (_).

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
signature	No	Signature | None	None	Explicit function signature. If this parameter is set to **None**, it is automatically inferred from UDF annotations. For more information about the signature, see **signature**.
imports	No	str | Iterable[str] | None	None	List of file/directory paths or URLs to be included in the UDF package.
runtime_env	No	str | Mapping | None	None	Runtime environment configuration (such as **python_version** and PyPI dependencies). The value can be a JSON string or dictionary. For more information about **runtime_env**, see **runtime_env**.
description	No	str | None	None	Description of the function.
pretty_name	No	str | None	None	Display name. If the parameter is set to **None**, the value of **name** is used.
is_strict	No	bool	True	Whether the function is strict (whether NULL is returned when the input is NULL).
volatility	No	str	VOLATILE	Function volatility. The options are **IMMUTABLE**, **STABLE**, and **VOLATILE**.
is_batch	No	bool	False	Whether to enable the batch processing mode.
batch_size	No	int | None	None	Size of each batch. This parameter is recommended when **is_batch** is set to **True**. The value must be a positive integer.
cpu_arch	No	str | None	None	CPU architecture. The value can be **arm** or **x86**.
cpus	No	float | None	None	Number of CPU cores. The minimum value is **0.33**.
memory	No	int | None	None	Memory size, in MB. The minimum value is **1**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
device_type	No	str | None	None	Device type, for example, NPU or GPU.
devices	No	float | None	None	Number of devices. The value must be a positive number.

Return Value

Returns a **Function** object.

Sample Code

```
import pyarrow as pa

def split_range(start: int, end: int) -> pa.Table:
    return pa.table({"v": list(range(start, end))})

func = client.create_udtf(
    udf=split_range,
    name="split_range",
    description="Generate a table of integers in the [start, end) range."
)
print(f"UDTF registered: {func.function_name}")
```

2.1.4.7 Creating a UDAF

This section describes how to register a UDAF with LakeFormation. A UDAF can perform aggregation calculations on multiple rows of data. Except that **udf_type** is fixed as **udaf**, its parameters are the same as those of **create_udf**.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- The UDF callable object (function, function class, or class instance) has been prepared and annotated with type annotations.

Function Signature

```
client.create_udaf(
    udaf, /, *,
    database=None,
    name,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
    batch_size=None,
    cpu_arch=None,
    cpus=None,
```

 Copyright © Huawei Technologies Co., Ltd.

```
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
udaf	Yes	Any (location parameter)	-	UDAF callable object (function or callable class).
database	No	str | None	default_database	Name of the target database. If this parameter is set to **None**, the default database on the client is used.
name	Yes	str	-	Function name. The value must start with a letter and contain 3 to 128 characters, including only letters, digits, and underscores (_).
signature	No	Signature | None	None	Explicit function signature. If this parameter is set to **None**, it is automatically inferred from UDF annotations. For more information about the signature, see **signature**.
imports	No	str | Iterable[str] | None	None	List of file/directory paths or URLs to be included in the UDF package.
runtime_env	No	str | Mapping | None	None	Runtime environment configuration (such as **python_version** and PyPI dependencies). The value can be a JSON string or dictionary. For more information about **runtime_env**, see **runtime_env**.
description	No	str | None	None	Description of the function.
pretty_name	No	str | None	None	Display name. If the parameter is set to **None**, the value of **name** is used.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
is_strict	No	bool	True	Whether the function is strict (whether NULL is returned when the input is NULL).
volatility	No	str	VOLATILE	Function volatility. The options are **IMMUTABLE**, **STABLE**, and **VOLATILE**.
is_batch	No	bool	False	Whether to enable the batch processing mode.
batch_size	No	int | None	None	Size of each batch. This parameter is recommended when **is_batch** is set to **True**. The value must be a positive integer.
cpu_arch	No	str | None	None	CPU architecture. The value can be **arm** or **x86**.
cpus	No	float | None	None	Number of CPU cores. The minimum value is **0.33**.
memory	No	int | None	None	Memory size, in MB. The minimum value is **1**.
device_type	No	str | None	None	Device type, for example, NPU or GPU.
devices	No	float | None	None	Number of devices. The value must be a positive number.

Return Value

Returns a **Function** object.

Sample Code

```
class WeightedSum:
    """UDAF: weighted sum (sum(value * weight)).

    accumulate accumulates the intermediate state (sum_vw) row by row; merge merges partitions;
    finish returns the final aggregation result.
    """

    def __init__(self):
        self._agg_state = 0.0

    @property
    def aggregate_state(self) -> float:
        return self._agg_state
```

 Copyright © Huawei Technologies Co., Ltd.

```
    def accumulate(self, value: int, weight: int) -> None:
        """Accumulate row by row: sum of value * weight"""
        if value is not None and weight is not None:
            self._agg_state += value * weight
    def merge(self, other_state: float) -> None:
        """Merge the intermediate states of other partitions"""
        self._agg_state += other_state
    def finish(self) -> float:
        """Return the final aggregation result"""
        return self._agg_state

func = client.create_agg_function(
    WeightedSum,
    database=database,
    name="weighted_sum_udaf",
    signature=sig,
    comment="UDAF: Weighted sum (sum(value * weight))",
)
```

2.1.4.8 Querying Function Details

This section describes how to retrieve the metadata of a single function from LakeFormation by its name.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- The target function is available in the corresponding database.

Function Signature

```
client.get_function(*, database=None, name) -> Function
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
database	No	str | None	default_database	Database name. If this parameter is set to **None**, the default database is used.
name	Yes	str	-	Name of a function.

Return Value

Returns a **Function** object containing fields such as **function_name**, **resource_uris**, **version_id**, **parameters** and **class_name**.

 Copyright © Huawei Technologies Co., Ltd.

Exception Description

Exception	Trigger Condition
FunctionNotFoundError	The specified function does not exist.
LakeFormationAPIError	Failed to call the LakeFormation API (due to network issues, authentication failures, 5*xx* errors, or other exceptions).

Sample Code

```
func = client.get_function(name="to_celsius")
print(func.function_name, func.version_id)
print(func.parameters.signature.to_dict())
```

2.1.4.9 Listing Functions

This section describes how to list all functions in a specified database.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- A database is available.

Function Signature

```
client.list_function(*, database=None) -> list[Function]
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
database	No	str | None	default_database	Database name. If this parameter is set to **None**, the default database is used.

Return Value

Returns a **list[Function]** containing all function objects in the database.

Sample Code

```
for func in client.list_function():
    print(func.function_name, func.parameters.udf_type)
```

 Copyright © Huawei Technologies Co., Ltd.

2.1.4.10 Deleting Functions

The section describes how to delete function records from LakeFormation and clear their UDF packages on OBS. The deletion process is as follows:

1. Retrieve the function metadata (used to locate **resource_uris**).
2. Delete the corresponding OBS package (if the function has been partially deleted, the SDK clears the package by prefix to prevent leakage).
3. Delete the function records from LakeFormation.
4. If the OBS package fails to be cleared but the function records are successfully deleted from LakeFormation, an exception is thrown to indicate the presence of orphan packages.

Prerequisites

- The SDK has been installed. For details, see **Obtaining and Installing the Catalog SDK**.
- The target database and function name are known. (If the function metadata is unavailable, the SDK clears the OBS package by the **{*Database*}/{*Name*}/*Prefix***.)

Function Signature

```
client.delete_function(*, database=None, name) -> None
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
database	No	str | None	default_database	Database name. If this parameter is set to **None**, the default database is used.
name	Yes	str	-	Name of a function.

Return Value

No value is returned (**None**).

Exception Description

Exception	Trigger Condition
FunctionNotFoundError	The function does not exist. This exception is safely ignored in the deletion process. Then the SDK clears OBS using the prefix.

 Copyright © Huawei Technologies Co., Ltd.

Exception	Trigger Condition
LakeFormationAPIError	The LakeFormation API fails to be called (due to network issues, authentication failures, or 5xx errors). The exception needs to be reported to notify the caller that the cleanup is not complete.
OBSOperationError	The OBS package fails to be cleared but the function records are successfully deleted from LakeFormation. This indicates the presence of orphan OBS packages.

Sample Code

```
client.delete_function(name="to_celsius")
print("Function deleted and OBS package cleaned")
```

NOTE

- When the function metadata is unavailable, the SDK clears OBS objects based on the database, function name, and prefix to prevent orphan packages.
- If the OBS package fails to be deleted but the function records have been deleted from LakeFormation, the SDK throws an OBSOperationError. The caller needs to check for orphan packages based on this error.

2.1.4.11 Registering a Data-Juicer Operator

This section describes how to register a Data-Juicer operator (an instance of a Filter or Mapper subclass) as a scalar UDF. Internally, the SDK uses **dj_operator_to_aura_udf** to adapt the operator into a callable UDF, and then delegates the registration to **create_udf**.

Since **cloudpickle** serializes Data-Juicer operator classes by reference, the SDK automatically adds **py-data-juicer** to **runtime_env.pypi** to ensure that the package is installed in the Aura runtime. User-defined **runtime_env** takes precedence over automatically injected items.

Prerequisites

The **data_juicer** package has been installed (used to construct operator instances locally).

Function Signature

```
client.create_datajuicer_operator(
    operator, /, *,
    name,
    input_columns,
    output_columns=None,
    database=None,
    signature=None,
    imports=None,
    runtime_env=None,
    description=None,
    pretty_name=None,
    is_strict=True,
    volatility="VOLATILE",
    is_batch=False,
```

 Copyright © Huawei Technologies Co., Ltd.

```
    batch_size=None,
    cpu_arch=None,
    cpus=None,
    memory=None,
    device_type=None,
    devices=None,
) -> Function
```

Parameter Description

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
operator	Yes	Any (location parameter)	-	Initialized Data-Juicer operator instance (subclass of Filter or Mapper).
name	Yes	str	-	Name of the function registered with LakeFormation.
input_columns	Yes	list[str]	-	The input column names expected by the operator. The sequence aligns with the location parameter passed by the engine.
output_columns	No	list[str] | None	None	The output column names produced by the operator. This parameter is required for Mapper operations and can be ignored for Filter operations.
database	No	str | None	default_database	Name of the target database.
signature	No	Signature | None	None	Explicit function signature. For more information about the signature, see **signature**.
imports	No	str | Iterable[str] | None	None	Additional files, directories, or URLs to be included in the UDF package.
runtime_env	No	str | Mapping | None	None	Runtime environment configuration, which will be merged with **{"pypi": ["py-data-juicer"]}**. For more information about **runtime_env**, see **runtime_env**.
description	No	str | None	None	Description of the function.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Type	Default Value	Description
pretty_name	No	str | None	None	Display name.
is_strict	No	bool	True	Whether the function is strict.
volatility	No	str	VOLATILE	Function volatility.
is_batch	No	bool	False	Whether to enable batch processing.
batch_size	No	int | None	None	Batch size.
cpu_arch	No	str | None	None	CPU architecture. The value can be **arm** or **x86**.
cpus	No	float | None	None	Number of CPU cores. The minimum value is **0.33**.
memory	No	int | None	None	Memory size, in MB. The minimum value is **1**.
device_type	No	str | None	None	Device type.
devices	No	float | None	None	Number of devices.

Return Value

Returns a **Function** object (actually returned by **create_udf**).

Exception Description

Exception	Trigger Condition
ValueError	The operator is a Mapper, but **output_columns** is not provided.
ImportError	The **data_juicer** package is not installed.

Sample Code

```
from data_juicer.ops.mapper import LowercaseMapper
```

 Copyright © Huawei Technologies Co., Ltd.

```
op = LowercaseMapper(text_column="content")

func = client.create_datajuicer_operator(
    operator=op,
    name="dj_lowercase",
    input_columns=["content"],
    output_columns=["content"],          # Mandatory for Mapper
    description="Data-Juicer lowercase mapping operator",
)
```

NOTE

- The Mapper and Filter operator types are supported.
- The Filter operators do not require **output_columns**. If provided, they are ignored.
- **runtime_env** merging rule: User-defined configurations take precedence. **py-data-juicer** is automatically added as a supplementary dependency. Duplicate package names are handled based on the deduplication policy.

2.1.4.12 Common Parameters

2.1.4.12.1 signature

The **signature** describes the function signature of a UDF, including the type information of the input parameters and return values. It consists of a parameter list and a **return_annotation**. During function registration, the signature is written into the LakeFormation metadata for the engine to match parameters and types during invocation.

signature parameters

A signature object is composed of a parameter list and a **return_annotation**.

Table 2-3 Property description

Property	Type	Description
name	str	Parameter name.
kind	Parameter Kind	Parameter type. The value can be **POSITIONAL_ONLY**, **POSITIONAL_OR_KEYWORD**, **KEYWORD_ONLY**, **VAR_POSITIONAL**, or **VAR_KEYWORD**.
default	Any	Default value. If there is no default value, the value is **EMPTY**.
annotation	Any	Type annotation. If there is no annotation, the value is **EMPTY**.

 Copyright © Huawei Technologies Co., Ltd.

When to Use

Scenario	Recommendation
The UDF has complete and accurate type annotations.	Pass **signature=None** and let the SDK automatically infer the signature.
The UDF annotation does not match the expected signature or needs to be overwritten.	Explicitly pass a signature object.
Batch mode requires the element types for the signature.	Explicitly pass a signature with element types so the SDK can map them to LakeFormation column types.

Auto Inference Rules (signature=None)

When the signature is not explicitly passed (i.e., **None**), the SDK automatically infers the function signature based on the UDF's code structure and type annotations. You do not need to manually construct a signature object. If the inference fails, a **SignatureInferenceError** is thrown.

Syntax example:

Annotate the parameters and return value of the UDF to trigger automatic inference.

```
# UDF with type annotations (recommended)
def to_celsius(fahrenheit: float) -> float:
    return (fahrenheit - 32) * 5 / 9

# When signature=None, the SDK automatically infers the equivalent signature:
#   signature = [Parameter(name="fahrenheit", kind=POSITIONAL_OR_KEYWORD, default=EMPTY,
annotation=float)]
#   return_annotation = float
client.create_udf(to_celsius, name="to_celsius")
```

Inference rules:

UDF Form	Inference Behavior
Regular functions	Use **inspect.signature** and **get_type_hints** to read parameters and return annotations.
Class-defined UDFs	Check the **__call__** method in the class and read its annotations. If the class has no custom **__call__** method, a **SignatureInferenceError** is raised.
Class-instance UDFs	Check the **__call__** method of the class to which the instance belongs. The instance has been serialized using **cloudpickle**, and **class_name** is derived from **type(udf).__name__**.
Binding method	Automatically unpacked and inferred as a regular function.

 Copyright © Huawei Technologies Co., Ltd.

NOTE

- All parameters must have type annotations. **FunctionSignature.from_signature** will raise a ValueError for parameters without annotations.
- Raw container types (such as **list** and **dict** without element types) are not invalid. You must use annotated types like **list[T]** and **dict[K, V]**.
- If the signature is **None** and inference fails (for example, the function has no type annotations or the class lacks __**call**__ method), the SDK will raise a **SignatureInferenceError**.

Signature Requirements in Batch Mode (is_batch=True)

In batch mode (**is_batch=True**), the UDF processes a batch of data at once rather than row by row. Therefore, the parameter annotations of the signature must use the batch types (PyArrow or Pandas array/table structures), and all parameters must have the same batch types to match the engine's batch parameter passing mechanism.

Syntax example:

Define a batch UDF using batch type annotations and enable **is_batch** during registration.

```
import pyarrow as pa

def double_batch(values: pa.Array) -> pa.Array:
    return pa.array([v * 2 for v in values])

client.create_udf(double_batch, name="double_batch", is_batch=True, batch_size=1024)
```

Batch type mapping

Batch Type	Detected input_type
pyarrow.Array/ pyarrow.Table	pyarrow
pandas.Series/ pandas.DataFrame	pandas

NOTE

- If the signature is automatically inferred from the annotations and the parameters are non-batch element types, the SDK throws a ValueError.
- If an explicit signature is passed using element types, the SDK will infer the **input_type** from the UDF annotations.
- It is recommended to use **list[T]** to carry element type information, which facilitates LakeFormation to store the original column data types. A raw **pyarrow.Array** or **pandas.Series** without element types will cause errors during the **from_signature** phase.

2.1.4.12.2 runtime_env

The **runtime_env** is used to declare the runtime environment required for UDF execution, including the Python version, PyPI dependencies, and environment variables. This configuration is serialized as a JSON string and stored in the

 Copyright © Huawei Technologies Co., Ltd.

parameters.runtime_env field of LakeFormation. The Aura engine prepares the environment according to this configuration when running the UDF.

Function

- **Ensuring execution**: The third-party packages and Python versions required by the UDF are installed on the engine as declared, preventing the issue of "local execution success but remote execution failure."
- **Version control**: The Python version (3.10/3.11) and specified package version can be fixed to ensure a consistent runtime environment.
- **Cross-end synchronization**: The environment configuration is persisted with the function metadata and is bound to the function's lifecycle.

When to Use

Table 2-4 Scenario

Scenario	Usage
UDF depends on a third-party Python package.	Declare the package name (for example, **["numpy"]**) in **pypi**/**pip**.
Need to specify a Python version	Set **python_version** (only 3.11 or 3.10 is supported).
Need to inject runtime environment variables	Set **env_vars**.
No special dependency	Pass **None**. The SDK constructs the default environment (**python_version=3.11**).

Construction Methods

runtime_env accepts the following three forms, which the SDK internally unifies into a **RuntimeEnv** object:

Table 2-5 Construction methods

Input Form	Example	Description
JSON string	`'{"python_version": "3.11", "pypi": ["numpy"]}'`	Parsed by **json.loads**.
Mapping/ dict	`{"python_version": "3.11", "pypi": ["numpy"]}`	Directly updated into the internal data structure.

 Copyright © Huawei Technologies Co., Ltd.

Input Form	Example	Description
None	-	Constructs the default **RuntimeEnv** with **python_version** set to 3.11.

Common Keys

Table 2-6 Common keys

Key	Type	Description
python_version	str	Python version. Only 3.11 (default) or 3.10 is supported. Passing other versions raises a ValueError.
pypi/pip	list[str]	List of PyPI dependencies. Merged by deduplicating package names.
env_vars	dict	Runtime environment variables.

Default Values and Verification Behavior

- When **runtime_env** is **None**, the SDK constructs an empty **RuntimeEnv** and automatically sets **python_version** to **3.11**.
- If **runtime_env** provides a **python_version**, the SDK verifies its value. A ValueError is raised if the value is not 3.11 or 3.10.
- **RuntimeEnv** is a mutable mapping (like a dictionary) that supports the **to_dict()**, **to_json()**, and **merge(other)** methods. The merge method uses a deep merge strategy: Dictionaries are merged recursively. Lists are merged with deduplication (for keys like pypi/pip/dependencies, deduplication is performed based on package names). For scalar conflicts, the left value takes precedence.

Return Value

Returns a **Function** object containing the **version_id** assigned by LakeFormation, **resource_uris** (OBS package addresses), and parameters (metadata such as the signature and runtime environment).

2.1.4.13 Exception Classes

When the SDK fails to call external services such as LakeFormation or OBS or during internal processing (such as type inference or UDF packaging), it throws the corresponding exception classes. You can quickly determine whether the issue is related to the service, storage, parameters, or types based on the exception class. All SDK exceptions are inherited from the base class **AIDatalakeCatalogError**.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-7 Exception classes

Exception Class	Description	Key Attribute
AIDatalakeCatalogError	Base class of all SDK exceptions.	-
LakeFormationAPIError	Failed to call the LakeFormation REST API.	status_code, function_name
OBSOperationError	Failed to upload or delete data to or from OBS.	obs_uri
TypeMappingError	The Python/LakeFormation/Aura/Arrow type cannot be mapped.	-
FunctionNotFoundError	The specified function is not found in LakeFormation.	function_name, database
FunctionAlreadyExistsError	A function with the same name already exists in LakeFormation.	function_name, database
SignatureInferenceError	Failed to infer the signature from the callable object.	callable_name
UDFPackagingError	Failed to package the UDF (into a ZIP file).	-

NOTE

Parameter verification errors (such as invalid function names and out-of-bounds resource parameters) are directly thrown as ValueError and do not belong to the preceding SDK exception hierarchy.

2.1.5 Aura DataFrame SDK (Python)

2.1.5.1 Aura DataFrame SDK Overview

Aura DataFrame SDK provides Pandas-like Python DataFrame development capabilities, enabling you to write data processing jobs using Python. Leveraging the powerful computing power of the Aura kernel, it offers easy-to-use and efficient data processing functions for data scientists, AI engineers, and other professionals.

Aura DataFrame SDK is built upon the open-source Ibis Python DataFrame framework, integrating the Ibis frontend framework with the Aura engine. You can compose data processing scripts using the familiar Ibis DataFrame API. The Ibis framework translates these Python APIs into executable SQL statements for the Aura engine, ensuring efficient execution of computational logic within the engine. The SDK supports Ibis APIs. For details, refer to **Ibis official documentation**.

The Aura DataFrame SDK supports the following environments:

 Copyright © Huawei Technologies Co., Ltd.

- Python 3.11 is recommended.
- The Python version of the UDF runtime environment supports only the 3.10 and 3.11 series. Version 3.11.9 is recommended.

Table 2-8 Python versions supported in different service scenarios

Service Scenario	Built-in Python Packages	Supported Python Versions
Python UDF	• cloudpickle • msgpack • pyarrow • protobuf • pandas • numpy • redis • opentelemetry-api • opentelemetry-sdk • opentelemetry-exporter-otlp-proto-grpc	Python 3.10 and Python 3.11
Multimodal data processing	• av (PyAV) • pillow • soundfile • decord • opencv-python-headless • numpy • scipy	Python 3.11
AI inference/Model service (UDF)	• torch • torchvision • onnx • onnxruntime • torch_npu • onnxruntime-cann • vlm • yreserve • Ascend-cann-toolkit • kernels	Python 3.11

If you need to manage objects and permissions during DataFrame development (for example, creating tables with special formats and configuring object authorization), see "Development Process Using Aura SQL" in the *Aura Syntax Reference*.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

The Aura DataFrame SDK supports only AuraJobV2 endpoints.

Aura DataFrame SDK Functions

The Aura DataFrame SDK provides the following types of APIs:

Table 2-9 Aura DataFrame SDK

API	Function
Session APIs	Establishes and manages sessions, serving as the entry point for using the SDK. This includes establishing connections (**ai_lake.connect**), configuring the UDF deployment workspace (**set_function_staging_workspace**), loading datasets (**load_dataset**), and executing SQL statements (**sql**). A session must be established via these APIs before any data operations.
Dataset APIs	Performs data processing on datasets, which is the core of development. This includes column operations (**select_columns**, **add_column**, **drop_columns**, **rename_columns**), row filtering (**filter**, **limit**, **take**), aggregation statistics (**aggregate**, **groupby**, **max**/**min**/**mean**), data transformation (**map**, **map_batches**, **flat_map**, **order_by**, **join**), execution triggering (**execute**, **show**), and execution plan generation (**explain_plan**/**explain_performance**).
Table APIs	Manages database tables, including creating a table (**create_table**), deleting a table (**drop_table**/**delete_table**), describing a table (**describe_table**), listing tables (**list_tables**), opening a table (**open_table**), as well as table CRUD operations (**insert**, **update**, **delete**), and column management (**add_columns**, **drop_columns**, **alter_column_type**, **rename_column**).
I/O APIs	Writes data to the lakehouse storage formats to achieve seamless integration with the data lakes. APIs such as **write_iceberg**, **write_orc**, and **write_parquet** are supported.
UDF APIs	Manages and uses UDFs, including creating functions (**create_scalar_function**, **create_agg_function**, and **create_table_function**), retrieving functions (**get_function**), listing functions (**list_functions**), and deleting functions (**delete_function**). Scalar UDFs, class UDFs, UDTFs and UDAFs are supported.

 Copyright © Huawei Technologies Co., Ltd.

API	Function
AI APIs	Invokes large language model capabilities, including text generation (**ai_generate**), model registration (**create_model**), model listing (**list_models**), model details (**describe_model**), and model deletion (**drop_model**). Large model inference can be directly performed in DataFrames.

2.1.5.2 Obtaining and Installing the Aura DataFrame SDK

You can install the Aura DataFrame SDK with a single **pip** command. The installation process automatically downloads the third-party libraries required by the SDK. This section describes how to obtain and install the SDK.

Prerequisites

Python 3.11 has been installed. For details, see **Python Environment Configuration**.

Obtaining the SDK

Download the SDK file. You are advised to download the **.sha256** file as well, which can be used to verify the integrity of the SDK package. For details about the verification method, see **Verifying the Integrity of the SDK Package**.

Table 2-10 Obtaining SDK files

SDK File	Description	Download URL
huawei_aura_connectorapi-x.x.x-py3-none-any.whl	Supports executing SQL queries via native APIs or standard database interfaces (PEP249).	• **SDK installation package** • **Verification file (SHA-256)**
huawei_aura_frame-x.x.x.x-py3-none-any.whl	Based on the Ibis framework, supports writing data processing jobs using the Dataframe/Dataset-like syntax and automatically converting them to SQL for execution on the Aura engine.	• **SDK installation package** • **Verification file (SHA-256)**

Verifying the Integrity of the SDK Package

When downloading the SDK installation package, you also need to download the **.sha256** file to verify the integrity of the SDK package. This section outlines the steps to use the **.sha256** file for verifying the integrity of the SDK package.

 Copyright © Huawei Technologies Co., Ltd.

1. **Querying the SHA-256 hash value of the SDK installation package**
   - **On Windows**
     i. Search for **cmd** in the **Start** menu to open the command window.
     ii. Run **cd** to go to the directory where the SDK installation package is saved.

     For example, if the SDK installation package is saved in the **SDK** folder on drive D, enter **cd D:\SDK**.
     iii. Run the following command:
     ```
     certutil -hashfile <SDK installation package name> SHA-256
     ```
     Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **certutil -hashfile huawei_aura_connectorapi-x.x.x-py3-none-any.whl SHA-256**.
     iv. After the command is executed, the command prompt will display the SHA-256 hash value of the SDK installation package.
   - **On Linux**
     i. Run **cd** to go to the directory where the SDK installation package is saved.

     For example, if the SDK installation package is saved in the **SDK** folder in the home directory, enter **cd ~/SDK**.
     ii. Enter the query command:
     ```
     sha256sum <SDK installation package name>
     ```
     Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **sha256sum sdk - huawei_aura_connectorapi-x.x.x-py3-none-any.whl**.
     iii. After the command is executed, the terminal will display the SHA-256 hash value of the SDK installation package.
2. **Comparing hash values**
   a. Open the downloaded **.sha256** file using the text editor.
   b. Find the SHA-256 hash value of the SDK installation package in the **.sha256** file.

   It is typically a long string, for example, **a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6***xxxxxxxxxxxxxxxxxxxxxxxx*.
   c. Compare the SHA-256 hash value of the SDK installation package obtained in **1. Querying the SHA-256 hash value of the SDK installation package** with the hash value in the **.sha256** file.
      - If the two hash values are identical, the SDK installation package was not altered during the download process and the file is intact.
      - If the hash values are inconsistent, an error occurred during the download of the SDK installation package or that the file was maliciously modified.

        In this case, you are advised to download the SDK installation package and its corresponding **.sha256** file again and then verify the package.

 Copyright © Huawei Technologies Co., Ltd.

Installing the SDK

Step 1 Open the CLI and go to the directory where the SDK file is stored.

Step 2 Run the following command to install the SDK. During the installation, the third-party dependencies are automatically downloaded.

```
pip install huawei_aura_connectorapi-x.x.x-py3-none-any.whl
pip install huawei_aura_frame-x.x.x.x-py3-none-any.whl
```

x.x.x.x and *x.x.x* indicate the version numbers. Replace them with the actual ones.

Step 3 Run the following command to check the SDK version:

```
python -c "import aura_frame; print(aura_frame.__version__)"
```

If the output similar to the following, the SDK is successfully installed.

```
3.9.0.1
```

----End

2.1.5.3 Getting Started with the Aura DataFrame SDK

This section describes the complete workflow of developing data applications using the Aura DataFrame SDK.

Workflow Overview

Step	Operation	Description
Step 1: Obtain And Install The Sdk Installation Package.	Obtain and install the SDK installation package.	Install the Aura DataFrame SDK package.
Step 2: Establish a Connection.	Establish a connection.	Use **ai_lake.connect()** to create a session. Before using UDFs, configure and deploy a workspace.
Step 3: Prepare Data.	Prepare data.	Load an existing dataset, create a data table, and write data.
Step 4: Process Data.	Process data.	Use DataFrame APIs to perform operations such as filtering, aggregation, sorting, and joining.
Step 5: Perform Advanced Data Development Operations.	Perform advanced data development operations.	Use UDFs and multi-modal processing capabilities based on service requirements.

 Copyright © Huawei Technologies Co., Ltd.

Step	Operation	Description
Step 6: View the Job Execution Results.	View the job execution results.	Check whether the job is running as expected and close the session to release resources.

Step 1: Obtain And Install The Sdk Installation Package.

After obtaining the SDK package, open the CLI, go to the directory where the SDK file is stored, and run the following command to install the SDK. During the installation, the third-party dependencies will be automatically downloaded.

```
pip install huawei_aura_connectorapi-x.x.x-py3-none-any.whl
pip install huawei_aura_frame-x.x.x.x-py3-none-any.whl
```

x.x.x.x and *x.x.x* indicate the version numbers. Replace them with the actual ones. For details, see **Obtaining and Installing the Aura DataFrame SDK**.

Step 2: Establish a Connection.

Use ai_lake.connect() to create a session connection to the Aura endpoint. Before using UDFs, you need to configure and deploy a workspace.

For details, see **Establishing a Connection with the AuraJobV2 Endpoint**.

Create a session and configure connection information (endpoint, workspace ID, catalog, AK/SK, and database).

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),          # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv ("aura_endpoint_id"),     # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),  # AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),    # Workspace ID
    lf_catalog_name=os.getenv ("lf_catalog_name"),       # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
    default_database="postgres",                         # Default database
    use_single_cn_mode=True,                             # Whether to enable the single-CN mode
    logging_level=logging.INFO,                          # Set the log level.
)
```

Step 3: Prepare Data.

Create a data table, write data, and load an existing dataset.

1. Create a table and data.
   a. Create a database in LakeFormation, for example, **database1**. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
   b. Create a data table (if the target table does not exist in the database).

      In the **create_table** statement, you can use the **store** parameter to specify the table format (such as ORC, PARQUET, or ICEBERG) and the **partition_by** parameter to specify the partition column.

 Copyright © Huawei Technologies Co., Ltd.

```
import ibis
table_name = "table_name"
table_schema = ibis.schema({"col_a": "int32", "col_b": "string"})
con.sql(f'DROP TABLE IF EXISTS database1."{table_name}";')
con.create_table(
    table_name,
    schema=table_schema,
    database="database1",
    table_format="orc",
    if_not_exists=True,
)
```

c. Run the SQL statement to write data.

```
con.sql("INSERT INTO database1.table_name VALUES(1, 's1'), (2, 's2')")
```

For details, see Creating a Table and Writing Data to the Table.

2. Load an existing dataset.

```
ds = con.load_dataset("table_name", database="database1")  # Load a table from the specified database and create a table object. Replace table_name and database1 with the actual table name and database name.
```

Step 4: Process Data.

Use DataFrame APIs to perform operations such as filtering, aggregation, sorting, and joining on data.

DataFrame operations use the lazy loading mechanism. The following operations only record the calculation intent. The Aura engine executes the operations and returns the Pandas DataFrame only when **execute()** is called.

For details, see **Data Processing and Analysis**.

```
# Select columns.
ds=ds.select_columns([ds.col_a, ds.col_b])
# Filter rows.
ds=ds.filter(ds.col_a > 0)
# Group and aggregate (aggregate the numeric column col_a).
ds=ds.groupby("col_a").aggregate(sum_b=ds.col_a.sum(), avg_b=ds.col_a.mean())
# Sort.
ds=ds.order_by(ds.col_a.desc())
# Trigger execution and return the Pandas DataFrame (key step).
df=ds.execute()
print(df)
```

Step 5: Perform Advanced Data Development Operations.

Use UDFs and multi-modal processing capabilities based on service requirements.

- For details about how to register and use scalar UDFs, class UDFs, and UDAFs, see **Using Scalar UDFs to Process Data**, **Using Class UDFs to Process Data** and **Using UDAFs for Aggregation Statistics**.
- For details about UDF types, registration methods, and development standards, see **UDFs**.

Step 6: View the Job Execution Results.

After the job runs successfully, the run logs display the execution process and results (such as table creation, SQL execution, and query result return). You can check whether the job is running as expected based on these logs.

For more examples, see **Submitting a Job Using the Aura DataFrame SDK**.

After the data operation is complete, close the session to release resources.

 Copyright © Huawei Technologies Co., Ltd.

```
con.close()  # Close the session and release resources.
```

2.1.5.4 Submitting a Job Using the Aura DataFrame SDK

2.1.5.4.1 Establishing a Connection with the AuraJobV2 Endpoint

This section describes how to create a session connection with the AuraJobV2 endpoint by calling **ai_lake.connect()**. All subsequent data operations are performed based on this connection.

Prerequisites

- The Aura DataFrame SDK has been installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.
- An AuraJobV2 endpoint has been created, and the endpoint name has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- A LakeFormation instance has been created, where a catalog and a database are ready. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
- A parallel file system has been created. For details, see **Creating a Parallel File System**.
- The parameters in **os.getenv()** involved in the code have been configured as environment variables.
- The Python 3.11 has been installed.

Example

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database="postgres",                       # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
```

The parameters of **ai_lake.connect()** are as follows:

Parameter	Description	Type	Default Value
aura_endpoint	Specifies the endpoint information of the connection. For details, see **Regions and Endpoints**.	str	None
aura_endpoint_id	Specifies the ID of the AuraJobV2 endpoint to be connected.	str	None

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Type	Default Value
aura_endpoint_name	Specifies the name of the AuraJobV2 endpoint to be connected.	str	None
aura_workspace_id	Specifies the ID of the workspace where the AuraJobV2 endpoint to be connected is located.	str	None
lf_catalog_name	Specifies the name of the LakeFormation catalog to be connected.	str	None
access_key	Specifies the Access Key ID (AK) of the connection.	str	None
secret_key	Specifies the Secret Key (SK) of the connection.	str	None
default_database	Specifies the name of the default database to be connected.	str	None
use_single_cn_mode	Specifies whether to enable the single-CN mode.	bool	False
logging_level	Specifies the log level.	int	logging.INFO

The following example demonstrates the complete process of establishing a connection and verifying connectivity:

```
import os
import logging
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),          # Endpoint where the AuraJobV2 endpoint is located,
for example:
    aura_endpoint_id=os.getenv ("aura_endpoint_id"),     # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),  # AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),    # Workspace ID
    lf_catalog_name=os.getenv ("lf_catalog_name"),       # LakeFormation Catalog name
    lf_instance_id=os.getenv ("lf_instance_id"),         # LakeFormation instance ID
    project_id=os.getenv ("project_id"),                 # Project ID
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
    default_database="postgres",                         # Default database
    use_single_cn_mode=True,                             # Whether to enable the single-CN mode
    logging_level=logging.INFO,                          # Set the log level.
)
try:
    with con.sql("select 1;") as cur:                    # Execute the SQL statement to verify the connectivity.
        result = cur.fetchall()
    print("Connection successful. Result of select 1:", result)
finally:
    con.close()                                          # Close the connection.
```

 Copyright © Huawei Technologies Co., Ltd.

Helpful Links

Session APIs

2.1.5.4.2 Configuring the UDF Deployment Workspace

Before using UDFs, you need to configure a temporary workspace required for UDF deployment.

Prerequisites

- The Aura DataFrame SDK has been installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.
- An AuraJobV2 endpoint has been created, and the endpoint name has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- A LakeFormation instance has been created, where a catalog and a database are ready. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
- A parallel file system has been created. For details, see **Creating a Parallel File System**.
- The parameters in **os.getenv()** involved in the code have been configured as environment variables.
- The Python 3.11 has been installed.

Example

```
import os
from aura_frame.multimodal import ai_lake

target_database = "test"
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                  # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
try:
    con.set_function_staging_workspace(
        obs_directory_base=os.getenv("obs_directory_base"),  # Storage path for UDFs in OBS
        obs_bucket_name=os.getenv("obs_bucket_name"),        # OBS bucket name
        obs_server=os.getenv("obs_server"),                  # OBS endpoint
        access_key=os.getenv("access_key"),                  # Access Key ID (AK)
        secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
    )
    print("The UDF deployment workspace has been configured.")
finally:
    con.close()
```

The parameters of **set_function_staging_workspace()** are as follows:

 Copyright © Huawei Technologies Co., Ltd.

Table 2-11 Parameter description

Parameter	Description	Type	Default Value
obs_directory_base	Path for storing UDFs in the parallel file system.	str	None
obs_bucket_name	Name of the parallel file system.	str	None
obs_server	OBS access address. For details, see *Endpoints and Domain Names*. For endpoint information of the specific connection, see **Regions and Endpoints**.	str	None
access_key	AK/SK. For details, see **AK/SK Obtaining**.	str	None
secret_key		str	None
security_token	Security token (required when a temporary AK/SK pair is used).	str	None

Helpful Links

Session APIs

2.1.5.4.3 Creating a Table and Writing Data to the Table

This section describes how to create a data table and write data into the table. This provides a data foundation for subsequent queries and analysis.

Prerequisites

- The Aura DataFrame SDK has been installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.
- An AuraJobV2 endpoint has been created, and the endpoint name has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- A LakeFormation instance has been created, where a catalog and a database are ready. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
- A parallel file system has been created. For details, see **Creating a Parallel File System**.
- The parameters in **os.getenv()** involved in the code have been configured as environment variables.
- The Python 3.11 has been installed.

 Copyright © Huawei Technologies Co., Ltd.

Example

This example creates three data tables (**text_samples**, **sales_line_items**, and **category_sales**) are writes text, sales details, and category sales data respectively. This provides a data foundation for subsequent queries, analysis, and UDF usage in **3.1.6.4.4**, **3.1.6.4.5**, **3.1.6.4.6**, **3.1.6.4.7**, **3.1.6.4.8**, **3.1.6.4.9**, **3.1.6.4.10**.

1. Import dependencies and connect to the AuraJobV2 endpoint.
2. Create the **text_samples** table (containing the **id** and **content** columns) and insert five text records.
3. Create the **sales_line_items** table (containing the **quantity**, **unit_price**, **discount_rate**, **tax_rate**, **return_flag**, **line_status**, and **ship_date** columns) and insert five sales records.
4. Create the **category_sales** table (containing the **category** and **amount** columns) and insert 12 category sales records.
5. Disable the AuraJobV2 endpoint connection.

```
import os
import logging
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
try:
    table_name = "text_samples"
    table_schema = ibis.schema({"id": "int32", "content": "string"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (id, content) VALUES
        (1, 'Hello, how are you today?'),
        (2, 'The quick brown fox jumps over the lazy cat.'),
        (3, 'Machine learning is transforming the way we process data.'),
        (4, 'Natural language processing enables computers to understand human language.'),
        (5, 'Tokenization is the first step in text processing pipelines.')
    '''
    con.sql(insert_sql)
    print(f"Table {table_name} has been created and data has been written.")
    table_name = "sales_line_items"
    table_schema = ibis.schema({
        "quantity": "int32",
        "unit_price": "float64",
        "discount_rate": "float64",
```

 Copyright © Huawei Technologies Co., Ltd.

```
        "tax_rate": "float64",
        "return_flag": "string",
        "line_status": "string",
        "ship_date": "date",
    })
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (quantity, unit_price, discount_rate, tax_rate,
return_flag, line_status, ship_date) VALUES
        (10, 1000.00, 0.05, 0.08, 'N', 'O', '1998-08-01'),
        (20, 2000.00, 0.10, 0.05, 'R', 'F', '1998-09-15'),
        (15, 1500.00, 0.08, 0.07, 'N', 'O', '1998-07-20'),
        (25, 2500.00, 0.05, 0.08, 'R', 'F', '1998-10-01'),
        (30, 3000.00, 0.02, 0.10, 'N', 'O', '1998-08-15')
    '''
    con.sql(insert_sql)
    print(f"Table {table_name} has been created and data has been written.")
    table_name = "category_sales"
    table_schema = ibis.schema({"category": "string", "amount": "float64"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    insert_sql = f'''
    INSERT INTO "{target_database}"."{table_name}" (category, amount) VALUES
        ('Electronics', 1299.00),
        ('Electronics', 799.00),
        ('Electronics', 599.00),
        ('Electronics', 1099.00),
        ('Books', 29.99),
        ('Books', 49.99),
        ('Books', 19.99),
        ('Books', 39.99),
        ('Clothing', 89.00),
        ('Clothing', 159.00),
        ('Clothing', 49.00),
        ('Clothing', 199.00)
    '''
    con.sql(insert_sql)
    print(f"Table {table_name} has been created and data has been written.")
finally:
    con.close()
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**

2.1.5.4.4 Establishing a Connection and Querying Data from a Specified Table

This section describes how to establish a connection to the AuraJobV2 endpoint, query data from the created table, and return the query results.

 Copyright © Huawei Technologies Co., Ltd.

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

Example

This example establishes a connection to the Aura service, queries text data from the **text_samples** table, and prints the results.

1. Import dependencies and connect to the AuraJobV2 endpoint.
2. Load the **text_samples** dataset.
3. Select the **id** and **content** columns.
4. Execute the query and print the results.
5. Disable the AuraJobV2 endpoint connection.

```
import os
import logging
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database=target_database,                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
try:
    ds = con.load_dataset("text_samples", database=target_database)
    ds = ds.select_columns([ds.id, ds.content])
    df = ds.execute()
    print(df)
finally:
    con.close()
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**

2.1.5.4.5 Data Processing and Analysis

This section demonstrates the complete Aura DataFrame processing workflow, including filtering, column calculation, grouping and aggregation, and sorting.

Scenario

In a sales data analysis scenario, it is necessary to calculate summary metrics for order details up to a specific cutoff date. The data should be grouped by return

 Copyright © Huawei Technologies Co., Ltd.

flag and order status. The metrics to be calculated include the total quantity, total base price, total discounted price, total tax-inclusive price, average value of each metric, and number of orders. By leveraging the chain operations of Aura DataFrame, you can sequentially perform filtering, derived column calculation, grouping and aggregation, and sorting. This achieves a data processing workflow equivalent to SQL, providing data support for sales reports and business decisions.

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

Example

This example performs Aura DataFrame operations such as filtering, derived column calculation, grouping and aggregation, and sorting on the **sales_line_items** table to demonstrate the complete data processing and analysis workflow.

1. Import dependencies and connect to the AuraJobV2 endpoint.
2. Load the **sales_line_items** dataset.
3. Filter records where **ship_date** is before the cutoff date.
4. Calculate derived columns such as the discounted price and tax-inclusive price.
5. Group the data by **return_flag** and **line_status** and then perform aggregate calculations (**sum**, **avg**, and **count**) within the groups.
6. Sort the data by **return_flag** and **line_status**.
7. Execute the query and print the results.
8. Disable the AuraJobV2 endpoint connection.

```
import os
from datetime import datetime, timedelta
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),              # Secret Key (SK)
    default_database=target_database,                    # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
try:
    # Load the dataset.
    ds = con.load_dataset("sales_line_items", database=target_database)
    cutoff = datetime(1998, 12, 1) - timedelta(days=90)
    # Filtering: Filter out rows that do not meet the conditions.
    q = ds.filter(expr=ds.ship_date <= cutoff)
    # Calculate derived columns.
```

 Copyright © Huawei Technologies Co., Ltd.

```
    discount_price = ds.unit_price * (1 - ds.discount_rate)
    charge = discount_price * (1 + ds.tax_rate)
    # Group by and aggregate.
    q = q.groupby(["return_flag", "line_status"])
    q = q.aggregate(
        sum_qty=ds.quantity.sum(),
        sum_base_price=ds.unit_price.sum(),
        sum_disc_price=discount_price.sum(),
        sum_charge=charge.sum(),
        avg_qty=ds.quantity.mean(),
        avg_price=ds.unit_price.mean(),
        avg_disc=ds.discount_rate.mean(),
        count_order=ds.count(),
    )
    # Sort.
    q = q.order_by(["return_flag", "line_status"])
    # Trigger execution.
    df = q.execute()
    print(df)
finally:
    con.close()
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**

2.1.5.4.6 Using Scalar UDFs to Process Data

This section describes how to register the custom calculation logic as a scalar UDF in the database and call it in Aura DataFrame to implement row-level data processing.

Scenario

In a sales data analysis scenario, it is necessary to calculate the final price including tax based on the original price, discount rate, and tax rate. The calculation logic involves the combined operations on multiple fields. Expressing this logic using built-in SQL functions is not intuitive and difficult to reuse. By encapsulating the calculation logic into a scalar UDF and registering it in the database, you can use it in the DataFrame like a built-in function. This makes the code clearer and the logic easier to maintain.

Notes and Constraints

- Only user-defined functions written in Python are supported.
- Users are responsible for managing Python environment dependencies. Compatibility among library versions must be ensured.
- The runtime environment requires Python 3.11.
- Input parameters and return values of scalar UDFs must have explicitly specified data types.
- It is recommended to call **delete_function** to clean up functions with the same name before registering a UDF to avoid errors during repeated execution.

 Copyright © Huawei Technologies Co., Ltd.

Prerequisites

A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.

Example

This example defines a scalar UDF to calculate the final tax-inclusive price. After registering it in the database, it is called in Aura DataFrame using the **map** method to perform row-level calculations on each row of the **sales_line_items** table.

1. Define the scalar UDF **calc_final_price**, which calculates the final tax-inclusive price based on the unit price, discount rate, and tax rate.
2. Connect to the AuraJobV2 endpoint and set the UDF workspace.
3. Delete the existing UDF with the same name (repeated execution is supported).
4. Register the scalar UDF to the target database.
5. Load the **sales_line_items** dataset.
6. Call the UDF using the **map** method to calculate the **final_price** column for each row of data.
7. Select the related columns, execute the query, and print the results.
8. Disable the AuraJobV2 endpoint connection.

```
import os
from aura_frame.multimodal import ai_lake

def calc_final_price(price: float, discount: float, tax: float) -> float:
    return price * (1 - discount) * (1 + tax)

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                    # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),        # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                  # OBS endpoint
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
)
try:
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("calc_final_price", database=target_database, if_it_exists=True)
    # Register the scalar UDF.
    udf = con.create_scalar_function(
        calc_final_price,
```

 Copyright © Huawei Technologies Co., Ltd.

```
        database=target_database,
    )
    # Use the UDF in DataFrame.
    ds = con.load_dataset("sales_line_items", database=target_database)
    ds = ds.map(fn=udf, on=[ds.unit_price, ds.discount_rate, ds.tax_rate], as_col="final_price")
    ds = ds.select_columns(ds.unit_price, ds.discount_rate, ds.tax_rate, ds.final_price)
    df = ds.execute()
    print(df)
finally:
    con.close()
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**
- **UDF APIs**

2.1.5.4.7 Using Class UDFs to Process Data

Class UDFs are implemented based on the object-oriented programming paradigm, making them suitable processing logic that requires initialization parameters, resource loading, or stateful operations. The **__init__** method is used to initialize resources (such as loading a model), the **__call__** method is used to perform calculations on each row of data, and the **__del__** method can be used to clean up resources as needed.

Scenario

In a text processing scenario, a SentencePiece tokenization model is required to perform subword tokenization on the text column **content** of the **text_samples** table. The model file needs to be loaded at once during UDF initialization and then reused for each row of text. By using a class UDF, the **spiece.model** file can be loaded and EOS encoding options configured in **__init__**. The **__call__** method then performs tokenization on each row of text and returns the tokenization results in JSON format, avoiding the overhead of loading the model repeatedly for each row.

Notes and Constraints

- The class must define an __init__ method, and its parameter names must match the parameter names used during call.
- The class must define a **__call__** member method, which serves as the main entry point of the UDF.
- The class can optionally define a **__del__** member method, which does not support passing parameters.
- When a class UDF is called, only constant values can be passed to the **__init__** method—data columns or other expressions are unsupported.

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

 Copyright © Huawei Technologies Co., Ltd.

Preparations

This example uses the SentencePiece tokenization model, so the model file **spiece.model** must be prepared. It is recommended to obtain the SentencePiece tokenizer included with the T5 model from Hugging Face. The acquisition method is as follows:

1. Install the SentencePiece dependency.
   ```
   pip install sentencepiece==0.2.0
   ```
2. Install the Hugging Face command line tool.
   ```
   pip install huggingface-hub
   ```
3. Download the model file to the same directory as the sample script.
   ```
   hf download google-t5/t5-small spiece.model --local-dir .
   ```

If the **hf download** command is unavailable, manually download the file using a browser.

- Search for **google-t5/t5-small** on the Hugging Face website and go to the model homepage.
- Click the **Files and versions** tab.
- Locate the **spiece.model** file and click the file name to go to its details page.
- Click the download button in the upper right corner of the file content to save the file to the local PC.
- Place the downloaded **spiece.model** file in the same directory as the example script.

Example

This example defines a class UDF based on the SentencePiece tokenization model. It loads the model file in **__init__** and performs subword tokenization on each row of text in **__call__**. After registering it in the database, it processes the text column of the **text_samples** table using the **map** method.

1. Define the class UDF class SPManager, which loads the SentencePiece model file in **__init__**, and performs tokenization on the text in **__call__**, returning JSON-formatted results.
2. Connect to the AuraJobV2 endpoint and set the UDF workspace.
3. Delete the existing UDF with the same name (repeated execution is supported).
4. Register the class UDF to the target database, specifying dependency packages and model files.
5. Obtain the registered UDF operator and load the **text_samples** dataset.
6. Call the UDF via the **map** method to tokenize the **content** column, generating the **pieces_column** column.
7. Select the **content** and **pieces_column** columns, execute the query, and print the results.
8. Disable the AuraJobV2 endpoint connection.

```
import os
from aura_frame.multimodal import ai_lake
from sentencepiece import SentencePieceProcessor

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating
```

 Copyright © Huawei Technologies Co., Ltd.

```
a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database=target_database,                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),        # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                  # OBS endpoint
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
)

class SPManager:
    def __init__(self, model_file: str, *, bos: bool = False, eos: bool = False, reverse: bool = False) -> None:
        sp = SentencePieceProcessor(model_file=model_file)
        opts = ":".join(opt for opt, flag in (("bos", bos), ("eos", eos), ("reverse", reverse)) if flag)
        sp.set_encode_extra_options(opts)
        self.spm = sp
    def __call__(self, input_text: str) -> str:
        import json
        return json.dumps(self.spm.encode(input_text, out_type=str))

try:
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("SPManager", database=target_database, if_it_exists=True)
    # Register the class UDF.
    con.create_scalar_function(
        SPManager,
        database=target_database,
        packages=["sentencepiece==0.2.0"],
        imports=["spiece.model"],
    )
    # Obtain the UDF operator and use it in DataFrame.
    udf = con.get_function("SPManager", database=target_database)
    ds = con.load_dataset("text_samples", database=target_database)
    ds = ds.map(
        fn=udf, on=[ds.content], as_col="pieces_column",
        model_file="spiece.model", eos=True,
    )
    ds = ds.select_columns(ds.content, ds.pieces_column)
    df = ds.execute()
    print(df)
finally:
    con.close()
```

Helpful Links

- Session APIs
- Dataset APIs
- Table APIs
- UDF APIs

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.4.8 Using UDAFs for Aggregation Statistics

User-defined aggregate functions (UDAFs) receive multiple rows of data and return a set of aggregated values. They are applicable to custom statistical scenarios where built-in aggregate functions cannot meet requirements.

Scenario

In a sales data analysis scenario, it is necessary to count the top K highest amounts for each product category. The built-in MAX function can only return the single maximum value and cannot return a Top-K list. By defining a custom UDAF **PythonTopK**, you can collect the amount values from each row in the **accumulate** method and sort them in descending order, merge the intermediate states from different partitions in the **merge** method, and output the Top-K list in JSON format in the **finish** method. This achieves flexible custom aggregation statistics.

Notes and Constraints

- UDAFs must implement the **aggregate_state**, **accumulate**, **merge**, and **finish** methods.
- The implemented **accumulate**, **aggregate_state**, and **merge** methods must be side-effect-free. This means their outputs should depend solely on input parameters and the current aggregation state, not on the factors such as invocation count, order, or external conditions.
- UDAFs cannot be used as window functions, that is, they cannot be used in OVER clauses.
- UDAFs do not support concurrent execution or vectorized computation.
- It is recommended to call delete_function to clean up functions with the same name before registering a UDAF to avoid errors during repeated execution.

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

Example

This example defines a UDAF that counts the top K highest amounts in each group. After registering it in the database, it performs grouping and aggregation on the **category_sales** table by product category, retrieving the top 3 amounts for each group.

1. Define the UDAF class **PythonTopK**, implementing the **aggregate_state**, **accumulate**, **merge**, and **finish** methods to count the top K records with the highest **amount** values in each group.
2. Connect to the AuraJobV2 endpoint and set the UDF workspace.
3. Delete the existing UDF with the same name (repeated execution is supported).
4. Register the UDAF to the target database.
5. Load the **category_sales** dataset.

Copyright © Huawei Technologies Co., Ltd.

6. Construct the aggregation expression using AggregateFnBuilder, grouping by **category** and taking the top 3 **amount** values.
7. Execute the aggregation query and print the results.
8. Disable the AuraJobV2 endpoint connection.

```
import os
import json
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database=target_database,                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),       # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                 # OBS endpoint
    access_key=os.getenv("access_key"),                 # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                 # Secret Key (SK)
)

# Define a UDAF to count the top K highest amounts in each group.
class PythonTopK:
    def __init__(self, k: int):
        self.k = k
        self._agg_state = []
    @property
    def aggregate_state(self):
        return self._agg_state
    def accumulate(self, value: float) -> None:
        if value is not None:
            self._agg_state.append(value)
            self._agg_state.sort(reverse=True)
            self._agg_state = self._agg_state[: self.k]
    def merge(self, other_state) -> None:
        merged = self._agg_state + other_state
        merged.sort(reverse=True)
        self._agg_state = merged[: self.k]
    def finish(self) -> str:
        return json.dumps(self._agg_state)
try:
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("PythonTopK", database=target_database, if_it_exists=True)
    # Register a UDAF.
    udaf = con.create_agg_function(PythonTopK, database=target_database)
    # Use the UDAF in DataFrame: Group by category and take the top 3 highest amounts for each group.
    ds = con.load_dataset("category_sales", database=target_database)
    udaf_builder = AggregateFnBuilder(fn=udaf, on=[ds.amount], as_col="top3", constructor_kwargs={"k": 3})
    summary = ds.aggregate([udaf_builder], by=[ds.category])
    df = summary.execute()
    print(df)
finally:
    con.close()
```

 Copyright © Huawei Technologies Co., Ltd.

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**
- **UDF APIs**

2.1.5.4.9 Directly Using a Registered Scalar UDF

When a UDF has already been registered in the database by another user or script, you can directly retrieve the registered UDF operator via **get_function** and call it in the DataFrame without needing to re-register it.

Scenario

In collaborative team development, UDF registration and usage are often separated: some team members are responsible for writing and registering UDFs, while others are responsible for calling the registered UDFs in the data processing workflows. This example demonstrates how to view the list of registered UDFs in the database, retrieve a specified UDF operator, and directly use it in DataFrame expressions.

Notes and Constraints

- Ensure that the target UDF is already registered in the specified database before use. Function names are stored in lowercase in the database.
- Both scalar UDFs and class UDFs are called using the **map** method, with the **on** parameter passing the input columns of **__call__**.
- For class UDFs, **__init__** parameters are passed via keyword arguments in the **map** method (such as **model_file** and **eos**).

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

Example

This example demonstrates how to view the list of registered UDFs in the database, retrieve the registered scalar UDF and class UDF operators, and directly call them in Aura DataFrame without re-registration.

1. Connect to the Aura service and set the UDF workspace.
2. View the list of registered UDF names in the target database.
3. If **calc_final_price** exists, retrieve the scalar UDF operator, load the **sales_line_items** dataset, and calculate the **final_price** column using the **map** method.
4. If **spmanager** exists, retrieve the class UDF operator, load the **text_samples** dataset, and perform tokenization on the **content** column using the **map** method (passing the **model_file** and **eos** parameters).

 Copyright © Huawei Technologies Co., Ltd.

5. Disable the Aura connection.

```
import os
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                    # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),       # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                 # OBS endpoint
    access_key=os.getenv("access_key"),                 # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                 # Secret Key (SK)
)
try:
    # View the names of all registered UDFs.
    udf_names = con._conn.udf.names(database=target_database)
    print(f"Registered UDFs: {udf_names}")
    if "calc_final_price" in udf_names:
        # Retrieve the registered scalar UDFs.
        calc_final_price_udf = con.get_function("calc_final_price", database=target_database)
        ds = con.load_dataset("sales_line_items", database=target_database)
        # Directly use the registered UDFs in the DataFrame.
        ds = ds.map(
            fn=calc_final_price_udf,
            on=[ds.unit_price, ds.discount_rate, ds.tax_rate],
            as_col="final_price",
        )
        ds = ds.select_columns(ds.unit_price, ds.discount_rate, ds.tax_rate, ds.final_price)
        df = ds.execute()
        print(df)
    if "spmanager" in udf_names:
        # Retrieve the registered class UDFs.
        sp_udf = con.get_function("spmanager", database=target_database)
        ds = con.load_dataset("text_samples", database=target_database)
        # Perform tokenization on the content column via the map method.
        ds = ds.map(
            fn=sp_udf, on=[ds.content], as_col="pieces_column",
            model_file="spiece.model", eos=True,
        )
        ds = ds.select_columns(ds.content, ds.pieces_column)
        df = ds.execute()
        print(df)
finally:
    con.close()
```

Helpful Links

- Session APIs
- Dataset APIs
- Table APIs

 Copyright © Huawei Technologies Co., Ltd.

- **UDF APIs**

2.1.5.4.10 Directly Using a Registered UDAF

After a UDAF has already been registered in the database by another user or script, you can retrieve the registered UDAF operator via **udaf.get** and call it directly in **aggregate** without needing to re-register it.

Scenario

In collaborative team development, UDAF registration and usage are often separated. This example demonstrates how to view the list of registered UDFs in the database, retrieve the registered UDAF operator **pythontopk**, and perform grouping and aggregation on the **category_sales** table by **category**, retrieving the top 3 highest **amount** values for each group.

Notes and Constraints

- Ensure that the target UDAF is already registered in the specified database before use. Function names are stored in lowercase in the database.
- The UDAF constructs an aggregation expression using AggregateFnBuilder, with the **__init__** parameters passed through **constructor_kwargs**.

Prerequisites

- A table has been created and data has been written to it. For details, see **Creating a Table and Writing Data to the Table**.
- The Python 3.11 has been installed.

Example

This example demonstrates how to view the list of registered UDFs in the database, retrieve the registered UDAF operator **pythontopk**, and perform grouping and aggregation on the **category_sales** table by product category to get the top 3 records by amount in each group.

1. Connect to the Aura service and set the UDF workspace.
2. View the list of registered UDF names in the target database.
3. If the **pythontopk** UDAF exists, obtain the UDAF operator.
4. Load the **category_sales** dataset.
5. Construct the aggregation expression using AggregateFnBuilder, grouping by **category** and taking the top 3 **amount** values.
6. Execute the aggregation query and print the results.
7. Disable the Aura connection.

```
import os
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
```

 Copyright © Huawei Technologies Co., Ltd.

```
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database=target_database,                  # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),        # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                  # OBS endpoint
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
)
try:
    # View the names of all registered UDFs.
    udf_names = con._conn.udaf.names(database=target_database)
    print(f"Registered UDFs: {udf_names}")
    if "pythontopk" in udf_names:
        # Retrieve the registered UDAF operator.
        topk_udaf = con._conn.udaf.get(name="pythontopk", database=target_database)
        ds = con.load_dataset("category_sales", database=target_database)
        # Use in aggregate, passing the __init__ parameter via constructor_kwargs.
        udaf_builder = AggregateFnBuilder(
            fn=topk_udaf, on=[ds.amount], as_col="top3",
            constructor_kwargs={"k": 3},
        )
        summary = ds.aggregate([udaf_builder], by=[ds.category])
        df = summary.execute()
        print(df)
finally:
    con.close()
```

Helpful Links

- Session APIs
- Dataset APIs
- Table APIs
- UDF APIs

2.1.5.4.11 Speech Recognition Case Study

This example demonstrates how to define a scalar UDF for speech recognition, converting binary audio data into text using the Whisper model. It highlights the UDF's capabilities in multimodal data processing.

Scenario

In audio data processing scenarios, binary audio data stored in database tables needs to be converted into text using a speech recognition model (Whisper). By defining a scalar UDF in class form, the Whisper model is loaded in the **__init__** method, and speech recognition is performed on each audio segment in the **__call__** method, returning the text result. This example downloads a real speech audio file from **sample-files.com** as test data, writes it into a database table, and processes it using the UDF.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

- The **transformers** package must be installed via **pip** during UDF runtime. Ensure the runtime environment has network access.
- The **__init__** parameters of a scalar UDF only support constant values.
- It is recommended to call **delete_function** to clean up functions with the same name before registering a UDF to avoid errors during repeated execution.

Prerequisites

- The Aura DataFrame SDK has been installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.
- An AuraJobV2 endpoint has been created, and the endpoint name has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- A LakeFormation instance has been created, where a catalog and a database are ready. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
- A parallel file system has been created. For details, see **Creating a Parallel File System**.
- The parameters in **os.getenv()** involved in the code have been configured as environment variables.
- The Python 3.11 has been installed.

Preparations

This example uses the Whisper speech recognition model. You need to prepare the model files, audio files, and Python dependencies.

1. Install the Python dependencies.
   ```
   pip install torch==2.6.0
   pip install transformers==5.3.0
   pip install av
   pip install scipy
   ```
2. Install the Hugging Face command line tool.
   ```
   pip install huggingface-hub
   ```
3. Download the test audio file to the same directory as the example script and rename it **test_audio.wav**.

   .
4. Download the Whisper base model file to the same directory as the example script.
   ```
   hf download openai/whisper-base --local-dir ./whisper_base
   ```

   If the **hf download** command is unavailable, manually download the file using a browser.

   a. Search for **openai/whisper-base** on the Hugging Face website and go to the model homepage.

   b. Click the **Files and versions** tab.

   c. Download the following files and save them to the local **whisper_base** directory:

 Copyright © Huawei Technologies Co., Ltd.

- config.json
- pytorch_model.bin
- preprocessor_config.json
- tokenizer_config.json
- tokenizer.json
- vocab.json
- merges.txt
- normalizer.json
- special_tokens_map.json
- added_tokens.json
- generation_config.json

d. Place the **whisper_base** directory in the same directory as the example script.

Example

This example defines a scalar UDF for speech recognition based on the Whisper model. After writing the test audio into the database, it calls the UDF via the **map_batches** method to convert binary audio data into text.

1. Define the class UDF class **SpeechRecognitionUDF**. Load the Whisper speech recognition model via **transformers** in **__init__**. In __call__, decode the audio binary data using the AV library, resample it to 16 kHz, and perform recognition.
2. Connect to the Aura service and set the UDF workspace.
3. Read the test audio file, create a table, and insert the binary audio data.
4. Delete the existing UDF with the same name (repeated execution is supported).
5. Register the scalar UDF to the target database, specifying dependency packages and model files.
6. Load the **audio_data** dataset.
7. Call the UDF for speech recognition via the **map_batches** method, generating the **hyp_text** column.
8. Select the related columns, execute the query, and print the results.
9. Disable the Aura connection.

```
import os
import ibis
import aura_frame
from aura_frame.multimodal import ai_lake



# Define a scalar UDF for speech recognition.
```

 Copyright © Huawei Technologies Co., Ltd.

```
class SpeechRecognitionUDF:
    def __init__(self, model_path: str) -> None:
        import torch
        from transformers import pipeline
        self._pipeline = pipeline(
            "automatic-speech-recognition",
            model=model_path,
            device=0 if torch.cuda.is_available() else -1,
        )
    def __call__(self, audios: bytes) -> str:
        import numpy as np
        import av
        import io
        from scipy.signal import resample_poly
        container = av.open(io.BytesIO(audios))
        stream = container.streams.audio[0]
        frames = []
        for frame in container.decode(stream):
            frames.append(frame.to_ndarray())
        if not frames:
            raise ValueError("Audio data is None.")
        audio_np = np.concatenate(frames, axis=1)
        if audio_np.ndim > 1:
            audio_np = audio_np.mean(axis=0)
        if audio_np.dtype == np.int16:
            audio_np = audio_np.astype(np.float32) / 32768.0
        elif audio_np.dtype == np.int32:
            audio_np = audio_np.astype(np.float32) / 2147483648.0
        elif audio_np.dtype != np.float32:
            audio_np = audio_np.astype(np.float32)
        # Resample to 16 kHz (required by Whisper).
        target_sr = 16000
        if stream.rate != target_sr:
            audio_np = resample_poly(audio_np, target_sr, stream.rate)
        result = self._pipeline({"raw": audio_np, "sampling_rate": target_sr})
        return result["text"]

try:
    # Read the test audio file.
    audio_path = "test_audio.wav"
    with open(audio_path, "rb") as f:
        audio_bytes = f.read()
    # Create a table and insert audio data.
    table_name = "audio_data"
    table_schema = ibis.schema({"id": "int32", "audio_data": "binary"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    hex_str = audio_bytes.hex()
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (id, audio_data) VALUES
        (1, '\\x{hex_str}'::bytea)
    ''')
    print("Audio data is written.")
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("speechrecognitionudf", database=target_database, if_it_exists=True)
    # Register a scalar UDF for speech recognition.
    con.create_scalar_function(
        SpeechRecognitionUDF,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./whisper_base"],
    )
    # Load the audio dataset.
```

 Copyright © Huawei Technologies Co., Ltd.

```
    ds = con.load_dataset("audio_data", database=target_database)
    # Use the scalar UDF for speech recognition.
    udf = con.get_function("speechrecognitionudf", database=target_database)
    ds = ds.map_batches(
        fn=udf,
        on=[ds.audio_data],
        as_col="hyp_text",
        num_cpus=2,
        memory=4096,
        arch="x86",
        model_path="./whisper_base",
    )
    ds = ds.select_columns(ds.id, ds.audio_data, ds.hyp_text)
    df = ds.execute()
    print(df)
finally:
    con.close()
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**
- **UDF APIs**

2.1.5.4.12 Video Frame Description and Summarization Example

This example demonstrates how to use scalar UDFs and UDAFs to describe video frames and generate video summaries. It extracts frames from a real video file via the AV library, stores them in a database table, uses the BLIP model to generate natural language descriptions for each frame, and employs the T5 model to aggregate these descriptions into a coherent video summary.

Scenario

In video content analysis, key frames need to be extracted from a video, natural language descriptions generated for each frame, and multiple frame descriptions synthesized into a coherent video summary. A scalar UDF in class form loads the BLIP model in **__init__** and generates image descriptions for each frame in **__call__**. A UDAF aggregates multiple frame descriptions and uses the T5 model to generate a summary. This example uses the AV library to extract frames from a real video file as test data.

Notes and Constraints

- The **transformers** package must be installed via **pip** during UDF runtime. Ensure the runtime environment has network access.
- The **__init__** parameters of a scalar UDF only support constant values.
- It is recommended to call **delete_function** to clean up functions with the same name before registering a UDF to avoid errors during repeated execution.

Prerequisites

- The Aura DataFrame SDK has been installed. For details, see **Obtaining and Installing the Aura DataFrame SDK**.

 Copyright © Huawei Technologies Co., Ltd.

- An AuraJobV2 endpoint has been created, and the endpoint name has been obtained. For details, see **Creating an Aura Engine Endpoint**.
- A LakeFormation instance has been created, where a catalog and a database are ready. For details, see **Creating a LakeFormation Instance and Planning Metadata**.
- A parallel file system has been created. For details, see **Creating a Parallel File System**.
- The parameters in **os.getenv()** involved in the code have been configured as environment variables.
- The Python 3.11 has been installed.

Preparations

This example uses the BLIP image captioning model and T5 text summarization model. You need to prepare the model files, video files, and Python dependencies.

1. Install the Python dependencies.
   ```
   pip install torch==2.6.0
   pip install transformers==5.3.0
   pip install pillow
   pip install av
   ```
2. Install the Hugging Face command line tool.
   ```
   pip install huggingface-hub
   ```
3. Download the test video file to the same directory as the example script and rename it **test_video.mp4**.

   Download a short MP4 video that contains real-world scenes (recommended under 10 seconds).
4. Download the BLIP image captioning model file to the same directory as the example script.
   ```
   hf download Salesforce/blip-image-captioning-base --local-dir ./blip-image-captioning-base
   ```

   If the **hf download** command is unavailable, manually download the file using a browser.

   a. Search for **Salesforce/blip-image-captioning-base** on the Hugging Face website and go to the model homepage.

   b. Click the **Files and versions** tab.

   c. Download the following files and save them to the local **blip-image-captioning-base** directory:

      - config.json
      - preprocessor_config.json
      - pytorch_model.bin
      - special_tokens_map.json
      - tokenizer_config.json
      - tokenizer.json
      - vocab.txt

 Copyright © Huawei Technologies Co., Ltd.

d. Place the **blip-image-captioning-base** directory in the same directory as the example script.

5. Download the T5 text summarization model file to the same directory as the example script.

```
hf download google-t5/t5-small --local-dir ./t5-small
```

If the **hf download** command is unavailable, manually download the file using a browser.

a. Search for **google-t5/t5-small** on the Hugging Face website and go to the model homepage.

b. Click the **Files and versions** tab.

c. Download the following files and save them to the local **t5-small** directory:

- config.json
- pytorch_model.bin
- spiece.model
- tokenizer.json
- tokenizer_config.json
- generation_config.json

d. Place the **t5-small** directory in the same directory as the example script.

Example

This example integrates the entire process of video frame description and summary generation into a single executable code block. It includes connection management, scalar UDF definition (generating frame descriptions based on the BLIP model), UDAF definition (generating video summaries based on the T5 model), function registration, video frame extraction, frame-level description generation, and video summary aggregation.

1. Import dependencies and define the **create_connect** context manager to encapsulate the creation and release of Aura service connections.
2. Define the scalar UDF class VideoContentAnalyzer. Load the BLIP model in **__init__**. In **__call__**, generate descriptions for each frame of JPEG data, analyze content features, and return results in JSON format.
3. Define the **VideoSummaryGenerator** class, load the T5 model, and aggregate multiple frame descriptions to generate a video summary.
4. Define the UDAF class **VideoSummarySystem**. Use **accumulate** to collect descriptions of each frame, and **finish** to call the T5 model to generate a video summary.
5. Connect to the AuraJobV2 endpoint and set the UDF workspace. Delete the existing UDFs and UDAFs with the same name, and then register them to the target database.
6. Use the AV library to extract a maximum of 10 frames (in JPEG format) from the video file, create the **video_frames** table, and write the binary frame data.

 Copyright © Huawei Technologies Co., Ltd.

7. Load the **video_frames** dataset and use **map_batches** to call the scalar UDF to analyze each frame and generate the **descriptions** column.
8. Call the UDAF via **AggregateFnBuilder** to aggregate multiple frame descriptions and generate a video summary, execute the query, and print the results.

```
import ibis
import typing
import os
from contextlib import contextmanager
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
@contextmanager
def create_connect():
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
    con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
        aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
        aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
        aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
        lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
        access_key=os.getenv("access_key"),                # Access Key ID (AK)
        secret_key=os.getenv("secret_key"),                # Secret Key (SK)
        default_database=target_database,                      # LakeFormation database name
        use_single_cn_mode=True,   # Whether to enable the single-CN mode
        logging_level=logging.INFO # Set the log level.
    )
    print(f"Establish ai_lake.connect: {con}")
    try:
        yield con
    finally:
        # Close the connection.
        con.close()

class VideoContentAnalyzer:
    def __init__(self, model_name="./blip-image-captioning-base"):
        import torch
        from transformers import BlipProcessor, BlipForConditionalGeneration
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.processor = BlipProcessor.from_pretrained(model_name)
        self.model = BlipForConditionalGeneration.from_pretrained(
            model_name,
            torch_dtype=torch.float16 if self.device == "cuda" else torch.float32,
        ).to(self.device)
        self.model.eval()
    def __call__(self, frame: bytes, frame_index: int) -> str:
        import json
        analyse = self.analyze_frame(frame, frame_index)
        return json.dumps({"description": analyse})
    def analyze_frame(self, frame: bytes, frame_index: int):
        try:
            frame_description = self._generate_frame_description(frame)
            content_features = self._analyze_content_features(frame_description)
            analyse = {
                "description": frame_description,
                "content_type": content_features["content_type"],
                "key_objects": content_features["key_objects"],
                "scene_context": content_features["scene_context"],
            }
        except Exception:
            analyse = {}
        return analyse
    def _generate_frame_description(self, frame: bytes):
        import io
```

 Copyright © Huawei Technologies Co., Ltd.

```
        from PIL import Image
        import torch
        pil_img = Image.open(io.BytesIO(frame)).convert("RGB")
        inputs = self.processor(images=pil_img, return_tensors="pt").to(self.device)
        with torch.no_grad():
            outputs = self.model.generate(
                **inputs,
                max_length=50,
                num_beams=5,
                early_stopping=True,
            )
        description = self.processor.decode(outputs[0], skip_special_tokens=True)
        return description.strip()
    def _analyze_content_features(self, description):
        description_lower = description.lower()
        content_categories = {
            "educational": ["tutorial", "lesson", "explain", "teach", "learn", "education"],
            "entertainment": ["funny", "comedy", "entertainment", "show", "performance"],
            "nature": ["outdoor", "nature", "landscape", "mountain", "forest", "animal"],
            "sports": ["sport", "game", "player", "team", "match", "competition"],
            "food": ["food", "cooking", "recipe", "meal", "restaurant", "kitchen"],
            "technology": ["computer", "tech", "device", "electronic", "software"],
            "people": ["person", "people", "man", "woman", "child", "group"],
        }
        detected_categories = []
        for category, keywords in content_categories.items():
            if any(keyword in description_lower for keyword in keywords):
                detected_categories.append(category)
        common_objects = [
            "person", "people", "man", "woman", "child", "car", "building",
            "tree", "house", "street", "water", "sky", "food", "animal",
        ]
        key_objects = [obj for obj in common_objects if obj in description_lower]
        if any(word in description_lower for word in ["indoors", "inside", "room"]):
            scene_context = "indoor"
        elif any(word in description_lower for word in ["outdoors", "outside", "park", "street"]):
            scene_context = "outdoor"
        else:
            scene_context = "unknown"
        return {
            "content_type": detected_categories[:2] if detected_categories else ["general"],
            "key_objects": key_objects[:5],
            "scene_context": scene_context,
        }

class VideoSummaryGenerator:
    def __init__(self, model):
        from transformers import pipeline
        import torch
        self.summarizer = pipeline(
            "text-generation",
            model=model,
            device=0 if torch.cuda.is_available() else -1,
        )
    def generate_summary(self, frame_analyses):
        all_descriptions = [analysis.get("description", "") for analysis in frame_analyses]
        if not all_descriptions:
            return "Unable to analyze the video content."
        content_analysis = self._analyze_video_content(frame_analyses)
        description_text = self._create_description_text(all_descriptions, content_analysis)
        summary = self._generate_detailed_summary(description_text, content_analysis)
        return summary
    def _analyze_video_content(self, frame_analyses):
        from collections import Counter
        content_types = []
        all_objects = []
        for analysis in frame_analyses:
            content_types.extend(analysis.get("content_type", []))
            all_objects.extend(analysis.get("key_objects", []))
```

 Copyright © Huawei Technologies Co., Ltd.

```
        common_content_types = [item for item, count in Counter(content_types).most_common(2)]
        common_objects = [item for item, count in Counter(all_objects).most_common(5)]
        return {
            "primary_categories": common_content_types,
            "dominant_objects": common_objects,
            "total_frames_analyzed": len(frame_analyses),
        }
    def _create_description_text(self, descriptions, content_analysis):
        from collections import Counter
        desc_counter = Counter(descriptions)
        representative_descs = [desc for desc, count in desc_counter.most_common(3)]
        text = "Video Content Analysis:\n"
        text += f"Primary Categories: {', '.join(content_analysis['primary_categories'])}\n"
        text += f"Key Objects: {', '.join(content_analysis['dominant_objects'][:3])}\n"
        text += "Representative Scenes:\n"
        for i, desc in enumerate(representative_descs, 1):
            text += f"{i}. {desc}\n"
        return text
    def _generate_concise_summary(self, content_analysis):
        import random
        primary_category = content_analysis["primary_categories"][0] if
content_analysis["primary_categories"] else "General content"
        main_objects = ", ".join(content_analysis["dominant_objects"][:2])
        templates = [
            f"This is a {primary_category} video that includes elements like {main_objects}.",
            f"The video presents a {primary_category} scenario, with its focus on {main_objects}.",
            f"The content centers on {primary_category}, highlighting elements such as {main_objects}.",
        ]
        return random.choice(templates)
    def _generate_detailed_summary(self, description_text, content_analysis):
        try:
            summary = self.summarizer(
                "summarize: " + description_text,
                max_length=150,
                min_length=50,
                do_sample=False,
            )[0]["generated_text"]
            return summary
        except Exception:
            return self._generate_concise_summary(content_analysis)

class VideoSummarySystem:
    def __init__(self, model="./t5-small"):
        self.summary_generator = VideoSummaryGenerator(model)
        self.frame_descriptions = []
    def accumulate(self, description: str) -> None:
        import json
        data = json.loads(description)
        self.frame_descriptions.append(data.get("description", data))
    def finish(self) -> str:
        import json
        summary = self.summary_generator.generate_summary(self.frame_descriptions)
        result = {
            "video_summary": summary,
            "frames_analyzed": len(self.frame_descriptions),
            "content_categories": list(
                set(cat for analysis in self.frame_descriptions for cat in analysis.get("content_type", []))
            ),
            "key_objects": list(
                set(obj for analysis in self.frame_descriptions for obj in analysis.get("key_objects", []))
            )[:8],
            "processing_details": {
                "total_frames": len(self.frame_descriptions),
                "successful_analyses": len(self.frame_descriptions),
            },
        }
        return json.dumps(result)
    @property
    def aggregate_state(self) -> typing.Dict[str, typing.Any]:
```

 Copyright © Huawei Technologies Co., Ltd.

```
        return {"frame_descriptions": self.frame_descriptions}
    def merge(self, other_state: typing.Dict[str, typing.Any]) -> None:
        self.frame_descriptions += other_state.get("frame_descriptions")

with create_connect() as con:
    con.set_function_staging_workspace(
        obs_directory_base=os.getenv("obs_directory_base"),
        obs_bucket_name=os.getenv("obs_bucket_name"),
        obs_server=os.getenv("obs_server"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key")
    )
    print("finish set_function_staging_workspace")
    con.delete_function("videocontentanalyzer", database=target_database, if_it_exists=True)
    con.create_scalar_function(
        VideoContentAnalyzer,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./blip-image-captioning-base"],
    )
    con.delete_function("videosummarysystem", database=target_database, if_it_exists=True)
    con.create_agg_function(
        VideoSummarySystem,
        database=target_database,
        packages=["transformers==5.3.0"],
        imports=["./t5-small"],
    )
    # Extract frames from a real video file through AV.
    import av
    import io
    from PIL import Image
    video_path = "test_video.mp4"
    container = av.open(video_path)
    stream = container.streams.video[0]
    frames = []
    frame_count = 0
    max_frames = 10
    for frame in container.decode(stream):
        if frame_count >= max_frames:
            break
        img = frame.to_image()
        buf = io.BytesIO()
        img.save(buf, format="JPEG")
        frames.append((frame_count, buf.getvalue()))
        frame_count += 1
    container.close()
    print(f"Extracted {len(frames)} frames from the video.")
    # Create a table and insert frame data.
    table_name = "video_frames"
    table_schema = ibis.schema({"frame_index": "int32", "frame_data": "binary"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    for frame_index, frame_bytes in frames:
        hex_str = frame_bytes.hex()
        con.sql(f'''
        INSERT INTO "{target_database}"."{table_name}" (frame_index, frame_data) VALUES
            ({frame_index}, '\\x{hex_str}'::bytea)
        ''')
    print("Video frame data has been written.")
    # Load the video frame dataset.
    ds = con.load_dataset("video_frames", database=target_database)
    ds.show(limit=1)
    # Use the scalar UDF to analyze each frame.
```

 Copyright © Huawei Technologies Co., Ltd.

```
udf = con.get_function("videocontentanalyzer", database=target_database)
ds = ds.map_batches(
    fn=udf,
    on=[ds.frame_data, ds.frame_index],
    as_col="descriptions",
    num_cpus=2,
    memory=4096,
    arch="x86",
    model_name="./blip-image-captioning-base",
)
# ds.select_columns([ds.frame_index, ds.descriptions]).show(1)
# Use UDAF to generate a video summary.
udaf = con.get_function("videosummarysystem", database=target_database)
agg_builder = AggregateFnBuilder(
    fn=udaf,
    on=[ds.descriptions],
    as_col="describe",
    num_cpus=1,
    memory=2048,
    arch="x86",
)
summary = ds.aggregate(agg_builder).execute()
for row in summary.itertuples():
    print(row)
```

Helpful Links

- **Session APIs**
- **Dataset APIs**
- **Table APIs**
- **UDF APIs**

2.1.5.5 Aura DataFrame SDK Summary

Aura DataFrame APIs are classified into five types: data processing, data management, input/output, utility, and action.

- Data processing: Offers a variety of transformation operations such as map(), flat_map(), filter(), join(), and groupby() to perform feature engineering and structured data manipulation on datasets.
- Data management: Provides capabilities for dataset create, read, update, delete (CRUD) operations, including insert(), update(), delete(), and index creation/deletion, enabling comprehensive lifecycle management.
- Input/output: Supports seamless integration with popular lakehouse formats like Parquet and Iceberg for efficient data writing.
- Utility: Includes functions like schema(), columns(), and explain_plan() for metadata queries and execution plan analysis, aiding in debugging and optimization.
- Action: Uses methods such as execute(), limit(), and take() to initiate computations and retrieve results, supporting lazy loading and on-demand execution.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-12 API overview

Operation Type	API	Description
Session APIs	load_dataset	Loads a dataset from a specified database.
	set_function_staging_workspace	Configures the temporary workspace required for function deployment by setting necessary credentials and storage details.
	sql	Executes a SQL query and returns the result.
Dataset APIs	columns	Retrieves a collection of all column names in a dataset.
	add_column	Adds columns to a dataset.
	drop_columns	Removes columns from a dataset.
	aggregate	Aggregates the dataset using specified grouping fields and aggregation functions.
	count	Calculates the total number of rows in a dataset.
	execute	Executes expressions.
	filter	Filters out rows that do not meet specified conditions.
	flat_map	Applies a given function to each row of the dataset and flattens the results.
	map_batches	Applies a given function to batch data.
	groupby	Creates a grouped dataset.
	join	Performs a join operation between two datasets.
	map	Applies a given function to each row of the current dataset.
	order_by	Sorts a dataset based on one or more expressions.
	rename_columns	Renames columns in a dataset.
	schema	Returns the schema of a dataset.
	select_columns	Selects specified columns from a dataset.
	show	Displays the data generated by the query results.

 Copyright © Huawei Technologies Co., Ltd.

Operation Type	API	Description
	take	Returns up to limit rows from the dataset, ideal for data inspection and previewing.
	max	Calculates the maximum value of specified columns in a dataset.
	min	Calculates the minimum value of specified columns in a dataset.
	mean	Calculates the mean value of specified columns in a dataset.
	unique	Calculates a set of unique values from a specified column in a dataset.
	limit	Selects rows from the current dataset starting at a given offset.
	take_batch	Returns up to **batch_size** rows from a dataset.
	create_async_job	Restores a job instance based on the job ID.
	explain_plan	Generates the execution plan for a job.
	explain_performance	Executes a query and generates a performance report with detailed metrics from the execution process.
Table APIs	insert	Inserts a dataset into a database table.
	show	Displays the first *N* rows of a table.
	to_dataset	Converts the current table object into a dataset object.
	add_columns	Adds one or more columns to a table.
	drop_columns	Removes one or more columns from a table.
	alter_column_type	Changes the data type of a specified column in a table.
	rename_column	Renames a column in a table.
	delete	Deletes rows from a table that meet the specified condition.
	update	Updates rows in a table that satisfy the given conditions.
	create_table	Creates a table in a specified database.

 Copyright © Huawei Technologies Co., Ltd.

Operation Type	API	Description
	delete_table	Removes a specified dataset from the database.
	describe_table	Describes the tables in a specified database.
	drop_table	Deletes a table from a specified database.
	list_tables	Lists all tables in a specified or the default database.
	open_table	Opens a table in a specified database.
I/O APIs	read_text	Reads text files from a specified OBS path and writes them to a dataset.
	read_json	Reads JSON files from a specified OBS path and writes them to a dataset.
	write_text	Writes text data to a specified OBS path.
	write_json	Writes JSON data to a specified OBS path.
	write_iceberg	Writes data to a specified Iceberg table.
	write_orc	Writes data to a specified ORC table.
	write_parquet	Writes data to a specified Parquet table.
UDF APIs	signature	Returns the function signature and return value type of the UDF from the backend database.
	get	Returns the UDF from the backend database.
	names	Returns the names of all UDFs from the backend database.
	unregister	Deletes a specified UDF from the backend database.
	create_agg_function	Registers a UDAF.
	create_scalar_function	Registers a new scalar UDF.
	create_table_function	Registers a new UDTF.
	delete_function	Deletes a UDF from a specified database.
	describe_function	Describes functions in a specified database.

 Copyright © Huawei Technologies Co., Ltd.

Operation Type	API	Description
	get_function	Retrieves a UDF from a specified or default database.
	list_functions	Lists all UDFs in a specified or default database.
AI APIs	ai_generate	Uses a LLM to generate text and adds it as a new column to a dataset. Ensure that **model_name** has been registered.
	create_model	Creates an abstract local model to be bound to a remote LLM service.
	list_models	Queries all models in a specified or default database.
	drop_model	Deletes the abstract local model with a specified name.
	describe_model	Queries the details of a specified model.

2.1.5.6 Session APIs

load_dataset

Description: Loads a dataset from a specified database.

Syntax:

```
load_dataset(name, *, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	Name of the dataset to load.
database	No	Optional[str]	Name of the database from which to load the dataset. If not specified, the default database is used.

Return type: Dataset.

set_function_staging_workspace

Description: Configures the temporary workspace required for function deployment by setting necessary credentials and storage details.

 Copyright © Huawei Technologies Co., Ltd.

Syntax:

```
set_function_staging_workspace(*, obs_server, obs_bucket_name, obs_directory_base, access_key, secret_key, security_token=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
obs_server	Yes	str	OBS address.
obs_bucket_name	Yes	str	Name of the OBS bucket to use.
obs_directory_base	Yes	str	Base directory path in OBS for storing workspace data.
access_key	Yes	str	Access key for authentication.
secret_key	Yes	str	Secret key for authentication.
security_token	No	Optional[str]	Security token for authentication. If no token is provided, the default value **None** is used.

Return type: None

sql

Description: Executes a SQL query and returns the result.

Syntax:

```
sql(query, **kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
query	Yes	str	SQL query statement to execute.
**kwargs	No	Any	Variable keyword arguments used to pass additional parameters for the query.

Return type: Any. The result of the SQL query is returned.

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.7 Dataset APIs

columns

Description: Retrieves a collection of all column names in a dataset.

Syntax:

```
columns()
```

Input parameters: None.

Return type: tuple[str, ...].

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*',
...     secret_key='*'
...
...
... )
>>> ds = con.load_dataset("demo_table_10", database='my_schema')
>>> ds.columns
('species',
  'island',
  'bill_length_mm',
  'bill_depth_mm',  'flipper_length_mm')
```

add_column

Description: Adds columns to a dataset.

Syntax:

```
add_column(*exprs, **mutations)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
exprs	No	Union[Sequence[Expr], None]	Sequence of named expressions to be added as columns.
mutations	No	Value	Named expressions passed as keyword arguments.

Return type: Updated dataset with new columns.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
```

 Copyright © Huawei Technologies Co., Ltd.

```
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_6", database='my_schema')
>>> ds
| species | bill_length_mm |
| ------- | -------------- |
| string  | float64        |
| ------- | -------------- |
| Adelie  | 40.0           |
| Adelie  | 41.0           |
| Adelie  | 42.0           |
| Adelie  | 43.0           |

#Add a new column from a per-element expression
>>> ds.add_column(new_col=ds.bill_length_mm + 1)
| species | bill_length_mm | new_col  |
| ------- | -------------- | -------- |
| string  | float64        | float64  |
| ------- | -------------- | -------- |
| Adelie  | 40.0           | 41.0     |
| Adelie  | 41.0           | 42.0     |
| Adelie  | 42.0           | 43.0     |
| Adelie  | 43.0           | 44.0     |
```

drop_columns

Description: Removes columns from a dataset.

Syntax:

```
drop_columns(*fields)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
fields	Yes	str	Name of the column to be removed.

Return type: Updated dataset after removing the columns.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_12", database='my_schema')
>>> ds
| species | island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| ------- | --------- | -------------- | ------------- | ----------------- |
| string  | string    | float64        | float64       | int64             |
| ------- | --------- | -------------- | ------------- | ----------------- |
| Adelie  | Torgersen | 40.0           | 22.0          | 180               |
| Adelie  | Torgersen | 41.0           | 19.0          | 183               |
| Adelie  | Torgersen | 42.0           | 20.0          | 190               |
| Adelie  | Torgersen | 43.0           | 23.0          | 185               |

# Drop one or more columns
```

 Copyright © Huawei Technologies Co., Ltd.

```
>>> ds.drop_columns("species")
| island    | bill_length_mm | bill_depth_mm | flipper_length_mm |
| --------- | -------------- | ------------- | ------------------ |
| string    | float64        | float64       | int64              |
| --------- | -------------- | ------------- | ------------------ |
| Torgersen | 40.0           | 22.0          | 180                |
| Torgersen | 41.0           | 19.0          | 183                |
| Torgersen | 42.0           | 20.0          | 190                |
| Torgersen | 43.0           | 23.0          | 185                |
```

aggregate

Description: Aggregates the dataset using specified grouping fields and aggregation functions.

Syntax:

```
aggregate(metrics=(), /, *, by=(), having=(), **kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
metrics	No	Union[Sequence[AggregateFnBuilder], None]	Aggregation expressions, which can be any expressions that generate scalar values, including aggregate functions such as **sum**.
by	No	Union[Sequence[Value], None]	Grouping expressions, typically a collection of column names from a dataset.
having	No	Union[Sequence[BooleanValue], None]	Post-aggregation filter. Its data dimensions must match those of metrics, and its output type must be boolean.
kwargs	No	-	Named aggregation expressions.

Return type: Aggregated dataset.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_13", database='my_schema')
>>> ds
| fruit  | price |
```

 Copyright © Huawei Technologies Co., Ltd.

```
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
>>> ds.aggregate(
...     by=["fruit"],
...     total_cost=ds.price.sum(),
...     avg_cost=ds.price.mean(),
...     having=ds.price.sum() < 0.5,
... ).order_by("fruit")
| fruit  | total_cost | avg_cost |
| ------ | ---------- | -------- |
| string | float64    | float64  |
| ------ | ---------- | -------- |
| banana | 0.25       | 0.25     |
| orange | 0.33       | 0.33     |
```

count

Description: Calculates the total number of rows in a dataset.

Syntax:

```
count()
```

Input parameters: None.

Return type: int.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_11", database='my_schema')
>>> ds
| a     |
| ----- |
| string|
| ----- |
| foo   |
| bar   |
| baz   |
>>> ds.count()
┌──┐
│ 3│
└──┘
```

execute

Description: Executes expressions.

Syntax:

```
execute(*, limit='default', params=None,  block=False, **kwargs)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
limit	No	Union[int, str, None]	The maximum number of rows that can be displayed in the expression execution result.
params	No	Union[Mapping[Value, Any], None]	A mapping of scalar parameter expressions to their values.
block	No	bool	Whether to execute expressions asynchronously. If the value is True, an AsyncJob object is returned.
kwargs	No	-	Keyword arguments.

Return type: Union[pandas.DataFrame, pandas.Series, AsyncJob, Any]

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_8", database='my_schema')
>>> ds.execute()
      a     b
0     1     c
1     1     a
2     2     a
[3 rows x 2 columns]
```

filter

Description: Filters out rows that do not meet specified conditions.

Syntax:

```
filter(expr=None, fn=None, on=None, num_cpus=None, memory=None, arch=None, device=None,
model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
expr	No	Union[list[ir.BooleanValue], Sequence[ir.BooleanValue], IfAnyAll, None]	A valid Python expression string.
on	No	Union[str, list[str]]	Column to which the function is applied.
fn	No	Union[str, Callable, None]	Name of the function that can be called.
num_cpus	No	Union[float, None]	Number of CPUs.
memory	No	Union[int, None]	Memory size, in MB.
arch	No	Union[str, None]	Architecture type. The value can be **x86** or **arm**.
device	No	Union[str, None]	Type of the AI processing unit. The value can be **NPU** or **GPU**.
model	No	Union[str, None]	Model of the AI processing unit, for example, 910B or A100.
num_models	No	Union[float, None]	Number of AI processing units. Configuration is supported only for NPUs and GPUs.
concurrency	No	Union[int, Tuple[int, int], None]	Concurrency during function execution.
fn_constructor_kwargs	No	-	Constructor arguments. This can only be provided when **fn** is a callable class.

Return type: Filtered dataset after removing non-compliant rows.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
```

 Copyright © Huawei Technologies Co., Ltd.

```
>>> ds = con.load_dataset("demo_table_3", database='my_schema')
>>> ds = ds.filter(expr=[ds.a < 2])
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |

# Filtered by User-defined functions
# my_schema.id_filter:inputting an int value, returning false if value is larger than 2.
>>> ds = con.load_dataset("demo_table_2", database='my_schema')
>>> ds = ds.filter(
...     fn='my_schema.id_filter',
...     on='a',
... )
>>> ds
| a     | b     |
| ----- | ----- |
| int64 | string|
| ----- | ----- |
| 1     | c     |
| 1     | a     |
```

flat_map

Description: Applies a given function to each row of the dataset and flattens the results.

Syntax:

```
flat_map(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None, model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
fn	Yes	Union[str, Callable]	Name of the function that can be called, for example, **myschema.myfunction**.
on	Yes	Union[str, list[str],]	Column or set of columns to which the function is applied.
as_col	No	Union[str, list[str], None]	Name of the new column. Its length must match the number of fields returned by **fn**.
num_cpus	No	Union[float, None]	Number of CPUs.
memory	No	Union[int, None]	Memory size, in MB.
arch	No	Union[str, None]	Architecture type. The value can be **x86** or **arm**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
device	No	Union[str, None]	Type of the AI processing unit. The value can be **NPU** or **GPU**.
model	No	Union[str, None]	Model of the AI processing unit, for example, 910B or A100.
num_models	No	Union[float, None]	Number of AI processing units. Configuration is supported only for NPUs and GPUs.
concurrency	No	Union[int, Tuple[int, int], None]	Concurrency during function execution.
fn_constructor_kwargs	No	-	Constructor arguments. This can only be provided when **fn** is a callable class.

Return type:

Dataset: A dataset containing the new column(s) named **as_col**.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_2", database='my_schema')
>>> ds.execute()
      a     b     c
0     1     4     1
[1 rows x 3 columns]

>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
...     on=['a', 'b', 'c'],
...     as_col=['outCol'])
>>> ds
| a     | b     | c     | outCol |
| ----- | ----- | ----- | ------ |
| int64 | int64 | int64 | int64  |
| ----- | ----- | ----- | ------ |
| 1     | 4     | 1     | 1      |
| 1     | 4     | 1     | 2      |
| 1     | 4     | 1     | 3      |
| 1     | 4     | 1     | 4      |

# Access table columns using the ds.attribute (dot notation) style
>>> ds = ds.flat_map(
...     fn='my_schema.py_generate_series',
```

 Copyright © Huawei Technologies Co., Ltd.

```
...     on=[ds.a, ds.b, ds.c],
...     as_col='udtf_col')
>>> ds
| a     | b     | c     | udtf_col |
| ----- | ----- | ----- | -------- |
| int64 | int64 | int64 | int64    |
| ----- | ----- | ----- | -------- |
| 1     | 4     | 1     | 1        |
| 1     | 4     | 1     | 2        |
| 1     | 4     | 1     | 3        |
| 1     | 4     | 1     | 4        |
```

map_batches

Description: Applies a given function to batch data.

Syntax:

```
map_batches(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None,
model=None, num_models=None, concurrency=None, **fn_constructor_kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
fn	Yes	Union[str, Callable]	Name of the function that can be called, for example, **myschema.myfunction**.
on	Yes	Union[str, list[str],]	Column or set of columns to which the function is applied.
as_col	No	Union[str, list[str], None]	Name of the new column. Its length must match the number of fields returned by **fn**.
num_cpus	No	Union[float, None]	Number of CPUs.
memory	No	Union[int, None]	Memory size, in MB.
arch	No	Union[str, None]	Architecture type. The value can be **x86** or **arm**.
device	No	Union[str, None]	Type of the AI processing unit. The value can be **NPU** or **GPU**.
model	No	Union[str, None]	Model of the AI processing unit, for example, 910B or A100.
num_models	No	Union[float, None]	Number of AI processing units. Configuration is supported only for NPUs and GPUs.
concurrency	No	Union[int, Tuple[int, int], None]	Concurrency during function execution.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
fn_constructor_kwargs	No	-	Constructor arguments. This can only be provided when **fn** is a callable class.

Return type:

Dataset: A dataset containing the new column(s) named **as_col**.

groupby

Description: Creates a grouped dataset.

Syntax:

```
groupby(*by, **key_exprs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
by	No	Union[str, Iterable[str], None]	Grouping expression.
key_exprs	No	Union[str, Iterable[str]]	Named grouping expression.

Return type: A grouped dataset.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_14", database='my_schema')
>>> ds
| fruit  | price |
| ------ | ----- |
| string | float |
| ------ | ----- |
| apple  | 0.50  |
| apple  | 0.50  |
| banana | 0.25  |
| orange | 0.33  |
```

 Copyright © Huawei Technologies Co., Ltd.

join

Description: Performs a join operation between two datasets.

Syntax:

```
join(ds, /, on=(), *, join_type='inner', left_suffix='left_', right_suffix='right_')
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
ds	Yes	Dataset	Right-side dataset to join with.
on	No	Union[str, Sequence[Union[str, BooleanColumn, Literal[True], Literal[False], tuple[str, str]]]]	Join condition. For details, see the example.
join_type	No	JoinKind	Join method, such as **inner** or **left**.
left_suffix	No	str	Suffix used to rename overlapping columns from the left table, for example, **left_**.
right_suffix	No	str	Suffix used to rename overlapping columns from the right table, for example, **right_**.

Return type: The joined dataset.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds1 = con.load_dataset("demo_table_15", database='my_schema')
>>> ds2 = con.load_dataset("demo_table_16", database='my_schema')

# Equality inner join on the shared id column. Note the _right suffix added to all overlapping columns from the right # table (in this case only the "id" column).
>>> ds1.join(ds2, "id", join_type="inner")
|   id   |  fruit  |  price  |     color     |
| ------ | ------- | ------- | ------------- |
| int64  | string  | float64 | string        |
| ------ | ------- | ------- | ------------- |
|      2 |   apple |    0.25 | Yellow        |
```

 Copyright © Huawei Technologies Co., Ltd.

```
|      3 |  banana |     0.33 | Orange       |

# Explicit equality join using the default join_type value of "inner". Note join_type there is no _right suffix
added # to the movieId column since this is an inner join and the movieId column is part of the join
condition.
>>> ds1.join(ds2, ds1.id == ds2.id)
|   id   |  fruit  |  price  |     color     |
|--------|---------|---------|---------------|
| int64  | string  | float64 | string        |
|--------|---------|---------|---------------|
|      2 |   apple |    0.25 | Yellow        |
|      3 |  banana |    0.33 | Orange        |
```

map

Description: Applies a given function to each row of the current dataset.

Syntax:

```
map(fn, on, as_col=None, num_cpus=None, memory=None, arch=None, device=None, model=None,
num_models=None, concurrency=None, **fn_constructor_kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
fn	Yes	Union[str, Callable]	Name of the function that can be called, for example, **myschema.myfunction**.
on	Yes	Union[str, list[str],]	Column or set of columns to which the function is applied.
as_col	No	Union[str, list[str], None]	Name of the new column. Its length must match the number of fields returned by **fn**.
num_cpus	No	Union[float, None]	Number of CPUs.
memory	No	Union[int, None]	Memory size, in MB.
arch	No	Union[str, None]	Architecture type. The value can be **x86** or **arm**.
device	No	Union[str, None]	Type of the AI processing unit. The value can be **NPU** or **GPU**.
model	No	Union[str, None]	Model of the AI processing unit, for example, 910B or A100.
num_models	No	Union[float, None]	Number of AI processing units. Configuration is supported only for NPUs and GPUs.
concurrency	No	Union[int, Tuple[int, int], None]	Concurrency during function execution.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
fn_constructor_kwargs	No	-	Constructor arguments. This can only be provided when **fn** is a callable class.

Return type:

Dataset: A dataset with a new column (**as_col**) containing the results of the applied function.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table", database='my_schema')
>>> ds = ds.map(
...     fn='my_schema.multiply_2',
...     on=['col_a'],
...     as_col='col_c'
... )
>>> ds
| col_a | col_b | col_c |
|-------|-------|-------|
| int64 | string | int64 |
|-------|-------|-------|
|     1 | c     | 2     |
|     1 | a     | 2     |
|     2 | a     | 4     |
```

order_by

Description: Sorts a dataset based on one or more expressions.

Syntax:

```
order_by(key, descending=False)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
key	No	Union[str, List[str], None]	Expression used for sorting.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
descending	No	Union[bool, List[bool]]	Whether to sort data in descending order (DESC). The default value is **False**.

Return type: Sorted dataset.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_15", database='my_schema')
>>> ds
| a     | b      | c     |
|-------|--------|-------|
| 3     | a      | 4     |
| 2     | B      | 6     |
| 1     | c      | 5     |
| 3     | D      | 7     |

# Sort by b. Default is ascending. Note how capital letters come before lowercase
>>> ds.order_by("b")
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 2     | B      | 6     |
| 3     | D      | 7     |
| 3     | a      | 4     |
| 1     | c      | 5     |

# Sort in descending order
>>> ds.order_by(ds.b.desc())
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 1     | c      | 5     |
| 3     | a      | 4     |
| 3     | D      | 7     |
| 2     | B      | 6     |

# Sort by multiple columns/expressions
>>> ds.order_by(["a", ds.c.desc()])
| a     | b      | c     |
|-------|--------|-------|
| int64 | string | int64 |
|-------|--------|-------|
| 1     | c      | 5     |
| 2     | B      | 6     |
| 3     | D      | 7     |
| 3     | a      | 4     |
```

 Copyright © Huawei Technologies Co., Ltd.

rename_columns

Description: Renames columns in a dataset.

Syntax:

```
rename_columns(names, /, **substitutions)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
names	Yes	Mapping[str, str]	A mapping (dictionary) specifying renaming rules as {*New column name*: *Old column name*}. Columns not included retain their original names.
substitutions	No	str	Columns to be explicitly renamed, specified as keyword arguments in the format of *New name*=*Old name*.

Return type: A new dataset with renamed columns.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_8", database='my_schema')
>>> ds
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.rename_columns({"rename_col": "a"})
| rename_col | b      |
|------------|--------|
| int64      | string |
|------------|--------|
| 1          | c      |
| 1          | a      |
| 2          | a      |
```

schema

Description: Returns the schema of a dataset.

Syntax:

 Copyright © Huawei Technologies Co., Ltd.

```
schema()
```

Input parameters: None.

Return type: Schema object.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_7", database='my_schema')
>>> ds.schema()
ibis.Schema {
  species            string
  island             string
  bill_length_mm     float64
  bill_depth_mm      float64
  flipper_length_mm  int64
}
```

select_columns

Description: Selects specified columns from a dataset.

Syntax:

```
select_columns(*exprs, **named_exprs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
exprs	Yes	Union[Value, str, Iterable[Union[Value, str]]]	Column expressions, column name strings, or a list of these.
named_exprs	No	Union[Value, str]	Column expressions (named parameters are supported).

Return type: A dataset containing the selected columns.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_5", database='my_schema')
>>> ds.select_columns("species", "bill_length_mm")
| species    | bill_length_mm |
|------------|----------------|
```

 Copyright © Huawei Technologies Co., Ltd.

```
| string     | float64        |
|------------|----------------|
| Adelie     | 40.0           |
| Adelie     | 41.0           |
| Adelie     | 42.0           |
| Adelie     | 43.0           |

# Selection by zero-indexed column position
>>> ds.select_columns(ds[0], ds[4])
| species    | flipper_length_mm |
|------------|-------------------|
| string     | int64             |
|------------|-------------------|
| Adelie     | 180               |
| Adelie     | 183               |
| Adelie     | 190               |
| Adelie     | 185               |
```

show

Description: Displays the data generated by the query results.

Syntax:

```
show(limit=10)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
limit	No	int	Maximum number of output rows. The default value is **10**.

Return type: None.

take

Description: Returns up to limit rows from the dataset, ideal for data inspection and previewing.

Syntax:

```
take(limit)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
limit	Yes	int	Maximum number of rows to be returned.

Return type: list[dict[str, Any]]: A list containing up to limit rows, with each row represented as a dictionary.

 Copyright © Huawei Technologies Co., Ltd.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_9", database='my_schema')
>>> ds.take(3)
[{'a': 1}, {'a': 1}, {'a': 2}]
```

max

Description: Calculates the maximum value of specified columns in a dataset.

Syntax:

```
max(on=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
on	No	Union[str, list[str], None]	Columns for which to calculate the maximum value.

Return type: Dataset.

min

Description: Calculates the minimum value of specified columns in a dataset.

Syntax:

```
min(on=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
on	No	Union[str, list[str], None]	Columns for which to calculate the minimum value.

Return type: Dataset.

 Copyright © Huawei Technologies Co., Ltd.

mean

Description: Calculates the mean value of specified columns in a dataset.

Syntax:

```
mean(on=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
on	No	Union[str, list[str], None]	Columns for which to calculate the mean value.

Return type: Dataset.

unique

Description: Calculates a set of unique values from a specified column in a dataset.

Syntax:

```
unique(col)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
col	Yes	str	Column for which to calculate the unique values.

Return type: list.

limit

Description: Selects rows from the current dataset starting at a given offset.

Syntax:

```
limit(n, /, *, offset=0)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
n	No	Union[int, None]	Number of rows to be included. If **None**, selects all rows from the offset.
offset	No	int	Number of rows to be skipped.

Return type:

Dataset: *N* rows starting from the offset.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*'
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_4", database='my_schema')
>>> ds
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
>>> ds.limit(2)
| a     | b      |
|-------|--------|
| int64 | string |
|-------|--------|
| 1     | c      |
| 1     | a      |
| 2     | a      |
# use None with offset to slice starting from a particular row
>>> ds.limit(None, offset=1)
| a      | b      |
|--------|--------|
| int64  | string |
|--------|--------|
| 1      | a      |
| 2      | a      |
```

take_batch

Description: Returns up to **batch_size** rows from a dataset.

Syntax:

```
take_batch(batch_size=20, *, batch_format='default')
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
batch_size	No	int	Number of rows in a batch. The default value is **20**.
batch_format	No	Optional[str]	Batch format. • **default**: Returns a pyarrow.Table. • **pandas**: Returns a pandas.DataFrame. • **numpy**: Returns a dictionary of NumPy arrays.

Return type: Dataset in the specified format.

create_async_job

Description: Restores a job instance based on the job ID.

Syntax:

```
create_async_job(statement_id)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
statement_id	Yes	str	Unique identifier of a job.

Return value: AsyncJob object.

explain_plan

Description: Generates the execution plan for a job.

Syntax:

```
explain_plan(verbose=True)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
verbose	No	bool	If **True**, provide the detailed description of the query plan. If **False**, provide only a simple execution plan.

Return type: str

explain_performance

Description: Executes a query and generates a performance report with detailed metrics from the execution process.

Syntax:

```
explain_performance()
```

Input parameters: None.

Return type: str, detailed execution plan.

2.1.5.8 Table APIs

insert

Description: Inserts a dataset into a database table.

Syntax:

```
insert(data, overwrite=False)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
data	Yes	dataset.Dataset	Dataset object to be inserted. The value must be of the **aura_frame.multimodal.dataset.Dataset** type.
overwrite	No	bool	Whether to overwrite existing data. The default value is **False**.

Return type: None.

show

Description: Displays the first *N* rows of a table.

 Copyright © Huawei Technologies Co., Ltd.

Syntax:

```
show(limit=10)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
limit	No	int	Maximum number of output rows. The default value is **10**.

Return type: None.

to_dataset

Description: Converts the current table object into a dataset object.

Syntax:

```
to_dataset()
```

Input parameters: None.

Return type: Dataset.

add_columns

Description: Adds one or more columns to a table.

Syntax:

```
add_columns(new_cols)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
new_cols	Yes	Dict[str, str]	Dictionary of columns to be added. The key is the column name (str), and the value is the column type (str).

Return type: None.

drop_columns

Description: Removes one or more columns from a table.

Syntax:

```
drop_columns(*columns)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
*columns	Yes	str	Names of the columns to remove. Accepts multiple string arguments.

Return type: None.

alter_column_type

Description: Changes the data type of a specified column in a table.

Syntax:

```
alter_column_type(column_name, new_type)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
column_name	Yes	str	Name of the column whose type is to be changed.
new_type	Yes	str	New data type.

Return type: None.

rename_column

Description: Renames a column in a table.

Syntax:

```
rename_column(column_name, new_name)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
column_name	Yes	str	Current column name.
new_name	Yes	str	New column name.

Return type: None.

 Copyright © Huawei Technologies Co., Ltd.

delete

Description: Deletes rows from a table that meet the specified condition.

Syntax:

```
delete(where)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
where	Yes	str	An SQL WHERE clause, used to specify deletion conditions.

Return type: None.

update

Description: Updates rows in a table that satisfy the given conditions.

Syntax:

```
update(values_expr, where)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
values_expr	Yes	Dict[str, ValueExpression]	A dictionary where keys are column names (str) and values are expressions (ValueExpression) used to update the column.
where	No	Union[ir.BooleanValue, list[ir.BooleanValue], IfAnyAll, None]	Condition used to filter the rows to be updated. If **None**, all rows are updated.

Return type: None.

create_table

Description: Creates a table in a specified database.

Syntax:

```
create_table(name, /, *, schema, table_format, database=None, temp=False, external=False, overwrite=False, if_not_exists=False, partition_by=None, table_properties=None, location=None)
```

 Copyright © Huawei Technologies Co., Ltd.

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	Name of the table to create.
schema	Yes	ibis.Schema	Definition of the table.
table_format	Yes	str	Storage format of the table.
database	No	Optional[str]	Name of the database where the table is to be created. If it is not specified, the default database is used.
temp	No	bool	If it is set to **True**, a temporary table is created.
external	No	bool	If it is set to **True**, an external table is created.
overwrite	No	bool	If it is set to **True**, the existing table is overwritten.
if_not_exists	No	bool	If it is set to **True**, the table is created only when it does not exist.
partition_by	No	Optional[ibis.Schema]	Schema used to partition the table.
table_properties	No	Optional[dict]	Other properties of the table.
location	No	Optional[str]	Location where the table data is stored.

Return type: Dataset.

delete_table

Description: Removes a specified dataset from the database.

Syntax:

```
delete_table(name, /, *, database=None, if_it_exists=False)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	Name of the table to be deleted.
database	No	Optional[str]	Name of the database from which the table is to be deleted. If not specified, the default database is used.
if_it_exists	No	bool	If this parameter is set to True, the table is deleted only when it exists. This prevents errors caused by the table's absence.

Return type: str.

describe_table

Description: Describes the tables in a specified database.

Syntax:

```
describe_table(name, /, *, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	The function name.
database	No	Optional[str]	Name of the database where the table is located.

Return type: str. Table metadata.

drop_table

Description: Deletes a table from a specified database.

Syntax:

```
drop_table(name, *, database=None, force=False)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	Name of the table to be deleted.
database	No	Optional[str]	Name of the database from which the table is to be deleted. If not specified, the default database is used.
force	No	bool	If this parameter is set to **True**, the table is forcibly deleted even if it contains data. The default value is **False**.

Return type: None.

list_tables

Description: Lists all tables in a specified or default database.

Syntax:

```
list_tables(database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
database	No	Optional[str]	Name of the database whose tables are to be listed. If it is **None**, the default database is used.

Return type: list[tuple], a list of tuples, where each tuple contains table names.

open_table

Description: Opens a table in a specified database.

Syntax:

```
open_table(name, *, database=None)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
name	Yes	str	Name of the table to be opened.
database	No	Optional[str]	Name of the database that contains the table to be opened. If not specified, the default database is used.

Return type: Table.

2.1.5.9 I/O APIs

write_iceberg

Description: Writes data to a specified Iceberg table.

Syntax:

```
write_iceberg(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None, explain = None, verbose = True)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
target_name	Yes	str	Name of the target table that data is written to.
database	Yes	str	Database that the table belongs to.
create_if_not_exists	No	bool	If this parameter is set to **True**, the table will be automatically created if it does not exist.
overwrite	No	bool	If this parameter is set to **True**, the existing data in the table will be overwritten.
is_external	No	bool	If this parameter is set to **True**, the table is treated as an external table.
schema	No	Any type	(Optional) Schema definition of the table.
explain	No	String	The value can be set to **plan** or **performance**. **plan** prints only the execution plan, while **performance** prints the execution details.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
verbose	No	bool	If this parameter is set to **True** and **explain** set to **plan**, additional detailed information will be printed.

Return type: str

write_orc

Description: Writes data to a specified ORC table.

Syntax:

```
write_orc(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None, explain = None, verbose = True)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
target_name	Yes	str	Name of the target table that data is written to.
database	Yes	str	Database that the table belongs to.
create_if_not_exists	No	bool	If this parameter is set to **True**, the table will be automatically created if it does not exist.
overwrite	No	bool	If this parameter is set to **True**, the existing data in the table will be overwritten.
is_external	No	bool	If this parameter is set to **True**, the table is treated as an external table.
schema	No	Any type	(Optional) Schema definition of the table.
explain	No	String	The value can be set to **plan** or **performance**. **plan** prints only the execution plan, while **performance** prints the execution details.
verbose	No	bool	If this parameter is set to **True** and **explain** set to **plan**, additional detailed information will be printed.

 Copyright © Huawei Technologies Co., Ltd.

Return type: str

write_parquet

Description: Writes data to a specified Parquet table.

Syntax:

```
write_parquet(target_name, database, create_if_not_exists=False, overwrite=False, is_external=False, schema=None, explain = None, verbose = True)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
target_name	Yes	str	Name of the target table that data is written to.
database	Yes	str	Database that the table belongs to.
create_if_not_exists	No	bool	If this parameter is set to **True**, the table will be automatically created if it does not exist.
overwrite	No	bool	If this parameter is set to **True**, the existing data in the table will be overwritten.
is_external	No	bool	If this parameter is set to **True**, the table is treated as an external table.
schema	No	Any type	(Optional) Schema definition of the table.
explain	No	String	The value can be set to **plan** or **performance**. **plan** prints only the execution plan, while **performance** prints the execution details.
verbose	No	bool	If this parameter is set to **True** and **explain** set to **plan**, additional detailed information will be printed.

Return type: str

write_text

Description: Writes text data to a specified OBS path.

Syntax:

```
write_text(path, field_delim, max_rows_per_file=None)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
path	Yes	str	OBS path for storing text files.
field_delim	Yes	str	Field delimiter for rows in the TEXT data file. The value cannot be any character in **\\.abcdefghijklmnopqr-stuvwxyz0123456789**.
max_rows_per_file	No	int	Maximum number of rows that can be written to a file

Return type: None.

write_json

Description: Writes JSON data to a specified OBS path.

Syntax:

```
write_json(path, max_rows_per_file=None, ignore_null_fields=False, nested_separator=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
path	Yes	str	OBS path for storing JSON data.
max_rows_per_file	No	int	Maximum number of rows that can be written to a file
ignore_null_fields	No	bool	Whether to ignore the JSON keys with **NULL** values when writing JSON objects.
nested_separator	No	str	Separator for concatenating nested JSON keys. The value cannot be a number sign (#). For example, if this parameter is set to an underscore (_), **{"a":{"b":1}}** will be mapped to column **a_b** with value **1**. By default, nested JSON read and write are not supported.

Return type: None.

 Copyright © Huawei Technologies Co., Ltd.

read_text

Description: Reads TEXT files from a specified OBS path and writes them to a dataset with a specified schema.

Syntax:

```
read_text(path, schema, field_delim='\t', drop_empty_lines=True, fill_missing_fields=False,
ignore_extra_data=False)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Data Type	Description
path	Yes	str	Directory for storing text files in OBS.
schema	Yes	dict	Column information for reading the text file.
field_delim	No	str	Field delimiter for rows in the TEXT data file. The value cannot be any character in **\\.abcdefghijklmnopqr-stuvwxyz0123456789**.
drop_empty_lines	No	bool	Whether to discard empty lines or single-line \n, \r, and \t in the text.
fill_missing_fields	No	bool	Whether to report an error when the number of columns after splitting by **field_delim** is less than the number of columns.
ignore_extra_data	No	bool	Whether to report an error when the number of columns after splitting by **field_delim** exceeds the number of columns.

Return type: Dataset, which is a collection of text data.

read_json

Description: Reads the JSON value of a specified JSON key from a specified OBS path based on the schema.

Syntax:

```
read_json(path, schema, skip_empty_files=False, nested_separator=None)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Data Type	Description
path	Yes	str	Directory for storing JSON files in OBS.
schema	Yes	dict	Column information for reading the specified key.
skip_empty_files	No	bool	Whether to skip the JSON content that is **{}**.
nested_separator	No	str	Separator for concatenating nested JSON keys. The value cannot be a number sign (#). For example, if this parameter is set to an underscore (_), **{"a":{"b":1}}** will be mapped to column **a_b** with value **1**. By default, nested JSON read and write are not supported.

Return type: Dataset, which is a collection of JSON objects.

2.1.5.10 UDF APIs

Explicit Registration Syntax for UDFs

Explicit registration refers to the manual embedding of registration logic directly into Python code, typically achieved through methods like **backend.register** or **register_from_file**, where calling these methods triggers the registration process. This method relies on having access to a backend session object before proceeding.

Scenarios recommending explicit registration: If you wish to explicitly control registration timing, allow intrusive addition of registration logic, or require separation of UDF registration and usage under the same backend connection.

A common scenario involves one team handling UDF registration while multiple teams utilize these functions, with no shared Python scripts between them.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-13 Explicit registration syntax for UDFs

UDF Type	UDF Type (Secondary)	Registration Type (Tertiary)	Code Entry	Reference
UDF, UDAF, UDTF	Python	Direct registration	backend.[udf | udaf | udtf].python.register(<*Registration function*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF Registration Parameters**
		File-based registration	backend.[udf | udaf | udtf].python.register_from_file(<*File path*>, <*Function name*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF registration parameters**
	Builtin	Direct registration	backend.[udf | udaf | udtf].builtin.register(<*Registration function*>, <*Registration parameters*>)	**Built-in UDF Registration Parameters**
		File-based registration	backend.[udf | udaf | udtf].builtin.register_from_file(<*File path*>, <*Function name*>, <*Registration parameters*>)	**Built-in UDF Registration Parameters**
	PyArrow	Direct registration	backend.udf.pyarrow.register(<*Registration function*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF Registration Parameters**
		File-based registration	backend.udf.pyarrow.register_from_file(<*File path*>, <*Function name*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF Registration Parameters**
	Pandas	Direct registration	backend.udf.pandas.register(<*Registration function*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF Registration Parameters**
		File-based registration	backend.udf.pandas.register_from_file(<*File path*>, <*Function name*>, <*Registration parameters*>)	**Python/ PyArrow/Pandas UDF Registration Parameters**

Implicit Registration Syntax for UDFs

Implicit registration simplifies this process by using the Python runtime to automatically detect and register UDFs. Instead of embedding registration logic directly into the code, you can decorate your Python functions with @ decorators.

 Copyright © Huawei Technologies Co., Ltd.

Once decorated, these functions can be referenced in DataFrames using their original identifiers, seamlessly completing the registration. With implicit registration, there is no need to obtain a backend session object when applying the @ decorator. The backend session is only required later when working with Ibis DataFrames.

This approach is particularly useful when you aim to register UDFs without intrusive modifications to your code and do not require separation between the registration and usage of UDFs under the same Backend connection.

A common use case involves scenarios where your Python script includes both the registration and application of UDFs in a single workflow.

Table 2-14 Implicit registration syntax for UDFs

UDF Type	UDF Type (Secondary)	Code Entry	Reference
UDF, UDAF, UDTF	Python	@aura.[udf | udaf | udtf].python(<*Registration parameters*>)	**Python/PyArrow/Pandas UDF Registration Parameters**
	Builtin	@aura.[udf | udaf | udtf].builtin(<*Registration parameters*>)	**Built-in UDF Registration Parameters**
	PyArrow	@aura.udf.pyarrow(<*Registration parameters*>)	**Python/PyArrow/Pandas UDF Registration Parameters**
	Pandas	@aura.udf.pandas(<*Registration parameters*>)	**Python/PyArrow/Pandas UDF Registration Parameters**

For implicit registration, the timing of the actual registration action differs depending on the DataFrame operation mode (Lazy or Eager).

As mentioned earlier in the **Ibis official documentation**, DataFrame operations are divided into Eager and Lazy modes, controlled by the **ibis.options.interactive** configuration item. By default, this is set to **false**, meaning all DataFrames operate in Lazy mode by default. For these two DataFrame execution modes, the timing of UDF registration varies as described below:

 Copyright © Huawei Technologies Co., Ltd.

Table 2-15 DataFrame execution modes

ibis.options.interactive	DataFrame Execution Mode	UDF Registration Time	UDF Usage Time
False	Lazy	When the entire DataFrame calls the **execute** method	When the entire DataFrame calls the **execute** method
True	Eager	First use in DataFrame	Every use in DataFrame

Python/PyArrow/Pandas UDF Registration Parameters

Registering a Python/PyArrow/Pandas UDF involves registering an original Python function or class into the database.

Whether it is explicit or implicit registration, whether it is scalar UDF, UDAF, or UDTF, for registering Python, PyArrow, or Pandas type UDFs, you can currently pass in the following parameters:

Table 2-16 Python/PyArrow/Pandas UDF Registration Parameters

Registration Parameter	Description	Type	Default Value
name	Specifies the actual storage name of the UDF in the database.	str | None	None
database	Specifies the LakeFormation database where the UDF resides.	str | None	None
fn	Specifies the original Python function of the UDF.	Callable	None
signature	Specifies the UDF function signature and return value type.	aura_frame.ibis.common.annotations.Signature | None	None
replace (currently unavailable)	Specifies whether the UDF can be replaced.	bool	False

 Copyright © Huawei Technologies Co., Ltd.

Registration Parameter	Description	Type	Default Value
temporary (currently unavailable)	Specifies whether the UDF has a session-level lifecycle.	bool	False
if_not_exist (currently unavailable)	Specifies whether to skip errors for existing UDFs.	bool	False
strict	Specifies whether the UDF automatically filters NULL values.	bool	True
volatility	Specifies the stability of the UDF.	VolatilityType.VOLATILE | VolatilityType.STABLE | VolatilityType.IMMUTABLE	VolatilityType.VOLATILE
runtime_version (currently unavailable)	Specifies the Python version for executing the UDF.	str	sys.version_info
runtime_env	Specifies the Python module on which the UDF depends and the installation mode (venv), which will be used to replace **packages**.	Union[str, Mapping[str, Any]] | None	None
imports	Specifies the external code files on which the UDF depends.	List[str]	None
packages	Specifies the Python modules on which the UDF depends.	List[Union[str, module]] | None	None
register_type	Specifies the registration form of the UDF.	RegisterType.INLINE | RegisterType.STAGED	RegisterType.INLINE
comment	Specifies user comments for the UDF.	str | None	None

 Copyright © Huawei Technologies Co., Ltd.

Python/PyArrow/Pandas UDF Registration Parameter Notes

- For the **imports** parameter, only file paths in the same directory or subdirectories as the **.py** file containing the current Python function or class are allowed.
- For the **fn** parameter, if **fn** is not in the **.py** file where the UDF is being registered, then the file path defining **fn** must also be added to the **imports** parameter, for example:

```
from process import outer
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# Register a UDAF.
udf = con.udaf.python.register(
    outer(), #fn introduced externally
    imports=["process.py"] #Add the file path for fn
)
```

- The **signature** parameter is currently optional for you to provide. If you specify it, your input takes precedence over automatic inference. When no signature is provided, the system defaults to inferring the parameter and return value types automatically. For more details, refer to **Type Inference of the signature Parameter**.
  - When registering a PyArrow UDF, whether you provide a **signature** parameter, it always relies on PyArrowVector. For example:

```
import aura_frame as aura
import pyarrow as pa
import pyarrow.compute as pc
from aura_frame.multimodal import ai_lake

# === With signature: Requires dependency on PyArrowVector. ===
def calculate_sum(
    prices: pa.ChunkedArray,
    quantities: pa.ChunkedArray,
) -> pa.ChunkedArray:
    return pc.multiply(prices, quantities)

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# Register a UDF.
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )
```

 Copyright © Huawei Technologies Co., Ltd.

```
# === Without signature: Also requires dependency on PyArrowVector. ===
def calculate_sum(
    prices: aura.PyarrowVector[float],
    quantities: aura.PyarrowVector[int],
) -> aura.PyarrowVector[float]:
    return aura.PyarrowVector[float](pc.multiply(prices, quantities))

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# Register a UDF.
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )
```

- When registering a Pandas UDF, whether you provide a **signature** parameter, it always relies on PandasVector. For example:

```
import aura_frame as aura
import pandas as pd
from aura_frame.multimodal import ai_lake

# === With signature: Requires dependency on PandasVector. ===
def calculate_sum(
    prices: pd.Series,
    quantities: pd.Series,
) -> pd.Series:
    return pd.Series(prices * quantities, dtype=pd.Float64Dtype())

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
        use_single_cn_mode=True
)

# Register a UDF.
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )

# === Without signature: Also requires dependency on PandasVector. ===
def calculate_sum(
    prices: aura.PandasVector[float],
    quantities: aura.PandasVector[int],
) -> aura.PandasVector[float]:
    return aura.PandasVector[float](prices * quantities)

con = ai_lake.connect(
        aura_endpoint=os.getenv("aura_endpoint"),
        aura_endpoint_name=os.getenv("aura_endpoint_name"),
        aura_workspace_id=os.getenv("aura_workspace_id"),
        lf_catalog_name=os.getenv("lf_catalog_name"),
        access_key=os.getenv("access_key"),
        secret_key=os.getenv("secret_key"),
        default_database=target_database,
```

 Copyright © Huawei Technologies Co., Ltd.

```
        use_single_cn_mode=True

)

# Register a UDF.
con.create_scalar_function(
        calculate_sum,
        database=target_database
    )
```

- For the **volatility** parameter, the meanings of the three enumeration types are:
  - **VolatilityType.VOLATILE**: Function results may change at any time.
  - **VolatilityType.STABLE**: Function results remain unchanged for a fixed input during a single scan.
  - **VolatilityType.IMMUTABLE**: The function always produces the same result for identical inputs.

  The **volatility** parameter does not impact the execution of function pushdown. Python UDFs can be pushed down to DNs regardless of whether they are classified as **IMMUTABLE**, **STABLE**, or **VOLATILE**.
- If you do not specify the **packages** parameter:
  - For PyArrow UDFs, the PyArrow version installed in the backend environment is automatically used.
  - For Pandas UDFs, the Pandas version installed in the backend environment is automatically used.

  To ensure that third-party dependencies for Python are installed in an isolated environment, the packages specified in the **packages** parameter are passed to RuntimeEnv for installation. The default installation mode is **venv**.

Builtin UDF Registration Parameters

For Builtin UDFs, registering them simply means obtaining a handle for existing database functions—no actual registration occurs.

Whether it is explicit or implicit registration, whether it is scalar UDF, UDAF, or UDTF, for registering Builtin type UDFs, you can currently pass in the following parameters:

Table 2-17 Builtin UDF Registration Parameters

Registration Parameter	Description	Type	Default Value
name	Specifies the actual storage name of the UDF in the database.	str | None	None
database	Specifies the LakeFormation database where the UDF resides.	str | None	None

 Copyright © Huawei Technologies Co., Ltd.

Registration Parameter	Description	Type	Default Value
fn	Specifies the original Python function of the UDF.	Callable	None
signature	Specifies the UDF function signature and return value type.	ibis.common.annotations.Signature | None	None

Builtin UDF Registration Parameter Notes

The **signature** parameter is currently optional for you to provide. If you specify it, your input takes precedence over automatic inference. When no signature is provided, the system defaults to inferring the parameter and return value types automatically. For more details, refer to **Type Inference of the signature Parameter**.

Type Inference of the signature Parameter

For the **signature** parameter, you may choose to provide the parameter/return value types or omit them entirely.

- If you supply the **signature** parameter, there is no requirement for the original Python function to utilize type hinting syntax. This enables immediate operational registration of the UDF.
- Conversely, if the signature parameter is not provided, you are advised to use type hinting syntax within the original Python function, though this precludes immediate operational registration of the UDF.

A comparison of these approaches is summarized below.

Table 2-18 signature parameters

signature Parameter	Description	Require Original Python Function with Type Hint Syntax	Support Immediate REPL Operation
User omits passing value	Auto-deduction (recommended)	No, but recommended	No
User specifies value	Specified value	No	Yes

 Copyright © Huawei Technologies Co., Ltd.

Note: Immediate operation pertains to the read-evaluate-print loop (REPL), commonly encountered in Python's interactive terminal environment.

Introduced in Python 3.5 via PEP 484, type hinting syntax involves appending a colon (:) followed by the type after the parameter name and indicating the return type post the parameter list using an arrow (->), exemplified as follows:

```
def greet(name: str) -> str:
    return f"Hello, {name}"
from typing import List, Dict, Optional

def process_data(data: List[int]) -> Dict[str, Optional[int]]:
    return {"max": max(data) if data else None}
```

For Python/PyArrow/Pandas UDFs, strict typing is mandated upon registration, requiring explicit specification of all parameter and return value types. If you fail to define these through the original Python function's type hinting, you must actively use the **signature** parameter to designate the **Ibis DataType**.

For Builtin UDFs, strict typing is not required upon registration (as the UDF has already been registered in the database). If you cannot specify the type annotations of the original Python function, you are advised to include only the parameter names without their types. If you later use the return value of the Builtin UDF (excluding Top SELECT UDF), then the function's return type needs to be specified, and when necessary, you should actively use the **signature** parameter to define the **Ibis DataType**. If not needed (for Top SELECT UDF), you may omit writing the function's return type.

Regarding cases where you do not provide the **signature** parameter and rely on auto-deduction, the following summary applies:

Table 2-19 Auto-deduction of signature

Registered UDF Type	Data Type	Return Type
Python/ Pyarrow/ Pandas UDF	Requires type hinting syntax for specification.	Requires type hinting syntax for specification.
Builtin UDF	Allows writing just parameter names without types.	Requires type hinting syntax when utilizing return values subsequently. Otherwise, not mandatory.

For cases where you do not pass in the **signature** parameter and it is inferred automatically, the underlying implementation principle is **inspect.signature**. Currently, the system accepts the following parameter/return value types from you:

Table 2-20 Accepted parameter/return value types

Python	Ibis DataType	Aura
DataType	DataType	-

 Copyright © Huawei Technologies Co., Ltd.

Python	Ibis DataType	Aura
type(None)	null	NULL
bool	Boolean	BOOLEAN
bytes	Binary	BYTEA
str	String	TEXT
numbers.Integral	Int64	BIGINT
numbers.Real	Float64	DOUBLE PRECISION
decimal.Decimal	Decimal	DECIMAL
datetime.datetime	Timestamp	TIMESTAMP/ TIMESTAMPTZ
datetime.date	Date	TIMESTAMP
datetime.time	Time	TIME
datetime.timedelta	Interval	INTERVAL
uuid.UUID	UUID	UUID
class	Struct	STRUCT
typing.Sequence, typing.Array	Array	ARRAY
typing.Mapping, typing.Map	Map	HSTORE
aura_frame.PyarrowVector[T]	T	T
aura_frame.PandasVector[T]	T	T

Notes:

- Python built-in int is a subclass of **numbers.Integral**.
- Python built-in float is a subclass of **numbers.Real**.

Python types not listed in the table above are currently not supported for automatic conversion.

For parameters/return values where you do not pass the **signature** parameter and also do not use Python type hinting syntax, automatic inference handles them as follows:

 Copyright © Huawei Technologies Co., Ltd.

Table 2-21 Special parameter type handling

Parameter Type	Generated Matching Pattern	Pattern Effectiveness
POSITIONAL_ONLY, KEYWORD_ONLY, POSITIONAL_OR_KEYWORD	ValueOf(None)	Exempts from **__signature__.validate**.
VAR_POSITIONAL	TupleOf(pattern=pattern)	Executes pattern in a for-loop.
VAR_KEYWORD	DictOf(key_pattern=InstanceOf(str), value_pattern=pattern)	Executes pattern in a for-loop.
Return	ValueOf(Unknown)	Provides UnknownScalar and UnknownColumn as UDF return values passed upward.

inspect.signature classifies parameter types (Parameter.kind) as follows:

Table 2-22 inspect.signature parameter types

Parameter Type	Description	Sample Code	Parameters Meeting Conditions
POSITIONAL_ONLY	Position-only parameter.	def func(a, /, b): pass	a
KEYWORD_ONLY	Keyword-only parameter.	def func(a, *, b): pass	b
POSITIONAL_OR_KEYWORD	Positional or keyword parameter.	def func(a, b): pass	a, b
VAR_POSITIONAL	Variable positional parameter.	def func(*args): pass	args
VAR_KEYWORD	Variable keyword parameter.	def func(**kwargs): pass	kwargs

Direct Operation Syntax for UDFs

In scenarios where registration and usage are separate, the direct operation syntax for scalar UDFs, UDAFs, and UDTFs is provided for you. You only need to know the

 Copyright © Huawei Technologies Co., Ltd.

UDF name (**name**) and the database name (**database**) where the UDF resides to directly use the UDF. The following operations rely on the UDF attribute of the backend session object.

signature

Description: Returns the function signature and return value type of the UDF from the backend database.

Syntax:

```
signature(name, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	Yes	str	The UDF name.
database	No	str	Name of the database to which the UDF belongs.

Return type: aura_frame.ibis.common.annotations.Signature - Registered UDF's signature and return type.

get

Description: Returns the UDF from the backend database.

Syntax:

```
get(name, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	Yes	str	The UDF name.
database	No	str	Name of the database to which the UDF belongs.

Return type: Callable[..., ibis.expr.types.Value] - Registered UDFs.

names

Description: Returns the names of all UDFs from the backend database.

Syntax:

```
names(database=None)
```

 Copyright © Huawei Technologies Co., Ltd.

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
database	No	str	Name of the database to which the UDF belongs.

Return type: List[str] - Names of all registered UDFs.

unregister

Description: Deletes a specified UDF from the backend database.

Syntax:

```
unregister(name, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	Yes	str	The UDF name.
database	No	str	Name of the database to which the UDF belongs.

Return type: None.

UDF WITH ARGUMENTS Syntax

Whether you use **explicit registration syntax**, **implicit registration syntax** to return a UDF operator, or **direct operation syntax** to access a registered UDF, all currently support passing arguments via the **with_arguments** method. These arguments fall into two categories:

- Special-purpose parameters: These parameters configure runtime resources, concurrency, and execution time limits for the UDF. Examples include **concurrency**, **timeout**, **dpu**, and **apu**. For details, see **Table 2-30**. All types of UDFs currently allow you to pass these configuration parameters using **with_arguments**.
- General parameters: These parameters initialize the state of the UDF during its setup phase. They can only be passed as scalar values if you have defined optional parameters in the __**init**__ method of class UDF, class UDTF, or UDAF. This allows for one-time initialization of internal states, which can then be reused multiple times.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-23 Applicability of WITH ARGUMENTS syntax

Parameter	Applicable UDF Type	Description
Special parameters (e.g., concurrency, timeout, dpu, apu)	All UDF types, including scalar UDF, class UDF, vectorized UDF, UDTF, UDAF	Configures runtime resources, concurrency, and execution time limits for the UDF.
General parameters defined by the Python class's **__init__** method	Class UDF types, including class UDF, class UDTF, UDAF	Sets up initial state for the UDF, suitable for cases involving internal caching, initialization parameters, or model objects.

All parameter values passed through the **with_arguments** method are scalar values. These values are collectively passed as a **kwargs dictionary in Python.

create_agg_function

Description: Registers a UDAF.

Syntax:

```
create_agg_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
udf	Yes	Callable	UDAF to be registered.
input_type	No	InputType	Input type of the function. The value can be **PYTHON**, **BUILTIN**, **PYARROW**, or **PANDAS**.
database	Yes	str	Name of the database to which the function belongs.
name	No	str	Name of the function. The default value is **None**.
signature	No	Signature	(Optional) Signature definition of a function.
imports	No	[Union[str, Iterable[str]]]	Import modules required by the function, which can be a single string or a list of strings.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
packages	No	[Union[str, Iterable[str]]]	Dependency packages required by the function, which can be a single package name or a list of package names.
runtime_env	No	[Union[str, Mapping[str, Any]]]	Dependency packages and installation methods required by the function, which can be a JSON string or a Python dictionary.
comment	No	str	(Optional) Comments of the function.
**kwargs	No	-	Other optional keyword arguments.

Return type:

Callable: Registered function.

create_scalar_function

Description: Registers a new scalar UDF.

Syntax:

```
create_scalar_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
udf	Yes	Callable	Scalar UDF to be registered.
input_type	No	InputType	Input type of the function. The value can be **PYTHON**, **BUILTIN**, **PYARROW**, or **PANDAS**.
database	Yes	str	Name of the database to which the function belongs.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	No	str	Name of the function. The default value is **None**.
imports	No	[Union[str, Iterable[str]]]	Import modules required by the function, which can be a single string or a list of strings.
packages	No	[Union[str, Iterable[str]]]	Dependency packages required by the function, which can be a single package name or a list of package names.
runtime_env	No	[Union[str, Mapping[str, Any]]]	Dependency packages and installation methods required by the function, which can be a JSON string or a Python dictionary.
comment	No	Optional[str]	(Optional) Comments of the function.
**kwargs	No	-	Other optional keyword arguments.

Return type: Callable

create_table_function

Description: Registers a UDTF.

Syntax:

```
create_table_function(udf, /, input_type=InputType.PYTHON, *, database, name=None, signature=None, imports=None, packages=None, runtime_env=None, comment=None, **kwargs)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
udf	Yes	Callable	UDTF to be registered.
input_type	No	InputType	Input type of the function. The value can be **PYTHON** or **BUILTIN**.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
database	Yes	str	Name of the database to which the function belongs.
name	No	str	Name of the function. The default value is **None**.
signature	No	Signature	(Optional) Signature definition of a function.
imports	No	[Union[str, Iterable[str]]]	Import modules required by the function, which can be a single string or a list of strings.
packages	No	[Union[str, Iterable[str]]]	Dependency packages required by the function, which can be a single package name or a list of package names.
runtime_env	No	[Union[str, Mapping[str, Any]]]	Dependency packages and installation methods required by the function, which can be a JSON string or a Python dictionary.
comment	No	str	(Optional) Comments of the function.
**kwargs	No	-	Other optional keyword arguments.

Return type:

Callable: Registered function.

delete_function

Description: Deletes a UDF from a specified database.

Syntax:

```
delete_function(name, /, *, database=None, if_it_exists=False)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	Yes	str	Name of the function to be deleted.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
database	No	str	Name of the database from which the function is to be deleted. If it is not specified, the default database is used.
if_it_exists	No	bool	If it is set to **True**, the deletion operation is performed only when the function exists. This prevents errors caused by the absence of the function.

Return type: str

describe_function

Description: Describes the functions in a specified database.

Syntax:

```
describe_function(name, /, *, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
name	Yes	str	The function name.
database	No	str	Name of the database where the function is located.

Return type: str, metadata information about the function.

get_function

Description: Retrieves a UDF from a specified or default database.

Syntax:

```
get_function(name, *, database=None)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
func_name	Yes	str	Name of the user-defined function to be retrieved.
database	No	Optional[str]	Name of the database from which to retrieve the function. If not specified, the default database is used.

Return type: Callable. A callable function is returned.

list_functions

Description: Lists all UDFs in a specified or default database.

Syntax:

```
list_functions(database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
database	No	str	Name of the database from which to list the function. If not specified, the default database is used.

Return type: pandas.DataFrame.

2.1.5.11 AI APIs

ai_generate

Description: Uses a LLM to generate text and adds it as a new column to a dataset. Ensure that **model_name** has been registered.

Syntax:

```
ai_generate(prompt, on, as_col, model_name, model_parameters=None)
```

Input parameters:

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Parameter Type	Description
prompt	Yes	str	Prompt template for text generation. The content of the input column will be appended to this prompt.
on	Yes	ir.Column	Dataset column that contains the input data to be processed.
as_col	Yes	str	Name of the new column where the generated text will be stored.
model_name	Yes	str	Name of the LLM used for text generation.
model_parameters	No	str	Additional parameters of the LLM, provided as a JSON string. The default value is **None**.

Return type: A dataset containing the new column.

Example:

```
>>> from aura_frame.multimodal import ai_lake
>>> con = ai_lake.connect(
...     aura_endpoint='*',
...     access_key='*',
...     secret_key='*'
...     ...
... )
>>> ds = con.load_dataset("demo_table_text", database='my_schema')
>>> ds
| id | text_content          |
| -- | --------------------- |
| int| string                |
| -- | --------------------- |
| 1  | You're a good man.    |
| 2  | You're a bad man.     |

# Use AI and model parameters to generate text.
>>> ds.ai_generate(
...     prompt="Is this a positive review? Just reply with 'Yes' or 'No'.",
...     on=ds.text_content,
...     as_col="judgement",
...     model_name="gpt-4",
...     model_parameters='{"temperature": 0.7, "max_tokens": 100}'
... )
| id | text_content          | judgement        |
| -- | --------------------- | ---------------- |
| int| string                | string           |
| -- | --------------------- | ---------------- |
| 1  | You're a good man.    | Yes              |
| 2  | You're a bad man.     | No               |
```

create_model

Description: Creates an abstract local model to be bound to a remote LLM service.

 Copyright © Huawei Technologies Co., Ltd.

Syntax:

```
create_model(model_name, base_url, api_key, remote_model_name, description=None, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
model_name	Yes	str	Name of the abstract local model.
base_url	Yes	str	Base URL of the remote LLM service.
api_key	Yes	str	API key required for connecting to the remote LLM service.
remote_model_name	Yes	str	Model name of the remote LLM service.
description	No	str	Description of the abstract local model. This parameter is optional.
database	No	str	Name of the database to which the model belongs. This parameter is optional.

Return type: None.

list_models

Description: Queries all models in a specified or the default database.

Syntax:

```
list_models(database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
database	No	str	Name of the database where the models to be queried are located. If it is **None**, the default database is used.

Return type: list[tuple], a list of tuples, where each tuple contains model names.

 Copyright © Huawei Technologies Co., Ltd.

drop_model

Description: Deletes the abstract local model with a specified name.

Syntax:

```
drop_model(model_name)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
model_name	Yes	str	Name of the abstract local model.

Return type: None.

describe_model

Description: Queries the details of a specified model.

Syntax:

```
describe_model(model_name, database=None)
```

Input parameters:

Parameter	Mandatory (Yes/No)	Parameter Type	Description
model_name	Yes	str	Name of the model to be queried.
database	No	str	Name of the database where the model is located. This parameter is reserved and its default value is **None**.

Return type: Details of the model.

2.1.5.12 UDFs

2.1.5.12.1 UDF Overview

A UDF is a function defined by a user according to a specific requirement, and is used to extend a function of a database or a data processing system. UDFs can perform various operations, such as data transformation, complex computing, and data aggregation. These operations may exceed the capabilities of built-in functions provided by the system.

UDFs are used to:

 Copyright © Huawei Technologies Co., Ltd.

- Extended functions: UDFs allow you to extend the functions of the database or data processing system and perform operations that are not supported by built-in functions.
- Complex computation: UDFs can be used to perform complex mathematical or logical computations that may involve multiple steps and conditions.
- Data conversion: UDFs can be used for data cleansing and conversion, such as formatting dates, converting strings, or encoding data.
- Service logic encapsulation: UDFs can encapsulate service logic within functions, making data processing more modular and reusable.
- Performance optimization: In some cases, complex computational logic can be executed on the database server via UDFs to reduce data transfer and enhance performance.

NOTE

In big data processing, if you need to execute a UDF on a DataFrame to handle complex logic or call third-party libraries, but the container or cluster executing the UDF only has basic Python libraries pre-installed, you can use the **pip install package** command within the UDF to install the latest version of the dependency, thereby flexibly utilizing various Python libraries. To install a dependency of a specified version, include the version number in the command, for example, **pip install pandas==2.2.2**.

Notes and Constraints

- UDFs can be programmed only in Python.
- Default values cannot be set for UDF parameters. Additionally, if a function is created with precision specified for its parameters or return value, no precision validation will be performed.
- Function overloading is not supported.

UDF Types

The Aura DataFrame SDK supports the registration and usage of various types of UDFs, including Scalar UDF, Class UDF, Vectorized UDF, UDTF, and UDAF.

Table 2-24 UDF types

Type	Input:	Output	Description
Scalar UDF	Single row, multiple parameters, for example, (a, b, c)	Single row output, a single scalar value (e.g., int, float, string, date)	This is the most common form, where only one row of data is processed each time the function is called. It can be used for simple calculations, formatting, cleansing, or conversions. For example, string cleansing, mathematical calculations, and date conversion. Analogous to regular functions in Python.

 Copyright © Huawei Technologies Co., Ltd.

Type	Input:	Output	Description
Class UDF	Same as Scalar UDF: single row input	Single row output	It is an OOP encapsulation of scalar UDFs and is suitable for scenarios involving internal caching, initialization parameters, or model objects. For example, a model, regular expression, or configuration needs to be loaded once inside the function. The database initializes resources internally only once, avoiding reconstruction with each call.
Vectorized UDF	Batch input of rows, with the type of pandas.Series/ pyarrow.ChunkedArray	Batch output of an equal number of rows in vectorized structure	The most efficient method for single-row conversions, designed for batch processing. It is suitable for vectorized mathematical computations, batch text processing, embedding handling, NLP tokenization. It has SIMD/batch execution advantages and is commonly used with Arrow and Pandas.
UDTF	Single-row input	Multiple row outputs	A single input may produce multiple output rows. It is similar to Python's yield. The function can be used for splitting, expanding, or unpacking structured data. For example, splitting JSON arrays, expanding multimodal data, audio/video slicing, tokenizing, and explosive expansion.
UDAF	Multiple row inputs, possibly from different partitions	Single output	Aggregates a group of data: summation, statistics, clustering, embedding aggregation. Features lifecycle methods like accumulate, merge, and finish, supporting distributed execution.

UDF Registration Methods

There are two primary methods for registering a UDF: explicit registration and implicit registration.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-25 UDF registration methods

Registration Method	Description	Dependent on Session Object	Intrusively Add Registration Logic	Applicable Scenarios	Reference
Explicit registration	Code explicitly specifies the registration information of the UDF.	Yes	Yes	If you wish to precisely control the registration timing, allow intrusive addition of registration logic, or require separation between registration and usage of scalar UDFs under the same backend connection.	**Explicit Registration Syntax for UDFs**
Implicit registration	Automatically discovers and registers UDFs during runtime.	No	No	If you prefer non-intrusive registration of scalar UDFs and do not require separation between registration and usage of scalar UDFs under the same backend connection.	**Implicit Registration Syntax for UDFs**

UDF Registration Types

Whether using explicit or implicit registration, the meaning of registration differs depending on the UDF type. Details are provided in the table below.

Table 2-26 UDF registration types

UDF Registration Type	Description	Vectorized	Applicable Scenarios & Characteristics	Reference
Python	Registers a raw Python function or class into the database.	No	Processes data row by row, suitable for simple or specific calculations, but with lower performance.	**Python/PyArrow/Pandas UDF Registration Parameters**

Copyright © Huawei Technologies Co., Ltd.

UDF Registration Type	Description	Vectorized	Applicable Scenarios & Characteristics	Reference
Builtin	Obtains the handle of an existing function in the database; no actual registration operation is performed.	No	Directly calls existing functions from the database backend, suitable for using native database functions.	**Builtin UDF Registration Parameters**
PyArrow	Registers a Python function or class that accepts pyarrow.ChunkedArray as input and output with the database.	Yes	Uses PyArrow's high-performance computing capabilities, perfect for handling large datasets or requiring efficient computations.	**Python/PyArrow/Pandas UDF Registration Parameters**
Pandas	Registers a Python function or class that accepts pandas.Series as input and output with the database.	Yes	Uses Pandas' vectorized operations, ideal for performing complex data processing at the Python level.	**Python/PyArrow/Pandas UDF Registration Parameters**

Scalar UDFs currently support all the preceding four registration types.

For UDAFs and UDTFs, only Python and builtin registration types are supported.

Additional Arguments During UDF Runtime

Regarding runtime parameters for UDFs, both explicit and implicit registrations allow additional arguments to be passed using the **with_arguments** method. These parameters include **concurrency**, **min_concurrency**, **max_concurrency**, **timeout**, **dpu**, and **apu**. They enable fine-grained resource allocation, concurrency control, and execution time limits for UDFs.

For details, see **WITH ARGUMENTS Syntax**.

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.12.2 Development Notes

Operating Environment: Requirements

- UDFs can be programmed only in Python.
- The Python version of the UDF runtime environment supports only the 3.10 and 3.11 series. Version 3.11.9 is recommended.
- When using Cloudpickle to serialize the main function, ensure that the local Python version is 3.11.9 and the Cloudpickle version is 3.0.0. Otherwise, the function body may fail to be parsed due to version inconsistency in the Aura running environment.

Function Definition Constraints

- Default values cannot be set for UDF parameters. Additionally, if a function is created with precision specified for its parameters or return value, no precision validation will be performed.
- Function overloading is not supported.

Function and Code Package Requirements

- You must define either the function body or the code package archive path.
- For complex business scenarios involving extensive UDF code, it is recommended to register the function as a file archive: Archive all code files required by the UDF into a single compressed package, upload the package to OBS, and specify the storage path of the package when creating the function. The requirements for the compressed package are as follows:
  - The package must contain the **{FUNCTION_NAME}_source.py** file. This file is treated as the main entry point for the function, and its content must be serialized into hexadecimal format using Cloudpickle.
  - The code files for modules imported by the main function must be correctly organized.
  - Only zip format is supported for the compressed package.

    An example of the compressed package is as follows:

```
my_udf.zip/
├── my_udf_source.py
├── my_helper.py
└── other/
    └── utils.py
```

Recommended Registration Method

Aura provides three methods for registering UDFs: **Explicit registration using the ibis-aura SDK, implicit registration**, and **DDL registration using SQL**. Explicit registration using the ibis-aura SDK is recommended because it offers the following advantages:

- The registration interface provided by the SDK fully adheres to the Aura DDL specifications. By correctly invoking this interface, you can avoid issues where UDFs fail due to constraint violations.
- The SDK provides additional constraint checks, such as Python version detection and automatic organization of UDF code packages.

 Copyright © Huawei Technologies Co., Ltd.

Dependency Management

- The third-party dependency package list for Python UDF must ensure compatibility among libs, and you are advised to explicitly specify stable versions for all dependencies.
- When debugging UDFs locally, you are advised to use third-party packages from Huawei sources to set up the debugging environment. This prevents issues such as installation failures or version incompatibilities that may arise during UDF execution in Aura but not in the local environment.

Storage and Permissions

You can only use buckets of the OBS parallel file system to store UDF code compressed packages, and you must grant read permissions to IAM users on LakeFormation.

2.1.5.12.3 Data Type Mapping

The following table shows the mapping between Python and Aura data types.

Table 2-27 Mapping between Python and Aura data types

Python	Aura
int	BIGINT
bool	BOOLEAN
float	DOUBLE PRECISION
str	TEXT
bytes	BYTEA
datetime.date	DATE
datetime.time	TIME
datetime.datetime	TIMESTAMP
decimal.Decimal	NUMERIC

2.1.5.12.4 Environment Variable Configuration

env_vars allows dynamic injection of environment variables into the Actor process during UDF invocation. This eliminates the need for creating redundant UDF definitions for different environment variables.

Basic Usage

The following code snippet shows only the usage of the **env_vars** parameter:

```
# Dataset API — Input dict.
ds = ds.map(
    fn="my_schema.data_loader",
    on=["url_col"],
```

 Copyright © Huawei Technologies Co., Ltd.

```
    env_vars={"DATA_SOURCE": "obs://bucket/path", "REGION": "cn-north-4"},
)

# with_arguments — Input dict or a JSON string.
result = my_udf(t.col_a).with_arguments(
    env_vars={"DATA_SOURCE": "obs://bucket/path"},
)
```

Access the environment variables in the UDF using **os.environ**:

```
def my_udf(url: str) -> str:
    import os
    data_source = os.environ.get("DATA_SOURCE", "")
    return load_from(url, base=data_source)
```

Type Verification

Verify types in **with_arguments()** in Python.

Input	Behavior
env_vars={"KEY": "VALUE"}	Normal. Both the key and value are of the str type.
env_vars={1: "val"}	TypeError: keys must be str
env_vars={"key": 123}	TypeError: values must be str
env_vars={}	Equivalent to not setting the key. No ENV_VARS clause is generated.
env_vars='{"K":"V"}'	JSON string passthrough
env_vars='not json'	TypeError: must be valid JSON
env_vars=None	Default value. No ENV_VARS clause will be generated.

Key Format Constraints

The key of **env_vars** must comply with the POSIX specifications ^[a-zA-Z_][a-zA-Z0-9_]*$. The maximum length is 1,022 bytes. If the kernel is invalid, **ERROR** is returned.

Example Key	Valid	Description
MY_KEY	Yes	Standard POSIX key
_UNDERSCORE_KEY	Yes	It starts with an underscore (_).
KEY123	Yes	It contains digits, but the first character cannot be a digit.
1KEY	No	It starts with a digit.

 Copyright © Huawei Technologies Co., Ltd.

Example Key	Valid	Description
MY-KEY	No	It contains hyphens (-).
MY.KEY	No	It contains periods (.).
MY=KEY	No	It contains equal signs (=).
my key	No	It contains spaces.

Safety Precautions

Avoiding sensitive credentials such as AK/SK in env_vars

```
# Forbidden — Transmitting credentials in plaintext through SQL statements may cause leaks.
my_udf(t.col).with_arguments(
    env_vars={"ACCESS_KEY_ID": "xxx", "SECRET_ACCESS_KEY": "yyy"}
)

# Forbidden — UDF_TEMP_AK/UDF_TEMP_SK is a reserved key for server credentials. Setting it will trigger an error.
my_udf(t.col).with_arguments(
    env_vars={"UDF_TEMP_AK": "xxx", "UDF_TEMP_SK": "yyy"}
)
# ERROR: env_vars key "UDF_TEMP_AK" conflicts with reserved server credentials. UDF invocation is terminated.
```

Using the Kernel IAM agency token for automatic injection

The kernel automatically obtains the temporary AK/SK using the IAM agency token and injects them as environment variables **UDF_TEMP_AK**, **UDF_TEMP_SK**, and **UDF_TEMP_SECURITY_TOKEN**. Non-credential configurations such as the OBS endpoint are passed through **env_vars**. The UDF reads the values through **os.environ** and uses them together with the esdk-obs-python SDK.

```
class ObsDataReader:
    """Read data from OBS.

    Credentials (UDF_TEMP_AK/SK/TOKEN) are automatically injected by
    the kernel via IAM delegation token — no client-side setup needed.
    UDF_OBS_ENDPOINT must be provided by the user via env_vars.
    """

    def __init__(self, bucket: str):
        self.bucket = bucket

    def _get_obs_config(self) -> tuple:
        """Read OBS config from env vars."""
        import os

        # Credentials — auto-injected by kernel, do NOT set these yourself
        ak = os.environ.get("UDF_TEMP_AK", "")
        sk = os.environ.get("UDF_TEMP_SK", "")
        token = os.environ.get("UDF_TEMP_SECURITY_TOKEN", "")

        # Endpoint — user-provided via env_vars
        endpoint = os.environ.get("UDF_OBS_ENDPOINT", "")

        if not ak or not sk:
            raise RuntimeError(
                "OBS credentials not available. "
                "Ensure IAM delegation token is configured on the server."
            )
```

 Copyright © Huawei Technologies Co., Ltd.

```
        if not endpoint:
            raise RuntimeError(
                "OBS endpoint not configured. "
                "Pass it via env_vars: with_arguments(env_vars={'UDF_OBS_ENDPOINT': '...'})"
            )

        return ak, sk, token, endpoint

    def __call__(self, object_key: str) -> str:
        """Read an object from OBS and return its content as string."""
        from obs import ObsClient

        ak, sk, token, endpoint = self._get_obs_config()

        obs_client = ObsClient(
            access_key_id=ak,
            secret_access_key=sk,
            security_token=token,
            server=endpoint,
        )

        try:
            resp = obs_client.getObject(
                self.bucket, object_key, loadStreamInMemory=True
            )
            if resp.status >= 300:
                raise RuntimeError(
                    f"OBS read failed: status={resp.status}, reason={resp.reason}"
                )
            return resp.body.buffer.decode("utf-8")
        finally:
            obs_client.close()
```

Use method:

```
import pandas as pd
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint="*",
    aura_workspace_id="*",
    lf_catalog_name="*",
    access_key="*",
    secret_key="*",
    default_database="my_schema",
    use_single_cn_mode=True,
)
# Register the UDFs
try:
    con.create_scalar_function(ObsDataReader, name="obs_reader", database="my_schema")
    # --- Direct expression API ---
    t = con.from_pandas(pd.DataFrame({"object_key": [""]}))._table
    result = con.get_function("obs_reader", database="my_schema")(
        t.object_key
    ).with_arguments(
        bucket="my-data-bucket",
        env_vars={"UDF_OBS_ENDPOINT": "obs.cn-north-4.myhuaweicloud.com"},
    )
    # --- Dataset API ---
    ds = con.from_pandas(pd.DataFrame({"object_key": [""]}))
    ds = ds.map(
        fn="my_schema.obs_reader",
        on=["object_key"],
        bucket="my-data-bucket",
        as_col="content",
        env_vars={"UDF_OBS_ENDPOINT": "obs.cn-north-4.myhuaweicloud.com"},
    )
    ds.show()
finally:
    con.close()
```

 Copyright © Huawei Technologies Co., Ltd.

Reserved keys for server credentials (Setting them in **env_vars** will trigger an error.)

Reserved Key	Purpose
UDF_TEMP_AK	Temporary access key
UDF_TEMP_SK	Temporary secret key
UDF_TEMP_SECURITY_TOKEN	Temporary security token
SHARED_WORK_DIR	Path for session-level shared storage. All UDF instances in the same session can access this path for read and write operations.

These reserved keys are automatically injected by the kernel and cannot be manually set by users.

Risks:

- The frontend **env_vars** are transmitted in plaintext JSON format through SQL.
- They may be captured by audit logs, slow query logs, and error logs.
- If a preset environment variable conflicts with a user key, the kernel returns an error and stops calling the UDF.

Risks of overwriting system variables: Overwriting variables such as **PATH**, **LD_PRELOAD**, **PYTHONPATH**, **HOME**, and **PYTHONHOME** may lead to unexpected results.

2.1.5.12.5 Scalar UDF

A scalar UDF is a basic row-level function. It takes zero or more input parameters, processes the data in one row, and returns one result per row.

Scalar UDFs are typically used for mathematical operations, complex type conversions, and custom formatting.

The behavior of a scalar UDF is similar to that of built-in functions. However, its implementation is user-defined.

Development Process

The overall process of developing a Python scalar UDF is as follows:

Step 1 Write and debug Python code according to user service logic.

Sample code:

```
import numpy as np

def pysum(arg1: int, arg2: int) -> int:
    printLog(arg1, arg2)
    return np.sum(np.array([arg1, arg2]))
```

Step 2 Register a Python UDF.

Create a session using **ibis-aura**.

 Copyright © Huawei Technologies Co., Ltd.

```
import aura_frame as aura
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                    # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
```

Register a Python UDF.

```
try:
    udf = con.create_scalar_function(pysum, packages=["numpy"])
finally:
    con.close()
```

Step 3 View the Python UDF.

Aura provides the **show functions** command to view all functions in a specific schema and the **describe function function_name** method to view details about a specific function. For details, see **SHOW** and **DESCRIBE**.

Step 4 Invoke the Python UDF.

Use any Aura client to submit a query, for example:

```
select "test".pysum(3,4);
```

Step 5 View and optimize Python UDF performance metrics.

Aura supports performance monitoring during Python UDF execution and provides APIs to configure resource specifications and concurrency during UDF operation. For details, see **Python UDF Performance Tuning**.

----End

Example

```
import os
import ibis
from aura_frame.multimodal import ai_lake
target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                  # Default database
    use_single_cn_mode=True,                        # Whether to enable the single-CN mode
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
```

 Copyright © Huawei Technologies Co., Ltd.

```
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),        # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                  # OBS endpoint
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
)
# Define a scalar UDF for summing two numbers.
def add_udf(lhs: float, rhs: int) -> float:
    return lhs + rhs
try:
    # Create a test table and write data into it.
    table_name = "product_orders"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("add_udf", database=target_database, if_it_exists=True)
    # Register the scalar UDF.
    udf = con.create_scalar_function(
        add_udf,
        database=target_database,
        comment="To test create_scalar_function",
    )
    # Use the UDF to process data in the product_orders table.
    ds = con.load_dataset("product_orders", database=target_database)
    ds = ds.map(fn=udf, on=[ds.price, ds.quantity], as_col="sum_column")
    ds = ds.select_columns(ds.price, ds.quantity, ds.sum_column)
    print(ds.execute())
finally:
    con.close()
```

2.1.5.12.6 Class UDF

A class UDF uses object-oriented programming to enable stateful processing logic. To implement it, you must define methods such as **__init__** and **__call__**.

Class UDF Registration

When registering a class UDF, you pass a Python class where **__init__**, **__call__**, and **__del__** have specific meanings (refer to the table below for details). Other class attributes and instance methods can be freely added by you without restrictions.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-28 Specific instance methods when registering a Python class

Python Class Method	Mandatory (Yes/No)	Parameter	Description	Applicable Scenarios
__init__(self, *args, **kwargs)	No	Parameters are passed through the **with_arguments** method. Only scalar values are allowed.	Constructor of the UDF.	Initializes UDF attributes (such as saving parameters, opening files, and establishing connections).
__call__(self, *args, **kwargs)	Yes	Parameters are passed through the UDF operator. Both scalar values and column names are allowed.	Entry point for the service logic of the UDF.	Processes data and performs calculations in the UDF.
__del__(self)	No	Parameters cannot be passed.	Destructor of the UDF.	Clears UDF resources (for example, closing files and disconnecting networks).

Note: The life cycle of **__init__** is called once for each batch. That is, the Aura kernel initializes variables each time it processes a batch.

Example

This example defines a class UDF with initialization parameters. It receives name parameters in **__init__** and generates a welcome message combined with the city name in **__call__**. After registering it to the database, it uses the **map** method to generate greeting messages for each row of data in the **city_list** table.

1. Define the class UDF class **Hello**. It receives the **firstname** and **lastname** parameters in **__init__** and generates a welcome message combined with the city name in **__call__**.
2. Connect to the Aura service and set the UDF workspace.
3. Create the **city_list** table (containing the **city** column) and insert five city records.
4. Delete the existing UDF with the same name (repeated execution is supported).
5. Register the class UDF to the target database.
6. Load the **city_list** dataset, call the UDF via the **map** method on the **city** column to generate greetings, passing the **firstname** and **lastname** parameters.

 Copyright © Huawei Technologies Co., Ltd.

7. Select the **city** and **new_column** columns, execute the query, and print the results.
8. Disable the Aura connection.

```
import os
import ibis
from aura_frame.multimodal import ai_lake

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database="postgres",                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),       # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                 # OBS endpoint
    access_key=os.getenv("access_key"),                 # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                 # Secret Key (SK)
)
# Define a class UDF: Greeting generator with initialization parameters.
class Hello:
    def __init__(self, firstname: str, lastname: str):
        self.firstname = firstname
        self.lastname = lastname
    def __call__(self, location: str) -> str:
        return f"Hello {self.firstname} {self.lastname}! Welcome to {location}"

try:
    # Create a test table and write data into it.
    table_name = "city_list"
    table_schema = ibis.schema({"city": "string"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (city) VALUES
        ('Beijing'),
        ('Shanghai'),
        ('New York'),
        ('London'),
        ('Tokyo')
    ''')
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("hello", database=target_database, if_it_exists=True)
    # Register the class UDF.
    hello_udf = con.create_scalar_function(
        Hello,
        name="hello",
        database=target_database,
    )
```

 Copyright © Huawei Technologies Co., Ltd.

```
    # Use the UDF to process data in the city_list table.
    ds = con.load_dataset("city_list", database=target_database)
    ds = ds.map(
        fn=hello_udf,
        on=[ds.city],
        as_col="new_column",
        firstname="Ethan",
        lastname="Carter",
    )
    ds = ds.select_columns(ds.city, ds.new_column)
    print(ds.execute())
finally:
    con.close()
```

2.1.5.12.7 Vectorized UDF

Designed to overcome performance limitations of traditional row-based UDFs, vectorized UDFs execute operations in batches. They typically accept and return data in PyArrow or Pandas formats.

Example

The two examples below demonstrate how to use vectorized UDFs.

- Example 1: Use PyArrow to accelerate vectorization. This example explicitly registers a PyArrow vectorized UDF using **con.create_scalar_function**. The framework automatically identifies the UDF as a PyArrow type based on the type annotation and then calls it using **ds.map**.

```
import os
import ibis
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.types.vectorized import PyarrowVector
import pyarrow.compute as pc

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database="postgres",                       # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),       # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                 # OBS endpoint
    access_key=os.getenv("access_key"),                 # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                 # Secret Key (SK)
)
def calculate_product_pyarrow(
    prices: PyarrowVector[float],
    quantities: PyarrowVector[int],
) -> PyarrowVector[float]:
```

 Copyright © Huawei Technologies Co., Ltd.

```
    return PyarrowVector[float](pc.multiply(prices, quantities))

try:
    # Create a test table and write data into it.
    table_name = "sales_records"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("calculate_product_pyarrow", database=target_database, if_it_exists=True)
    # Explicitly register the PyArrow vectorized UDF.
    udf_pyarrow = con.create_scalar_function(
        calculate_product_pyarrow,
        name="calculate_product_pyarrow",
        database=target_database,
        register_type=aura.udf.RegisterType.STAGED,
    )
    ds = con.load_dataset("sales_records", database=target_database)
    ds = ds.map(fn=udf_pyarrow, on=[ds.price, ds.quantity], as_col="product_column")
    ds = ds.select_columns(ds.price, ds.quantity, ds.product_column)
    print("PyArrow vectorization result:")
    print(ds.execute())
finally:
    con.close()
```

- Example 2: Use Pandas to accelerate vectorization. This example implicitly registers a Pandas vectorized UDF using the **@aura.udf.pandas** decorator, with the registration type set to **STAGED**.

```
import os
import ibis
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.types.vectorized import PandasVector
import pandas as pd

target_database = "test"
# Implicitly register a UDF.
@aura.udf.pandas(database=target_database, register_type=aura.udf.RegisterType.STAGED)
def calculate_product_pandas(
    prices: PandasVector[float],
    quantities: PandasVector[int],
) -> PandasVector[float]:
    return PandasVector[float](prices * quantities, dtype=pd.Float64Dtype())
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is
located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    lf_instance_id=os.getenv ("lf_instance_id"),        # LakeFormation instance ID
    project_id=os.getenv ("project_id"),               # Project ID
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database=target_database,                 # Default database
    use_single_cn_mode=True,                         # Whether to enable the single-CN mode
)
```

 Copyright © Huawei Technologies Co., Ltd.

```
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),  # Path for storing UDFs in the parallel file
system.
    obs_bucket_name=os.getenv("obs_bucket_name"),        # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                  # OBS endpoint
    access_key=os.getenv("access_key"),                  # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                  # Secret Key (SK)
)
try:
    # Create a test table and write data into it.
    table_name = "sales_records"
    table_schema = ibis.schema({"price": "float64", "quantity": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (price, quantity) VALUES
        (100.0, 10),
        (200.0, 20),
        (150.0, 15)
    ''')
    # Delete the existing UDF with the same name (repeated execution is supported).
    con.delete_function("calculate_product_pandas", database=target_database, if_it_exists=True)
    # Use Pandas vectorized UDF (which has been implicitly registered using the decorator above).
    ds2 = con.load_dataset("sales_records", database=target_database)
    ds2 = ds2.map(fn=calculate_product_pandas, on=[ds2.price, ds2.quantity],
as_col="product_column")
    ds2 = ds2.select_columns(ds2.price, ds2.quantity, ds2.product_column)
    print("Pandas vectorization result:")
    print(ds2.execute())
finally:
    con.close()
```

2.1.5.12.8 UDTF

A user-defined table function (UDTF) processes one input row and generates multiple output rows. This makes it ideal for multimodal data processing scenarios like video/audio frame splitting or sample generation.

The development process for a UDTF is similar to that of a scalar UDF, with the following key considerations:

- The handler currently only supports returning iterators, so using yield is recommended.
- UDTFs do not support concurrent execution or vectorized computation.
- If the handler enters an infinite loop or produces endless output, Aura will automatically terminate the program based on the timeout setting, which defaults to 1,000 seconds.

Example

The example below demonstrates how to register and use a UDTF. The example uses the **sequence_config** table (which contains three **int32** columns: **start_val**, **end_val**, and **step_val**) and the UDTF **py_generate_series** to generate an integer sequence from **start_val** to **end_val** with a step of **step_val** for each row.

```
import os
import ibis
import aura_frame as aura
```

 Copyright © Huawei Technologies Co., Ltd.

```
from aura_frame.multimodal import ai_lake
import ibis.expr.datatypes as dt

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),        # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),               # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),               # Secret Key (SK)
    default_database="postgres",                      # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),    # Path for storing UDFs in the parallel file system.
    obs_bucket_name=os.getenv("obs_bucket_name"),          # Name of the parallel file system.
    obs_server=os.getenv("obs_server"),                    # OBS endpoint
    access_key=os.getenv("access_key"),                    # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                    # Secret Key (SK)
)

# Define a UDTF: Enter start, endnum, and step to generate an integer sequence.
def py_generate_series(start: int, endnum: int, step: int):
    current = start
    while current <= endnum:
        yield {"col1": current}
        current += step

try:
    # Create a test table and write data into it.
    table_name = "sequence_config"
    table_schema = ibis.schema({"start_val": "int32", "end_val": "int32", "step_val": "int32"})
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (start_val, end_val, step_val) VALUES
      (1, 5, 1),
      (10, 20, 5),
      (0, 10, 2)
    ''')
    # Delete the existing UDTF with the same name (repeated execution is supported).
    con.delete_function("py_generate_series", database=target_database, if_it_exists=True)
    func_signature = aura.Signature(
        parameters=[
            aura.Parameter(name="start", annotation=int),
            aura.Parameter(name="endnum", annotation=int),
            aura.Parameter(name="step", annotation=int),
        ],
        return_annotation=dt.Struct({"col1": int}),
    )
    con.create_table_function(py_generate_series, database=target_database, signature=func_signature)
    ds = con.load_dataset("sequence_config", database=target_database)
    py_generate_series_handler = con.get_function("py_generate_series", database=target_database)
    ds = ds.flat_map(
        fn=py_generate_series_handler,
```

 Copyright © Huawei Technologies Co., Ltd.

```
        on=[ds.start_val, ds.end_val, ds.step_val],
        as_col="udtf_col",
    )
    res = ds.execute()
    print(res)
finally:
    con.close()
```

2.1.5.12.9 UDAF

A user-defined aggregate function (UDAF) aggregates multiple input rows into a single set of values. It is particularly useful for summarizing statistical features from multimodal data, such as video, audio, or images, or consolidating multimodal labels.

The overall development process for UDAF can be modeled after scalar UDFs, with the following key considerations:

- The handler must specify a Python class that includes four essential methods: **accumulate**, **aggregate_state**, **merge**, and **finish**. The implemented **accumulate**, **aggregate_state**, and **merge** methods must be side-effect-free. This means their outputs should depend solely on input parameters and the current aggregation state, not on the factors such as invocation count, order, or external conditions.
- The **finish** method of the UDAF will be called globally only once for each group (specified by the **GROUP BY** clause), regardless of whether it is executed in single-CN mode or multi-DN mode.
- When defining a non-STRICT UDAF, you must ensure the **accumulate** method correctly handles NULL values (represented as **None** in Python).
- UDAFs cannot be used as window functions. That is, they cannot be used within an **OVER** clause.
- UDAFs do not support concurrent execution or vectorized computation.

UDAF Registration

To register a UDAF, you must provide a Python class that includes six specific instance methods: **__init__**, **aggregate_state**, **accumulate**, **merge**, **finish**, and **__del__**. These methods have predefined meanings as outlined in the table below. Beyond these requirements, you are free to add other classes or instance methods without restrictions.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-29 Specific instance methods when registering a UDAF

Python Class Method	Mandatory (Yes/No)	Parameter	Description	Applicable Scenarios
__init__(self, *args, **kwargs)	No	Parameters are passed through the **with_arguments** method. Only scalar values are allowed.	Constructor of the UDAF.	Initializes UDAF attributes (such as saving parameters, opening files, and establishing connections).
aggregate_state(self)	Yes	None.	Serializable intermediate aggregation state for cross-partition transfer.	Indicates the sharing status between accumulate and merge.
accumulate(self, *args, **kwargs)	Yes	Parameters are passed through the UDAF operator. Both scalar values and column names are allowed.	Incrementally updates the intermediate aggregation state.	Processes data during normal scanning phase.
merge(self, other_state)	Yes	Intermediate state from other partitions.	Merges states from other partitions into the current intermediate state.	Combines results across multiple partitions during distributed reduction.
finish(self)	Yes	None.	Outputs the final aggregated result.	Finalizes the aggregation process.
__del__(self)	No	Parameters cannot be passed.	Destructor of the UDAF.	Clears UDAF resources (for example, closing files and disconnecting networks).

Example

```
import os
import json
import datetime as pydt
import ibis
```

 Copyright © Huawei Technologies Co., Ltd.

```
import ibis.expr.datatypes as dt
import pandas as pd
from typing import Any, Dict
import aura_frame as aura
from aura_frame.multimodal import ai_lake
from aura_frame.multimodal.function import AggregateFnBuilder

target_database = "test"  #Create a database in LakeFormation, for example, test. For details, see Creating a DataArts Lake Formation Instance and Planning Metadata.
#Connect to the AuraJobV2 endpoint. For details about the parameters, see Establishing a Connection with the AuraJobV2 Endpoint.
con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database="postgres",                       # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)
#Configure the UDF deployment workspace. For details about the parameters, see Configuring the UDF Deployment Workspace.
con.set_function_staging_workspace(
    obs_directory_base=os.getenv("obs_directory_base"),
    obs_bucket_name=os.getenv("obs_bucket_name"),
    obs_server=os.getenv("obs_server"),
    access_key=os.getenv("access_key"),
    secret_key=os.getenv("secret_key"),
)
class _AggState:
    def __init__(self):
        self.sum_px_qty = 0.0
        self.sum_qty = 0.0
        self.rows_seen = 0
        self.rows_used = 0
class PythonVWAP:
    def __init__(self) -> None:
        self._state = _AggState()
    def accumulate(self, quantity: int, price: float, date: pydt.date, symbol: str) -> None:
        self._state.rows_seen += 1
        if quantity is None or price is None or date is None or symbol is None:
            return
        self._state.sum_px_qty += price * quantity
        self._state.sum_qty += quantity
        self._state.rows_used += 1
    @property
    def aggregate_state(self) -> Dict[str, Any]:
        return {
            "sum_px_qty": self._state.sum_px_qty,
            "sum_qty":    self._state.sum_qty,
            "rows_seen":  self._state.rows_seen,
            "rows_used":  self._state.rows_used,
        }
    def merge(self, other_state: Dict[str, Any]) -> None:
        self._state.sum_px_qty += other_state.get("sum_px_qty", 0.0)
        self._state.sum_qty    += other_state.get("sum_qty",    0.0)
        self._state.rows_seen   += other_state.get("rows_seen",   0)
        self._state.rows_used   += other_state.get("rows_used",   0)
    def finish(self) -> str:
        vwap = (self._state.sum_px_qty / self._state.sum_qty
                if self._state.sum_qty != 0 else None)
        result = {
            "vwap":           round(vwap, 4) if vwap is not None else None,
            "total_quantity": self._state.sum_qty,
            "total_value":    round(self._state.sum_px_qty, 2),
            "rows_seen":      self._state.rows_seen,
```

 Copyright © Huawei Technologies Co., Ltd.

```
            "rows_used":      self._state.rows_used,
        }
        return json.dumps(result)

try:
    # Create a test table and write data into it.
    table_name = "stock_trades"
    table_schema = ibis.schema({
        "quantity": "int32",
        "price": "float64",
        "date": "date",
        "symbol": "string",
    })
    con.sql(f'DROP TABLE IF EXISTS "{target_database}"."{table_name}";')
    con.create_table(
        table_name,
        schema=table_schema,
        database=target_database,
        table_format="orc",
        if_not_exists=True,
    )
    con.sql(f'''
    INSERT INTO "{target_database}"."{table_name}" (quantity, price, date, symbol) VALUES
        (100, 150.25, '2024-01-15', 'AAPL'),
        (200, 152.30, '2024-01-16', 'AAPL'),
        (150, 148.50, '2024-01-17', 'AAPL'),
        (300, 145.75, '2024-01-15', 'GOOG'),
        (250, 147.80, '2024-01-16', 'GOOG'),
        (100, 143.20, '2024-01-17', 'GOOG'),
        (500, 380.50, '2024-01-15', 'MSFT'),
        (300, 385.40, '2024-01-16', 'MSFT'),
        (200, 382.10, '2024-01-17', 'MSFT')
    ''')
    con.delete_function("pythonvwap", database=target_database, if_it_exists=True)
    con.create_agg_function(
        PythonVWAP,
        database=target_database,
        signature=aura.Signature(
            parameters=[
                aura.Parameter(name="quantity", annotation=int),
                aura.Parameter(name="price", annotation=float),
                aura.Parameter(name="date", annotation=dt.date),
                aura.Parameter(name="symbol", annotation=str),
            ],
            return_annotation=str,
        ),
        volatility=aura.udf.VolatilityType.IMMUTABLE,
        strict=False,
    )
    ds = con.load_dataset("stock_trades", database=target_database)
    vwap_handler = con.get_function("pythonvwap", database=target_database)
    udaf_builder = AggregateFnBuilder(
        fn=vwap_handler,
        on=[ds.quantity, ds.price, ds.date, ds.symbol],
        as_col="VWAP_column",
        num_cpus=0.5,
    )
    ds = ds.aggregate([udaf_builder], by=[ds.symbol])
    res = ds.execute()
    df = res.copy()
    parsed = df["VWAP_column"].apply(json.loads).apply(pd.Series)
    df = pd.concat([df.drop(columns=["VWAP_column"]), parsed], axis=1)
    print(df)
finally:
    con.close()
```

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.12.10 WITH ARGUMENTS Syntax

The **WITH ARGUMENTS** syntax enhances the capabilities of existing UDFs, primarily enabling users to configure the initial state upon UDF execution. With this feature, users can:

- Configure runtime resources and concurrency: Specify the necessary resources (for example, memory, CPU) for UDF operation along with the count of concurrent execution instances to enhance performance and resource efficiency.
- Set class initialization parameters: For UDFs defined within a class structure, employ the **WITH ARGUMENTS** syntax to pass initialization constructor arguments, thereby establishing a particular initial state during UDF instantiation.

Key usage considerations include:

- **WITH ARGUMENTS** accommodates several predefined parameters. Refer to **Table 2-30** for specifics. Ensure class initialization parameter names do not conflict with these fixed parameters, noting that parameter names are case-sensitive.
- Only constants can be used as input parameters.
- For UDFs defined as classes, users should define the **__init__** and **__del__** member methods as needed, but the **__call__** member method must be defined, as it serves as the main function entry point.
- **min_concurrency** and **max_concurrency** must be set simultaneously. Successful configuration indicates that adaptive concurrency is enabled.
- To set **init_concurrency**, enable **enable_udf_global_actor_pool** and set **min_concurrency** and **max_concurrency** first. Otherwise, an error will be reported.

Table 2-30 UDF runtime configuration list

Parameter	Description	Default Value
concurrency	Concurrency of the UDF execution, with a value range of [1, 1024].	1
min_concurrency	Minimum concurrency of the UDF execution, with a value range of [1, 1024].	null
max_concurrency	Maximum concurrency of the UDF execution, with a value range of [1, 1024].	null
init_concurrency	Warm-up concurrency of the UDF execution. The value range is [min_concurrency, max_concurrency].	(min_concurrency + max_concurrency) / 2

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Description	Default Value
timeout	Timeout duration for the UDF execution, with a value range of [1, 3600], in seconds.	1000
cpu	CPU resources of a single actor during UDF execution. The value can be a fractional value from 0.33 to 1 or an integer from 1 to 16. **Note that the system rounds the input value and retains two decimal places.**	1
memory	Memory resources of a single actor during UDF execution. The value ranges from 128 to 65536, in MiB. Decimals are not supported.	4096
arch	Physical architecture type during UDF execution. The value can be **arm** or **x86**. The default value of **arch** varies depending on the value of **apu**.	• NPU: The default value is **arm**. • Others: The default value is **x86**.
apu	Type and specifications of heterogeneous resources, for example, **NPU/.+/1**	null
env_vars	Runtime environment variable, which is dict[str, str] or a JSON string. For details, see **Environment Variable Configuration**.	null
scheduling_strategy	UDF Actor placement policy, which is a **SchedulingStrategy** object, object list, or JSON string. For details, see **UDF Actor Placement Policy**.	null

2.1.5.12.11 Adaptive Parallelism

Scenario

In AI data engineering scenarios involving massive data processing, it is necessary to call actors in parallel to enhance data processing efficiency and conduct distributed computing. However, the fixed level of parallelism relies on parameter-tuning experiences and repeated testing feedback. To improve user experience and reduce parameter-tuning time, adaptive parallelism is introduced to increase the ease of use of UDFs.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

Constraints on this feature:

- Valid values for **min_concurrency** and **max_concurrency** must be specified when executing a UDF.
- If the row count of the current batch is less than the computed parallelism level, the actual number of activated actors equals the row count. Otherwise, the actual number of activated actors equals the currently computed parallelism level.

When executing a UDF, setting both **min_concurrency** and **max_concurrency** parameters enables the adaptive parallelism feature.

The UDF begins executing the relevant data batch at the **min_concurrency** parallelism level. Each batch execution calculates the percentage of effective UDF computation time relative to the total time. When this percentage reaches 80% during a batch execution, the current parallelism level increases by steps of two up to the **max_concurrency** level, after which no further changes occur. If a round does not reach 80%, the parallelism level for the next round remains unchanged.

With the adaptive parallelism feature enabled for UDFs, significant variations exist in statistical information due to actors executing data across different batches sequentially. For details about how to view statistics in adaptive parallelism scenarios, see **Python UDF Performance Tuning**.

2.1.5.12.12 UDF Error Handling and Retry

Scenario

In AI data engineering, UDFs often involve error-prone operations such as network calls, external model inference, and data parsing. When an exception occurs during UDF execution, the default behavior is to throw an exception and stop the query. However, you may need more flexible error handling policies, for example:

- Ignoring: Some data exceptions do not affect the overall result and can be ignored.
- Logs: Error information is recorded for post-event troubleshooting, but the query is not interrupted.
- Automatic retry: In scenarios such as network timeout and temporary service unavailability, automatic retry is used to improve the success rate.
- Exception type-specific strategy: Different policies are used for different types of exceptions, for example, retry upon timeout and ignoring data format errors.

UDF error handling and retry allow users to dynamically wrap the error handling capability for registered UDFs using the **wrap_function** API without modifying source code of the UDFs. This function supports a combination of ignoring, logging, retrying, and backoff policies.

How to Use

1. **Register the source UDF**: Register a Python UDF (Scalar, Class, PyArrowVector, or PandasVector) following the normal process.

 Copyright © Huawei Technologies Co., Ltd.

2. **Call wrap_function for wrapping**: Use **AuraConnection.wrap_function()** to add error handling configuration to the source UDF and return a wrapped callable object.

```
wrapped = con.wrap_function(
    source_udf_name="my_func",       # Source UDF name
    source_database="my_db",         # Database where the source UDF is located
    config=ErrorHandlerConfig(...),   # Error handling configuration
    wrapper_name="my_func_wrapped",   # (Optional) Name of the wrapped UDF
    imports=["file1", "file2"],  # Additional import files for wrapping the UDF
    packages=["module1", "module2"]  # Additional dependency packages for wrapping the UDF
)
```

3. **Use the wrapped UDF to perform queries**: Use the wrapped UDF in operations such as **ds.map()** and **df.select()**.

```
ds = con.load_dataset("my_table", database="my_db")
result_ds = ds.map(wrapped, on="input_col", as_col="result")
df = result_ds.execute()
con.close()
```

Configuration Items

ErrorHandlerConfig is used to configure the error handling behavior. The parameters are as follows.

Table 2-31 ErrorHandlerConfig parameters

Parameter	Type	Description	Default Value
on_error	OnErrorAction	Error handling policy in the calling phase. • **RAISE**: An exception is thrown (default). • **IGNORE**: **None** is returned. • **LOG**: Logs are recorded and **None** is returned. • **RETRY**: Retry is performed for all exceptions.	RAISE
retry_on	list[type[Exception]]	Whitelist of exception types that trigger a retry during the calling phase. If a match is found, the retry logic is triggered. If no match is found, the **on_error** logic is triggered.	None
ignore_on	list[type[Exception]]	Whitelist of exception types that trigger the exception ignoring during the calling phase. If a match is found, **None** is returned with the **highest priority**.	None
log_on	list[type[Exception]]	Whitelist of exception types that trigger logging during the calling phase. If a match is found, logs are recorded and **None** is returned.	None

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description	Default Value
max_retries	int	Maximum number of retries during the calling phase.	0
retry_cleanup	Callable	Cleanup callback before retry during the calling phase. Signature: **(exception, instance) -> None**. The framework automatically injects the **is_exception_match** function into **__globals__** to facilitate branch processing by exception type. Exceptions thrown by the callback itself will be captured and a logged as warnings, without affecting the retry process.	None
init_on_error	OnErrorAction	Exception handling policy during the **__init__** phase. • **RAISE**: An exception is thrown (default). • **RETRY**: Retry is performed for all exceptions.	RAISE
init_max_retries	int	Maximum number of retries during the **__init__** phase.	0
init_retry_on	Sequence[Type[Exception]]	List of exception types that trigger a retry during the **__init__** phase.	()
init_retry_cleanup	Callable	Cleanup callback before retry during the **__init__** phase. Signature: **(exception, None) -> None**. Note that the **instance** parameter is **None** (the object has not been created).	None
backoff	BackoffStrategy	Backoff policy. • **FIXED**: Fixed interval. • **LINEAR**: Linear increase. • **EXPONENTIAL**: Exponential increase.	FIXED
initial_delay_ms	int	Initial delay, in milliseconds, which is used as the base for backoff calculation.	100
max_delay_ms	int	Maximum delay, in milliseconds. The calculation results of all backoff policies are subject to this constraint.	60000

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Type	Description	Default Value
log_fn	Callable	Custom log conversion function, with the signature **(exception, log_context) -> dict | None**. • Returns a **dict** for logging. • Returns **None** to skip logging for this instance. • Returns non-dict or raises an exception, it degrades to using the default **log_context**.	None
local_log_path	str	Local error log path. If the value is **None**, the Python logger outputs warnings.	None
obs_log_config	ObsLogConfig	OBS log uploading configuration, including: • **obs_server**: OBS server address. • **obs_bucket_name**: OBS bucket name. • **obs_directory_base**: Root directory of OBS logs. • **max_buffer_size**: Buffer size threshold (bytes). The default value is **1024**. The value range is [1, 10 x 1024 x 1024]. Upload is triggered when the buffer reaches this threshold. • **flush_interval_s**: Scheduled flush interval (seconds). The default value is **600.0**. Upload is triggered at this interval even if the buffer is not full. • **obs_log_format**: Log file format. The default value is **jsonl**. The options are **jsonl** and **parquet**.	None

Error Handling Priority

If exceptions occur during UDF execution, policies are matched based on the following priorities:

```
┌─ Priority 1: ignore_on (highest)
│  if ignore_on and isinstance(e, ignore_on):
│      return None                ← Ignore the exceptions.
│
├─ Priority 2: log_on
│  if log_on and isinstance(e, log_on):
│      _log_error(e, ...)
│      return None                ← Record a log and return None.
│
```

 Copyright © Huawei Technologies Co., Ltd.

```
├─ Priority 3: retry_on (Perform retry for a specified exception type.)
│  if retry_on and isinstance(e, retry_on)
│        and retry_count < max_retries:
│     delay = _compute_delay(...)
│     sleep(delay)
│     continue                    ← Retry
│
├─ Priority 4: on_error=RETRY (Perform retry for all exceptions.)
│  if on_error==RETRY and not retry_on
│        and retry_count < max_retries:
│     delay = _compute_delay(...)
│     sleep(delay)
│     continue                    ← Perform retry for all exceptions.
│
├─ Priority 5: on_error=IGNORE
│  if on_error == IGNORE:
│     return None                 ← Ignore the exceptions and return None.
│
├─ Priority 6: on_error=LOG
│  if on_error == LOG:
│     _log_error(e, ...)
│     return None                 ← Record a log and return None.
│
└─ Priority 7: raise (default/retry exhausted)
   raise                          ← Throw the original exceptions.
```

Key Rules

- **ignore_on** has the highest priority. After a match is found, **None** is returned, and the retry or logging logic is not executed.
- **log_on** has a higher priority than **retry_on**. After a match is found, a log is recorded and **None** is returned, and the retry logic is not executed.
- **retry_on** performs retry for a specified type of exceptions and performs retry for all exceptions. If both of them are used, **retry_on** takes precedence.
- If retries are exhausted, **on_error** determines the final behavior (**RAISE**, **IGNORE**, or **LOG**).

Backoff Policy

The retry interval is calculated based on the backoff policy and **initial_delay_ms**. All policies are subject to **max_delay_ms**.

Table 2-32 Interval of the backoff policy

Backoff Policy	Formula	Example (initial = 100 ms, max = 5,000 ms)
FIXED	delay = initial_delay_ms	100, 100, 100, ...
LINEAR	delay = initial_delay_ms × (attempt + 1)	100, 200, 300, 400, ...
EXPONENTIAL	delay = initial_delay_ms × 2^attempt	100, 200, 400, 800, 1600, 3200, ...

All policies are subject to **max_delay_ms**: **min(delay, max_delay_ms)**.

 Copyright © Huawei Technologies Co., Ltd.

Notes and Constraints

- **Custom exception restrictions**: Custom exceptions defined in classes are not supported. Exceptions with the same name defined in modules with the same name are considered the same exception.
- **No wrapping of BUILTIN functions**: **wrap_function** can only wrap Python UDFs and cannot wrap built-in database functions, such as **ABS()** and **UPPER()**.
- **WRAPPER defaults to TEMPORARY**: A wrapped UDF is temporary by default and is automatically destroyed after the session ends.
- **Vectorized UDF batch failure**: For UDFs of the **PyarrowVector** or **PandasVector** type, if an exception is triggered in any row of a batch, **None** is returned for the entire batch.
- **Sensitive data security**: If user-defined log fields and UDF parameters contain sensitive data (such as accounts and passwords), you need to desensitize them.

Example

Example 1: retry for a specified type of exception

When a UDF is called over the network and a TimeoutError occurs, exponential backoff is used for a maximum of three retry attempts:

```
from aura_frame import ErrorHandlerConfig, BackoffStrategy
from aura_frame.multimodal import ai_lake

con = ai_lake.connect(
    aura_endpoint=os.getenv("aura_endpoint"),         # Endpoint where the AuraJobV2 endpoint is located
    aura_endpoint_id=os.getenv("aura_endpoint_id"),    # AuraJobV2 endpoint ID
    aura_endpoint_name=os.getenv("aura_endpoint_name"),# AuraJobV2 endpoint name
    aura_workspace_id=os.getenv("aura_workspace_id"),  # Workspace ID
    lf_catalog_name=os.getenv("lf_catalog_name"),      # LakeFormation Catalog name
    access_key=os.getenv("access_key"),                # Access Key ID (AK)
    secret_key=os.getenv("secret_key"),                # Secret Key (SK)
    default_database="postgres",                       # LakeFormation database name
    use_single_cn_mode=True,   # Whether to enable the single-CN mode
    logging_level=logging.INFO # Set the log level.
)

try:
    wrapped = con.wrap_function(
        source_udf_name="call_api",
        source_database="my_db",
        config=ErrorHandlerConfig(
            retry_on=[TimeoutError],
            max_retries=3,
            backoff=BackoffStrategy.EXPONENTIAL,
            initial_delay_ms=200,
            max_delay_ms=5000,
        ),
    )
    ds = con.load_dataset("api_input", database="my_db")
    result_ds = ds.map(wrapped, on="input", as_col="result")
    df = result_ds.execute()
finally:
    con.close()
```

Example 2: combination of multiple policies

 Copyright © Huawei Technologies Co., Ltd.

The source UDF supports retry, log degradation, and silent ignoring. Exceptions are matched based on **ignore_on**, **log_on**, **retry_on**, and **on_error** (**RAISE** by default) in sequence.

1. **IndexError**: The error is silently ignored, and **None** is returned. No log is recorded, and no retry is performed.
2. **ValueError/TypeError**: After the error is recorded in a log, **None** is returned. Since **local_log_path="/tmp/udf_errors.log"** is configured, the log is written to this directory.
3. **ConnectionError/TimeoutError**:
   a. A maximum of five retries are allowed. Before each retry, a log is recorded, the delay is calculated, and the system enters the sleep state.
   b. If the exception persists after five retries, the default value of **on_error** is **RAISE**, and the original exception is thrown. The execution of this line fails.
4. Other unconfigured exceptions (such as **RuntimeError** and **KeyError**): The default value of **on_error** is **RAISE**, and the original exception is thrown. The execution of this line fails.

```
wrapped = con.wrap_function(
    source_udf_name="complex_model",
    source_database="my_db",
    config=ErrorHandlerConfig(
        retry_on=[ConnectionError, TimeoutError],
        log_on=[ValueError, TypeError],
        ignore_on=[IndexError],
        max_retries=5,
        backoff=BackoffStrategy.LINEAR,
        initial_delay_ms=500,
        local_log_path="/tmp/udf_errors.log",
    ),
)
```

Example 3: custom exception + custom log + cleanup callbackA UDF may throw multiple exceptions. If an exception requires retrying, some cleanup actions need to be performed before the retry.

1. In a UDF, you can use **log_extra["key"] = "..."** to add log fields and use **log_fn** to modify log content or define recorded log fields.
2. **retry_cleanup/init_retry_cleanup** configures the cleanup callbacks for the call/initialization phase.

```
import random
import pandas
from aura_frame.error_handler import (
    BackoffStrategy,
    ErrorHandlerConfig,
    ObsLogConfig,
    OnErrorAction,
)
from aura_frame.multimodal import ai_lake
class InvalidInputError(Exception):
    """Invalid input"""
class ExceedsLength(RuntimeError):
    """The string exceeds the maximum length."""
class WriteFileError(Exception):
    """Failed to write the file."""
class DataTransformer:
    """Class UDF for invoking external HTTP services to convert data"""
    def __init__(self):
        self.handler = open("/tmp/demo.txt", "a+")  # noqa: SIM115
```

 Copyright © Huawei Technologies Co., Ltd.

```
    def __call__(self, col: str) -> int:
        import os
        log_extra = globals().get("log_extra", None)
        if isinstance(log_extra, dict):
            log_extra.update(
                {"ak": os.getenv("UDF_TEMP_AK"), "sk": os.getenv("UDF_TEMP_SK")}
            )
        if not isinstance(col, str) or not col:
            raise InvalidInputError("The input cannot be an empty string.")
        if len(col) > 10:
            raise ExceedsLength("The input string is exceeds length.")
        if random.random() < 0.5:  # Simulate a 50% chance of file-write failure.
            raise WriteFileError("Failed to write to file.")
        try:
            self.handler.write(col + "\n")
        except Exception as e:  # noqa: BLE001
            raise WriteFileError(f"Failed to write to file: {e}")
        return id(self.handler)
    def __del__(self):
        self.handler.close()
session = ai_lake.connect(
    aura_endpoint="...",
    aura_endpoint_name="...",
    aura_workspace_id="...",
    lf_catalog_name="...",
    access_key="...",
    secret_key="...",
    security_token="...",
    default_database="err_handler",
    use_single_cn_mode=True,
)
session.sql("create schema if not exists err_handler;")
session.sql("set current_schema to err_handler;")
session.set_function_staging_workspace(
    obs_directory_base="...",
    obs_bucket_name="...",
    obs_server="...",
    access_key="...",
    secret_key="...",
    security_token="...",
)
session.delete_function("data_transformer_src", if_it_exists=True)
source_udf = session.create_scalar_function(
    DataTransformer,
    name="data_transformer_src",
)
def on_retry_cleanup(exc, instance):
    """Execute different cleanup logic based on the exception type before retry."""
    if instance is None:
        return  # In the __init__ phase, the object has not been created.
    # is_exception_match is automatically injected into __globals__ by the framework.
    if globals()["is_exception_match"](exc, WriteFileError):
        instance.handler.close()
    instance.handler = open("/tmp/demo.txt", "a+")  # noqa: SIM115
def custom_log_fn(exc, log_context):
    """Custom log format: Desensitization + Service label addition"""
    import traceback
    record = dict(log_context)
    record["exc_msg"] = traceback.extract_tb(exc.__traceback__)
    record["sk"] = "******"  # Desensitize sensitive information.
    return record
# Wrap the source UDF.
wrapped_udf = session.wrap_function(
    source_udf_name="data_transformer_src",
    wrapper_name="data_transformer",
    config=ErrorHandlerConfig(
        # Handle exceptions by category.
        on_error=OnErrorAction.LOG,
        ignore_on=[InvalidInputError],  # Invalid input -> Ignore
```

 Copyright © Huawei Technologies Co., Ltd.

```
        log_on=[ExceedsLength],  # Format error -> Log + Skip
        retry_on=[WriteFileError],  # Transient error -> Retry
        # Retry backoff policy
        backoff=BackoffStrategy.EXPONENTIAL,
        # Retry control during the calling phase
        max_retries=10, # A maximum of 10 retries are allowed.
        initial_delay_ms=100,
        max_delay_ms=5000,
        retry_cleanup=on_retry_cleanup,
        # Retry in the initialization phase
        init_on_error=OnErrorAction.RETRY,
        init_max_retries=2,
        init_retry_on=[ConnectionError],
        init_retry_cleanup=on_retry_cleanup,
        # Log definition
        log_fn=custom_log_fn,  # Custom log format
        local_log_path="/tmp/...",  # Local log directory
        obs_log_config=ObsLogConfig(
            obs_server="...",
            obs_bucket_name="...",
            obs_directory_base="...",
            max_buffer_size=4096,
            flush_interval_s=300.0,
            obs_log_format="parquet",
        ),
    ),
)
# Use the wrapped UDF for query (the usage is identical to the source UDF).
ds = session.from_pandas(pandas.DataFrame({"col": ["", "123", "1234", "12345", "123456", "1234567",
"123456789012345"]}))
ds = ds.map(wrapped_udf, on=[ds.col])
ds.show()
session.close()
```

2.1.5.12.13 UDF Actor Placement Policy

scheduling_strategy is used to control the placement policy of UDF actor. Two policy types are supported.

Policy 1: InstanceSchedulingStrategy

This policy determines the placement relationship of multiple actor instances used for the same query (co-location or dispersion).

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    InstanceSchedulingStrategy, InstanceAffinityMode
)

# Multiple instances must be deployed on the same node (shared cache/communication).
InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_REQUIRED)

# Multiple instances must be deployed on different nodes (fault tolerance/load balancing).
InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_REQUIRED)

# Preferentially on the same node, with rollback allowed
InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_PREFERRED)

# Preferentially on different nodes, with rollback allowed
InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_PREFERRED)
```

SAME_REQUIRED first distribution restriction: During the first distribution, if the POD does not have tags of instances used for the same query, the required constraint is not met, and an error is returned. This policy is valid only when multiple actors are distributed in sequence. SAME_PREFERRED or DIFF_PREFERRED is recommended for production. The following is an example:

 Copyright © Huawei Technologies Co., Ltd.

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    InstanceSchedulingStrategy,
    InstanceAffinityMode,
)

# Scenario: data preprocessing (dispersion first) + model inference (co-location required)
# Actor-A is distributed and placed first, and Actor-B traces it using the same label.

# Actor-A: DIFF_PREFERRED is used for distribution to different nodes.
ds = ds.map(
    fn="my_schema.preprocess",
    on=["raw_col"],
    as_col="preprocessed",
    scheduling_strategy=InstanceSchedulingStrategy(
        mode=InstanceAffinityMode.DIFF_PREFERRED,
        label="inference-pipeline",
    ),
)

# Actor-B: SAME_REQUIRED is used to trace the node where Actor-A is located and share the preprocessing
cache.
ds = ds.map(
    fn="my_schema.inference",
    on=["preprocessed"],
    as_col="score",
    scheduling_strategy=InstanceSchedulingStrategy(
        mode=InstanceAffinityMode.SAME_REQUIRED,
        label="inference-pipeline",
    ),
)
```

Key points:

- **Same label**: The two actors must use the same label string.
- **Non-REQUIRED first**: Actors with DIFF_PREFERRED or SAME_PREFERRED are placed first.
- **REQUIRED second**: Actors with SAME_REQUIRED are placed using labels.

If co-location is not required, use **SAME_PREFERRED** without customizing a label.

```
# Co-location is preferred and rollback is allowed —— simple scenario where SAME_REQUIRED is not
required
scheduling_strategy=InstanceSchedulingStrategy(mode=InstanceAffinityMode.SAME_PREFERRED),
```

Label naming rules:

- It contains a maximum of 63 characters.
- It must start and end with a letter or digit.
- Only letters, digits, hyphens (-), and underscores (_) are allowed.
- The reserved prefix **actor_** is not allowed.

Policy 2: NodeLabelSchedulingStrategy

This policy controls the locations of actors based on Kubernetes node tags:

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    NodeLabelSchedulingStrategy, In, NotIn, Exists, DoesNotExist
)

# Hard constraint: The node must be in the gpu pool.
NodeLabelSchedulingStrategy(hard={"pool": In("gpu")})

# Soft constraint: High-bandwidth nodes are preferred. If the requirement is not met, perform a roll back.
NodeLabelSchedulingStrategy(soft={"high_speed_net": Exists()})

# Combined use
```

 Copyright © Huawei Technologies Co., Ltd.

```
NodeLabelSchedulingStrategy(
    hard={"pool": In("gpu")},          # The requirement must be met.
    soft={"high_mem": In("true")},     # The requirement should be preferentially met.
)
```

The four operators are as follows.

Operator	Syntax	Description
In	{"pool": In("gpu", "npu")}	The label value matches one of the specified values.
NotIn	{"env": NotIn("test")}	The label value is not within the specified range.
Exists	{"npu_type": Exists()}	The label key exists.
DoesNotExist	{"maintenance": DoesNotExist()}	The label key does not exist.

Scheduling in the Multi-Workgroup Scenario

Devices of the same type (such as GPUs) in a cluster may be divided into multiple workgroups (resource pools). Each workgroup is identified by the **endpointId** tag key on the node, and the tag value is the workgroup name. When there are multiple workgroups, parameters such as **device="GPU"** can schedule actors to GPU nodes, but cannot distinguish which workgroup the nodes belong to.

NOTE

endpointId is the ID of an Aura endpoint. It may vary depending on the cluster. You can view your endpoint ID on the tenant plane.

You can use NodeLabelSchedulingStrategy to specify the target workgroup during runtime:

```
from aura_frame.ibis.expr.operations.scheduling_strategy import (
    NodeLabelSchedulingStrategy, InstanceSchedulingStrategy,
    InstanceAffinityMode, In,
)

# Schedule different instances in the GPU workgroup named "gpu-pool-a".
# The label key is endpointId (obtained from the tenant plane), and the value is the workgroup name.
ds = ds.map(
    fn="my_schema.inference",
    on=["feature_col"],
    as_col="result",
    device="GPU",
    model="A100",
    num_models=1,
    scheduling_strategy=[
        NodeLabelSchedulingStrategy(hard={"<endpointId>": In("gpu-pool-a")}),
        InstanceSchedulingStrategy(mode=InstanceAffinityMode.DIFF_PREFERRED),
    ],
)
```

 Copyright © Huawei Technologies Co., Ltd.

Table 2-33 Workgroup scheduling priorities and coexistence rules

Source	Level	Priority	Description
work_group_name	DDL registration	Highest	Specified in **CREATE FUNCTION** and added as an AND constraint by the kernel using **LabelInOperator(endpointId, {workgroup_name})**
NodeLabelSchedulingStrategy	Runtime call	Medium	Specified in **WITH ARGUMENTS**, and flexible and variable.

Key rules:

- **Coexistence of work_group_name and scheduling_strategy**: **work_group_name** is added to the label list as an AND constraint (the key is **endpointId**) and does not overwrite or clear the **scheduling_strategy** constraints.
- **Explicit selection of multiple workgroups**: If there are multiple workgroups of the same type, you must specify the target using **NodeLabelSchedulingStrategy** or **work_group_name**. Otherwise, the scheduling result is unpredictable.

Table 2-34 Differences between registration and runtime settings

Dimension	work_group_name	scheduling_strategy
Setting time	UDF registration (CREATE FUNCTION)	Query and call (WITH ARGUMENTS)
Flexibility	One-time registration, always effective	Varies depending on the query
Scope	All calls to the UDF	Only the current query
Change cost	UDF re-registration	Code modification
SQL level	DDL	DML

Table 2-35 Selection recommendations

Scenario	Recommendation
The UDF **always** runs in the same workgroup.	Specify **work_group_name** during registration to avoid repeated settings for each query.
The UDF **switches between different workgroups** (gray release and multi-tenancy)	**Do not set work_group_name** during registration. Flexibly specify it using **scheduling_strategy**.

 Copyright © Huawei Technologies Co., Ltd.

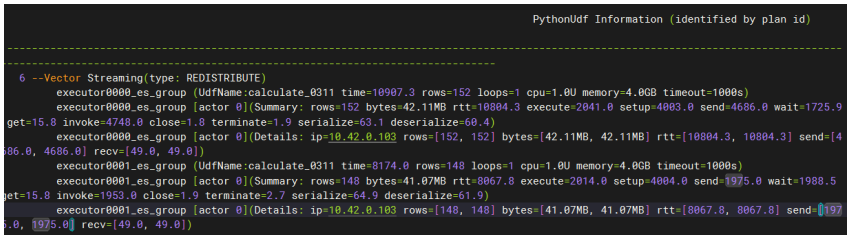
2.1.5.12.14 Python UDF Performance Tuning

Aura provides key runtime performance metrics for monitoring. Based on these metrics, you can fine-tune resource specifications and parallelism to optimize the execution of your Python UDFs.

UDF Performance Monitoring

You can run the **explain performance** command to print the performance metrics of the UDF actor. As illustrated in **Figure 2-2**, it gathers statistics on the performance of the UDF actor associated with the function **calculate_0311** across each DN. The following uses three lines of performance data as an example.

Figure 2-2 Performance metrics of Python UDF actors



- In **executor0000_es_group (UdfName:calculate_0311 time=10907.3 rows=152 loops=1 cpu=1.0U memory=4.0GB timeout=1000s)**:
  - **executor0000_es_group** indicates the DN ID.
  - **UdfName** indicates the name of the UDF to be executed.
  - **time** indicates the total runtime of the UDF.
  - **rows** indicates the amount of data processed on this DN.
  - **loops** indicates the number of times that the UDF was called.
  - **cpu** indicates the CPU size requested by the actor during UDF execution, measured in U.
  - **memory** indicates the memory size requested by the actor during UDF execution, measured in GB.
  - **timeout** indicates the maximum execution duration for a single batch of the actor during UDF execution, measured in seconds. Exceeding this limit will result in an error.
- **executor0000_es_group [actor 0](Summary: rows=152 bytes=42.11MB rtt=10804.3 execute=2041.0 setup=4003.0 send=4686.0 wait=1725.9 get=15.8 invoke=4748.0 close=1.8 terminate=1.9 serialize=63.1 deserialize=60.4)** provides detailed information about the 0th UDF actor, as outlined below:
  - **rows** indicates the amount of data processed on this actor.
  - **bytes** indicates the size of data handled by this actor.
  - **rtt** indicates the end-to-end time from when the kernel sends data to the actor until the actor receives and processes it.

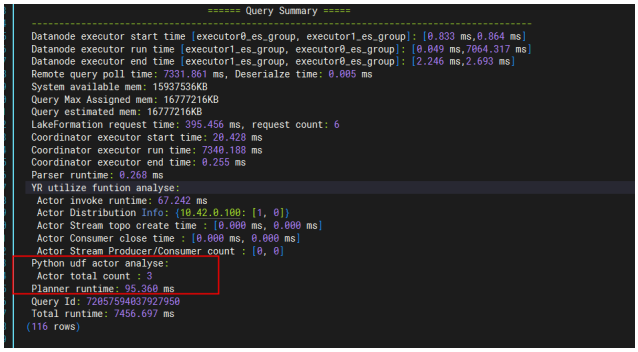
 Copyright © Huawei Technologies Co., Ltd.

  - **execute** indicates the effective computation time of the actor.
  - **setup** indicates the time spent preparing the environment and loading the UDF code post-initialization.
  - **send** indicates the time taken to transmit data from the kernel to the actor.
  - **wait** indicates the delay while the kernel awaits data returns from the actor.
  - **get** indicates the successful retrieval time of actor data by the kernel.
  - **invoke** indicates the time required for starting an actor.
  - **close** indicates the time when the actor closes the computation.
  - **terminate** indicates the time required for the data system to bring the actor offline.
  - **serialize** indicates the time when data undergoes serialization.
  - **deserialize** indicates the time taken for data deserialization.
- In **executor0000_es_group [actor 0](Details: ip=10.42.XX.XX rows=[152, 152] bytes=[42.11MB, 42.11MB] rtt=[10804.3, 10804.3] send=[4686.0, 4686.0] recv=[49.0, 49.0])**:
  - **ip** indicates the IP address of the actor.
  - **rows** indicates the minimum and maximum numbers of rows in all batches executed by the actor.
  - **bytes** indicates the minimum and maximum sizes of data in all batches executed by the actor.
  - **rtt** indicates the minimum and maximum end-to-end execution times of all batches performed by the actor.
  - **send** indicates the minimum and maximum values of data sent from the kernel to the actor in all batches executed by the actor.
  - **recv** indicates the minimum and maximum values of data received by the kernel from Python across all batches executed by the actor.

As illustrated in the figure below, Query Summary displays the summary information about all UDF actors in the query.

Actor total count indicates the total number of actors invoked by the current query.

Figure 2-3 Overview of data stream transmission information of Python UDF actors



 Copyright © Huawei Technologies Co., Ltd.

UDF Pushdown Optimization

If a function returns records, the function cannot be pushed down to DNs, and the following message is displayed: "Function xxx can not be shipped because return record."

Example:

```
SELECT jsonb_each(data::jsonb) FROM table1;
```

If pushdown is required, do not use functions that return records. Instead, use functions that return desired values.

```
SELECT  data::jsonb ->>'car_model' AS car_model, data::jsonb ->>'color' AS color FROM  table1;
```

2.1.5.13 Data Types

2.1.5.13.1 Images

Images are processed internally using the Pillow library. By obtaining a Python Imaging Library (PIL) image object, you can process images through the PIL. Supported image formats include JPEG and PNG.

Table 2-36 Properties

Property	Type	Description
filename	str	File name or path of the image (if loaded from a file).
format	str	Image format (e.g. PNG, JPEG).
height	int	Image height in pixels.
width	int	Image width in pixels.
data	bytes	Image data.

Table 2-37 Methods

Method	Parameter	Return Value	Description
__init__	• filename: Union[str, os.PathLike] = None • data: Union[str, PIL.Image] = None • format: Optional[str] = None	None	Constructor: Loads an image from a file (**filename**) or data (**data**), supporting bytes or Pillow objects.
to_numpy	-	np.ndarray	Converts the image to a NumPy array (float32).

 Copyright © Huawei Technologies Co., Ltd.

Method	Parameter	Return Value	Description
resize	new_shape: tuple[int, int]	Image	Returns a new resized image based on the specified dimensions.
show	-	None	Displays the image using Pillow's default viewer.
to_thumbnail	size: tuple[int, int] = (100, 100)	Image	Generates a thumbnail with the specified size.
crop	box: tuple[float, float, float, float]	Image	Crops a specified area in an image and returns a new image.
getbands	-	tuple: [str, ...]	Calls Pillow's **getbands()**.
getmode	-	str	Returns the image mode.
to_struct	-	dict	Returns the structured image data.

2.1.5.13.2 Videos

Video data is processed internally using PyAV.

- Supported formats: MP4, MKV, WebM, AVI, MOV, FLV, M4V, M4A, 3GP, 3G2, and TS
- Supported codecs: AVC1, AVC3, H.264, HVC1, H.265, HEVC, VP9, VP8, and MP4V

The video object includes specific attributes and methods for further processing.

Table 2-38 Attributes

Attribute	Type	Description
filename	str	Path of the video file (if loaded from a file).
format	str	Video format (e.g., mp4, avi).
data	bytes	Video data.

 Copyright © Huawei Technologies Co., Ltd.

Table 2-39 Methods

Method	Parameter	Return Value	Description
__init__	filename: Union[str, os.PathLike] = None,data: Union[str, bytes] = None	None	Constructor: Loads video from a file or binary data.
truncate	**start_time**: float (start time, in seconds) **end_time**: float (end time, in seconds) **shape**: Optional[tuple[int, int]] (target resolution) **fps: float** = None (target frame rate) **codec: str** = "avc1" (encoding format) **format: str** = "mp4" (output format)	Video	Trims a video segment, allowing adjustments to resolution, frame rate, and encoding format. Returns a new video object.
write	filename: str, codec: str	None	Saves the video to a file.
convert_format	format	Video	Converts the video to another one in a different format.

2.1.5.13.3 Audio

Audio data is processed internally with the PyAV library. Supported audio formats include MP3, AIFF, FLAC, WAV, WMA, M4A, and AAC. The audio object also provides various properties and methods for manipulation.

Table 2-40 Properties

Property	Type	Description
filename	str	File name or path of the audio (if loaded from a file).

 Copyright © Huawei Technologies Co., Ltd.

Property	Type	Description
format	str	Audio format (for example, WAV or MP3).
subtype	str	Audio encoding.
data	bytes	Audio data.

Table 2-41 Methods

Method	Parameter	Return Value	Description
__init__	filename: Union[str, os.PathLike] = None data: Union[str, bytes, sf.SoundFile] = None	None	Constructor: Loads audio from a file (**filename**) or data (**data**), supporting binary data or soundfile objects.
truncate	start_time: float = 0 end_time: Optional[float] = None	Audio	Extracts a segment of the audio (in seconds) and returns a new audio object.
convert_format	format: str subtype: Optional[str] = None	Audio	Converts the audio to a specified format (for example, from WAV to MP3) and returns a new audio object.
write	filename: str	None	Saves the audio to the specified file (format is automatically inferred).
split	interval: int	list[Audio]	Splits the audio into a list of segments based on a specified interval (in seconds).

 Copyright © Huawei Technologies Co., Ltd.

2.1.5.13.4 Vectors

The EmbeddingVector class is encapsulated in vectors. This class object represents the one-dimensional vector type.

Table 2-42 Properties

Property	Type	Description
vector	list[float]	One-dimensional vector list.
dims	int	Vector dimensionality.

Table 2-43 Methods

Method	Parameter	Return Value	Description
__init__	obj: dict	None	Constructor: Initializes a vector object from a dictionary object.
to_struct	None	dict	Returns a dictionary object that contains the vector and dimension of the current vector class object.
from_struct	cls, obj:dict	EmbeddingVector	Inputs a dictionary and calls the class method to return a class object.

Creating a Vector

The following example demonstrates how to create a vector type.

```
from aura_frame.multimodal.types import EmbeddingVector

vec_dict = {"vector": [1, 2, 3], "dims": 3}
vec = EmbeddingVector(vec_dict)
print(vec.vector)
print(vec.dims)
```

2.1.6 Spark SDK (Python)

Copyright © Huawei Technologies Co., Ltd.

2.1.6.1 Obtaining and Initializing the Spark SDK

The Spark SDK encapsulates the Spark APIs provided by the AI DataLake service to simplify development. You can directly call the API functions provided by the SDK to submit Spark SQL and Spark jobs. This section describes how to obtain and initialize the Spark SDK.

Prerequisites

You have completed the operations in **Python Environment Configuration**.

Step 1: Obtaining the SDK

Download the SDK file. You are advised to download the **.sha256** file as well, which can be used to verify the integrity of the SDK package. For details about the verification method, see **Step 2: Verifying the Integrity of the SDK Package**.

- **SDK installation package**
- **Verification file (SHA-256)**

Step 2: Verifying the Integrity of the SDK Package

When downloading the SDK installation package, you also need to download the **.sha256** file to verify the integrity of the SDK package. This section outlines the steps to use the **.sha256** file for verifying the integrity of the SDK package.

1. **Querying the SHA-256 hash value of the SDK installation package**
   - **On Windows**
     i. Search for **cmd** in the **Start** menu to open the command window.
     ii. Run **cd** to go to the directory where the SDK installation package is saved.

     For example, if the SDK installation package is saved in the **SDK** folder on drive D, enter **cd D:\SDK**.
     iii. Run the following command:
     ```
     certutil -hashfile <SDK installation package name> SHA-256
     ```
     Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **aidatalake-python-sdk-1.0.0-261027.tar.gz**.
     iv. After the command is executed, the command prompt will display the SHA-256 hash value of the SDK installation package.
   - **On Linux**
     i. Run **cd** to go to the directory where the SDK installation package is saved.

     For example, if the SDK installation package is saved in the **SDK** folder in the home directory, enter **cd ~/SDK**.
     ii. Enter the query command:
     ```
     sha256sum <SDK installation package name>
     ```
     Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **aidatalake-python-sdk-1.0.0-261027.tar.gz**.

 Copyright © Huawei Technologies Co., Ltd.

iii. After the command is executed, the terminal will display the SHA-256 hash value of the SDK installation package.

2. **Comparing hash values**

   a. Open the downloaded **.sha256** file using the text editor.

   b. Find the SHA-256 hash value of the SDK installation package in the **.sha256** file.

      It is typically a long string, for example, **a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6***xxxxxxxxxxxxxxxxxxxxxxxx*.

   c. Compare the SHA-256 hash value in the SDK installation package obtained in **Step 1** with the hash value in the **.sha256** file.

      - If the two hash values are identical, the SDK installation package was not altered during the download process and the file is intact.
      - If the hash values are inconsistent, an error occurred during the download of the SDK installation package or that the file was maliciously modified.

        In this case, you are advised to download the SDK installation package and its corresponding **.sha256** file again and then verify the package.

Step 3: Installing the SDK

1. Click **aidatalake-python-sdk-x.x.x.tar.gz** to download the SDK package.
2. Decompress the SDK installation package.
   ```
   tar -xzf aidatalake-python-sdk-x.x.x.tar.gz
   ```
3. Open the CLI, decompress the package, and navigate to the corresponding directory to install the SDK.
   ```
   cd aidatalake-python-sdk-x.x.x
   python setup.py install
   ```

Step 4: Initializing the Client

To use the AI DataLake SDK to access AI DataLake, you need to initialize the AI DataLake client. You can use the AK/SK authentication mode to initialize the client.

```
from huaweicloudsdkaidatalake.v2.aidatalake_client import AIDataLakeClient

client = AidatalakeClient(
    region="cn-north-4",
    project_id="xxxx-xxxx",
    workspace_id="xxx-xxxx",
    ak="xxxx",
    sk="xxxx",
    security_token="xxx",
    endpoint="{regionName}.myhuaweicloud.com",
    obs_endpoint="obs.{regionName}.myhuaweicloud.com",
    endpoint_name="spark-job/sql-xxx"
)
```

 Copyright © Huawei Technologies Co., Ltd.

Table 2-44 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
region	Yes	None	Region name.
project_id	Yes	None	Project ID.
workspace_id	Yes	None	Workspace ID.
ak	Yes	None	Access key ID.
sk	Yes	None	Secret key.
security_token	No	None	Used when a temporary credential is used.
endpoint	Yes	None	AI DataLake URL.
obs_endpoint	No	None	OBS endpoint. If this parameter is not set, **obs.{RegionName}.myhuaweicloud.com** is used by default.
endpoint_name	No	None	AI DataLake endpoint, for example, **fabric.{regionName}.myhuaweicloud.com**.

2.1.6.2 Submitting a Spark SQL Job

Description

This section describes how to submit a SQL job using AI DataLake SDK.

Prerequisites

- The Python SDK environment has been configured and initialized. For details, see **Obtaining and Initializing the Spark SDK**.
- Permissions related to APIs and SQL jobs have been configured. For details, see **Permissions and Supported Actions**.
- The compute resources and endpoints in use are available.

Functions

AidatalakeClient

```
# Synchronous submission function
execute_sql();

# Asynchronous submission function
async_execute_sql();
```

Sample Code

- Asynchronous SQL submission

```
try:
    sql_job = client.async_execute_sql(
```

 Copyright © Huawei Technologies Co., Ltd.

```
        "SELECT 1 AS value",
        database_name="database_name",
        endpoint_name="endpoint_name",
        catalog="catalog_name",
    )
    logger.info("Job submitted. Statement ID: %s", sql_job.job_id)
    # Poll for completion
    status, msg = sql_job.get_status()
    logger.info("Initial status: %s", status)
except Exception as e:
    logger.error("Example 2 FAILED: %s", e)
```

- Synchronous SQL submission

```
try:
    sql_job = client.execute_sql(
        "SHOW DATABASES",
        database_name="database_name",
        endpoint_name="endpoint_name",
        catalog="catalog_name",
    )
    logger.info("Job completed. ID: %s, Status: %s", sql_job.job_id, sql_job.job_status)
except Exception as e:
    logger.error("Example 1 FAILED: %s", e)
```

- SQL query

```
try:
    job = client.new_sql_job_builder("SELECT * FROM demo_db1.demo_tbl1 LIMIT 10") \
        .catalog_name(catalog_name) \
        .database_name(database_name) \
        .labels([{"key": "env", "value": "dev"}]) \
        .running_timeout(300) \
        .queue_timeout(60) \
        .job_timeout(600) \
        .build()
    # Submit and wait
    job.submit()
    logger.info("Job completed. ID: %s", job.job_id)
    result = job.get_result()
    if result:
        row_count = sum(1 for _ in result)
        logger.info("Rows: %d", row_count)
        if hasattr(result, 'close'):
            result.close()
except Exception as e:
    logger.error("Example 3 FAILED: %s", e)
```

Parameters

Table 2-45 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
database_name	Yes	None	Default LakeFormation database name.
endpoint_name	Yes	None	SparkSQL endpoint name.
catalog	Yes	None	Default LakeFormation catalog name.

 Copyright © Huawei Technologies Co., Ltd.

2.1.6.3 Submitting a Spark Job

Description

This section describes how to submit a Spark job using AI DataLake SDK.

Prerequisites

- The Python SDK environment has been configured and initialized. For details, see **Obtaining and Initializing the Spark SDK**.
- Permissions related to APIs and SQL jobs have been configured. For details, see **Permissions and Supported Actions**.
- The compute resources and endpoints in use are available.

Functions

- SparkJarJob

```
# Synchronous submission function
submit_and_wait();

# Asynchronous submission function
submit();
```

- SparkPythonJob

```
# Synchronous submission function
submit_and_wait();

# Asynchronous submission function
submit();
```

- SparkSqlScriptingJob

```
# Synchronous submission function
submit_and_wait();

# Asynchronous submission function
submit();
```

Sample Code

- spark_jar_job

```
try:
    jar_job = client.new_jar_job_builder(
        "test-jar-job",
        "org.example.WordCount",
        "obs://obs-bucket/spark-jar-test-1.0-SNAPSHOT.jar",
    ) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name") \
        .main_args(["500000"]) \
        .job_agency("job_agency") \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
        .executor_number(2) \
        .labels([{"key": "env", "value": "dev"}]) \
        .spark_config({
            "spark.dynamicAllocation.enabled": "true",
            "spark.dynamicAllocation.maxExecutors": "3",
        }) \
        .build()
    # Submit asynchronously, then wait
    jar_job.submit_and_wait()
    logger.info("JAR job completed. ID: %s, Status: %s",
                jar_job.job_id, jar_job.status)
```

 Copyright © Huawei Technologies Co., Ltd.

```
except AidatalakeException as e:
    logger.error("Example 1 FAILED: %s", e)
```

Table 2-46 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
endpoint_name	Yes	None	SparkJob endpoint name.
catalog_name	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
main_args	No	None	Arguments of the **Main** class.
job_agency	No	None	IAM agency for the job, which is used by users to access OBS or Lakeformation.
spark_config	No	None	Spark parameters.
driver_cpu	No	None	Number of millicores in the Driver pod.
driver_memory	No	None	Memory size of the Driver pod, in MB.
executor_cpu	No	None	Number of millicores in the Executor pod.
executor_memory	No	None	Memory size of the Executor pod, in MB.
executor_number	No	None	Number of Executor instances.

- spark_python_job

```
try:
    python_job = client.new_python_job_builder(
        "test-python-job",
        "obs://obs-bucket/sleep_1.py",
    ) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name") \
        .main_args(["500000"]) \
        .job_agency("job_agency") \
        .dependency_py_files(["obs://obs-bucket/utils.py"]) \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
        .executor_number(1) \
        .build()
    python_job.submit_and_wait()
    logger.info("Python job completed. ID: %s, Status: %s",
                python_job.job_id, python_job.status)
```

 Copyright © Huawei Technologies Co., Ltd.

```
except AidatalakeException as e:
    logger.error("Example 2 FAILED: %s", e)
```

Table 2-47 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
endpoint_name	Yes	None	SparkJob endpoint name.
catalog_name	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
main_args	No	None	Arguments of the **Main** class.
job_agency	No	None	IAM agency for the job, which is used by users to access OBS or Lakeformation.
dependency_py_files	No	None	Dependent Python files.
driver_cpu	No	None	Number of millicores of the Driver pod.
driver_memory	No	None	Memory size of the Driver pod, in MB.
executor_cpu	No	None	Number of millicores of the Executor pod.
executor_memory	No	None	Memory size of the Executor pod, in MB.
executor_number	No	None	Number of Executor instances.

- spark_sql_scripting_job

```
try:
    sql_scripting_job = client.new_sql_scripting_job_builder(
        "test-sql-scripting-job",
    ) \
        .sql_scripting_file("obs://obs-bucket/test_sql.sql") \
        .sql_scripting_result_to_obs(True) \
        .endpoint_name("endpoint_name") \
        .catalog_name("catalog_name_sql") \
        .job_agency("job_agency") \
        .driver_cpu(1000).driver_memory(4096) \
        .executor_cpu(1000).executor_memory(4096) \
        .executor_number(1) \
        .build()
    sql_scripting_job.submit_and_wait()
    logger.info("SQL Scripting job completed. ID: %s, Status: %s",
                sql_scripting_job.job_id, sql_scripting_job.status)
    # Read results from OBS
```

 Copyright © Huawei Technologies Co., Ltd.

```
    result = sql_scripting_job.get_result()
    row_count = 0
    for row in result:
        row_count += 1
        if row_count <= 10:
            logger.info("  Row %d: %s", row_count, row)
    logger.info("Total rows: %d", row_count)
    if hasattr(result, 'close'):
        result.close()
except AidatalakeException as e:
    logger.error("Example 3 FAILED: %s", e)
```

Table 2-48 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
sql_scripting_file	Yes	None	OBS path to the SQL scripting file.
endpoint_name	Yes	None	SparkJob endpoint name.
catalog_name	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
job_agency	No	None	IAM agency for the job, which is used by users to access OBS or Lakeformation.
driver_cpu	No	None	Number of millicores of the Driver pod.
driver_memory	No	None	Memory size of the Driver pod, in MB.
executor_cpu	No	None	Number of millicores of the Executor pod.
executor_memory	No	None	Memory size of the Executor pod, in MB.
executor_number	No	None	Number of Executor instances.

2.1.6.4 Connecting to the Database and Querying Data Using Spark

Description

This section describes how to use AI DataLake DB-API to connect to a database and query data.

 Copyright © Huawei Technologies Co., Ltd.

Prerequisites

- The Python SDK environment has been configured and initialized. For details, see **Obtaining and Initializing the Spark SDK**.
- Permissions related to APIs and SQL jobs have been configured. For details, see **Permissions and Supported Actions**.
- The compute resources and endpoints in use are available.

Functions

cursor

```
# Synchronous submission function
execute();

# Function for retrieving SQL query results
fetchall();
```

Sample Code

- Creating a connection

```
conn = connect(
    region="your_region",
    project_id="your_project_id",
    workspace_id="your_workspace_id",
    ak="your_access_key",
    sk="your_secret_key",
    endpoint="aidatalake_endpoint",
    database="default",
)
```

- Executing SQL statements

```
try:
    cursor = conn.cursor()
    cursor.execute("SELECT 1 AS value, 'hello' AS greeting")
    row = cursor.fetchone()
    logger.info("Row: %s", row)
    cursor.close()
```

```
try:
    cursor = conn.cursor()
    cursor.execute(
        "SELECT * FROM demo_table WHERE id = ? AND name = ?",
        [42, "test"],
    )
    rows = cursor.fetchall()
    logger.info("Fetched %d rows", len(rows))
    cursor.close()
```

```
cursor = conn.cursor()

# Retrieve the query result.
cursor.execute(
    "SELECT * FROM demo_table WHERE id = :id AND name = :name",
    {"id": 42, "name": "test"},
)
for row in cursor:
    logger.info("Row: %s", row)

# List catalogs
catalogs = conn.list_catalogs()
logger.info("Catalogs: %s", catalogs)

# List schemas (databases)
schemas = conn.list_schemas(catalog="default")
logger.info("Schemas in 'default': %s", schemas)
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-49 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
region	Yes	None	Region name.
project_id	Yes	None	Project ID.
ak	Yes	None	Access key ID.
sk	Yes	None	Secret key.
endpoint	Yes	None	URL of the AI DataLake.
database	No	None	Name of the LakeFormation database.
workspace_id	Yes	None	Workspace ID.
catalog	No	None	LakeFormation catalog name.
security_token	No	None	Used when a temporary credential is used.
obs_endpoint	No	None	OBS endpoint. If this parameter is not set, **obs.{RegionName}.myhuaweicloud.com** is used by default.
endpoint_name	No	None	AI DataLake endpoint. Example: **fabric.{regionName}.myhuaweicloud.com**

2.2 Java SDKs

2.2.1 Configuring the Java Environment

Scenario

Before installing and using Java SDKs, make sure you have completed the basic configuration of your development environment.

Java SDKs require JDK 1.8 or later. To ensure compatibility with future versions, you are advised to use version 1.8.

Once the Java runtime environment is configured, you can open the Windows CLI and execute the command **Java -version** to check the version information.

Procedure

1. Install the JDK. Download the JDK 1.8 installation package from **Oracle official website** and install it.

 Copyright © Huawei Technologies Co., Ltd.

2. Set environment variables. Specifically, choose **Start** > **Control Panel**. Click **System** > **Advanced system settings**. In the displayed **System Properties** dialog box, click **Advanced** and then click **Environment Variables** to switch to the **Environment Variables** dialog box.
3. Create the system variable **JAVA_HOME** with **Value** set to the JDK installation path, for example, **D:\Java\jdk1.8.0_45**.
4. Edit the **Path** variable and add **%JAVA_HOME%\bin** to **Variable value**.
5. Create the **CLASSPATH** variable with **Variable value** set to **.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar**.
6. Check whether the configurations succeed. Specifically, at the **Start** menu, enter **cmd** and press **Enter** to bring up the command prompt window. Enter ***java -version*** and press **Enter**. If the version information is displayed, as shown in **Figure 2-4**, Python is successfully installed and configured.

Figure 2-4 Checking the configuration

```
C:\Users\          >java -version
java version "1.8.0_172"
Java(TM) SE Runtime Environment (build 1.8.0_172-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)
```

2.2.2 Spark SDK (Java)

2.2.2.1 Obtaining and Initializing the Spark SDK

The Spark SDK encapsulates the Spark APIs provided by the AI DataLake service to simplify development. You can directly call the API functions provided by the SDK to submit Spark SQL and Spark jobs. This section describes how to obtain and initialize the Spark SDK.

Prerequisites

You have completed the operations in **Configuring the Java Environment**.

Step 1: Obtaining the SDK

Download the SDK file. You are advised to download the **.sha256** file as well, which can be used to verify the integrity of the SDK package. For details about the verification method, see **Step 2: Verifying the Integrity of the SDK Package**.

- **SDK installation package**
- **Verification file (SHA-256)**

Step 2: Verifying the Integrity of the SDK Package

When downloading the SDK installation package, you also need to download the **.sha256** file to verify the integrity of the SDK package. This section outlines the steps to use the **.sha256** file for verifying the integrity of the SDK package.

1. **Querying the SHA-256 hash value of the SDK installation package**

 Copyright © Huawei Technologies Co., Ltd.

- **On Windows**
  i. Search for **cmd** in the **Start** menu to open the command window.
  ii. Run **cd** to go to the directory where the SDK installation package is saved.
  For example, if the SDK installation package is saved in the **SDK** folder on drive D, enter **cd D:\SDK**.
  iii. Run the following command:
  ```
  certutil -hashfile <SDK installation package name> SHA-256
  ```
  Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **aidatalake-spark-sdk-1.0-261030.tar.gz**.
  iv. After the command is executed, the command prompt will display the SHA-256 hash value of the SDK installation package.
- **On Linux**
  i. Run **cd** to go to the directory where the SDK installation package is saved.
  For example, if the SDK installation package is saved in the **SDK** folder in the home directory, enter **cd ~/SDK**.
  ii. Enter the query command:
  ```
  sha256sum <SDK installation package name>
  ```
  Replace *<SDK installation package name>* with the actual SDK installation package name, for example, **aidatalake-spark-sdk-1.0-261030.tar.gz**.
  iii. After the command is executed, the terminal will display the SHA-256 hash value of the SDK installation package.

2. **Comparing hash values**
   a. Open the downloaded **.sha256** file using the text editor.
   b. Find the SHA-256 hash value of the SDK installation package in the **.sha256** file.
   It is typically a long string, for example, **a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6***xxxxxxxxxxxxxxxxxxxxxxxx*.
   c. Compare the SHA-256 hash value in the SDK installation package obtained in **Step 1** with the hash value in the **.sha256** file.
      - If the two hash values are identical, the SDK installation package was not altered during the download process and the file is intact.
      - If the hash values are inconsistent, an error occurred during the download of the SDK installation package or that the file was maliciously modified.
      In this case, you are advised to download the SDK installation package and its corresponding **.sha256** file again and then verify the package.

Step 3: Installing the SDK

1. Click **aidatalake-spark-sdk-x.x.x.jar** to download the SDK package.

 Copyright © Huawei Technologies Co., Ltd.

2. Run the following command to install the JAR package to the local Maven repository:

```
mvn install:install-file -Dfile=aidatalake-spark-sdk-x.x.x.jar -DgroupId=com.huawei.aidatalake -
DartifactId=aidatalake-spark-sdk -Dversion=x.x.x -Dpackaging=jar
```

3. Add the following dependencies to the **pom.xml** file:

```
<dependency>
    <groupId>com.huawei.aidatalake</groupId>
    <artifactId>aidatalake-spark-sdk</artifactId>
    <version>x.x.x</version>
</dependency>
```

Step 4: Initializing the Client

To use the AI DataLake SDK to access AI DataLake, you need to initialize the AI DataLake client. You can use the AK/SK authentication mode to initialize the client.

```
// Initialize the client context.
JobClientContext jobContext = new JobClientContext();
jobContext.setRegionName("cn-north-4");
jobContext.setProjectId("xxxx-xxxx");
jobContext.setWorkspaceId("xxx-xxxx");
jobContext.setLakeformationInstanceId("xxxx-xxxx");
jobContext.setEndpointName("xxxx");
// Use the STS token.
jobContext.setCredentialProvider(StaticCredentialProvider.create("{accesskey}", "{secretKey}",
"{securityToken}", "{projectId}"));
jobClientContext.setAiDataLakeEndpoint("xxx.xxxx");
jobClientContext.setLakeformationInstanceId("xxx-xxxx");
jobClientContext.setObsEndpoint("xxx.xxxx");

// Initialize AIDataLakeSparkSqlClient (Spark SQL client).
AIDataLakeSparkSqlClient sparkSqlClient = new AIDataLakeSparkSqlClient(jobContext);

// Initialize AIDataLakeSparkJobClient (Spark Job client).
AIDataLakeSparkJobClient sparkJobClient = new AIDataLakeSparkJobClient(jobContext);

// Initialize AIDataLakeLakeFormationClient (Schema metadata query client).
AIDataLakeLakeFormationClient lakeformationClient = new AIDataLakeLakeFormationClient(jobContext);
```

Table 2-50 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
RegionName	Yes	None	Region name.
ProjectId	Yes	None	Project ID.
WorkspaceId	Yes	None	Workspace ID.
LakeformationInstanceId	Yes	None	ID of the LakeFormation instance bound to the workspace.
AccessKey	Yes	None	Access key ID.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Description
SecretKey	Yes	None	Secret key.
SecurityToken	No	None	Used when a temporary credential is used.
EndPointName	Yes	None	AI DataLake endpoint, for example, **fabric.{regionName}.myhuaweicloud.com**.
ObsEndpoint	No	None	OBS endpoint. If this parameter is not set, **obs.{RegionName}.myhuaweicloud.com** is used by default.

2.2.2.2 Submitting a Spark SQL Job

Description

This section describes how to submit a SQL job using AI DataLake SDK.

Prerequisites

- The Java SDK environment has been configured and initialized. For details, see **Obtaining and Initializing the Spark SDK**.
- Permissions related to APIs and SQL jobs have been configured. For details, see **Permissions and Supported Actions**.
- The compute resources and endpoints in use are available.

Functions

SparkSqlJob

```
# Synchronous submission function
submit();

# Asynchronous submission function
asyncSubmit();
```

Sample Code

- Asynchronous SQL submission

```
public static void asyncSubmit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException
{
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                .catalogName(catalogName)
```

 Copyright © Huawei Technologies Co., Ltd.

```
                .databaseName(databaseName)
                .sparkConfig(sparkConfig)
                .build();
    sqlJob.asyncSubmit();
}
```

- Synchronous SQL submission

```
public static void submit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException {
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                .catalogName(catalogName)
                .databaseName(databaseName)
                .sparkConfig(sparkConfig)
                .build();
    sqlJob.submit();
}
```

- SQL query

```
public static void submit(AIDataLakeSparkSqlClient sparkSqlClient) throws AIDataLakeException {
    String databaseName = "DATABASE_NAME";
    String catalogName = "CATALOG_NAME";
    String sqlStatement = "SELECT xxx";
    String statementId;
    Map<String, String> sparkConfig = new HashMap<>();
    // spark conf starting with "spark."
    sparkConfig.put("spark.sql.shuffle.partitions", "200");
    SparkSqlJob sqlJob = sparkSqlClient.newSparkSqlJobBuilder(sqlStatement)
                .catalogName(catalogName)
                .databaseName(databaseName)
                .sparkConfig(sparkConfig)
                .build();
    sqlJob.submit();
    statementId = sqlJob.getStatementId();
    ResultSet resultSet = sparkSqlClient.getResultsWithStatementId(statementId).getResultSet();
    while (resultSet.hasNext()) {
        Row row = resultSet.read();
        List<Column> schema = row.getSchema();
        List<Object> record = row.getRecord();
        for (int i = 0; i < schema.size(); i++) {
            DataType.TypeName typeName =
DataType.TypeName.fromName(schema.get(i).getType().getName());
            switch (typeName) {
                case STRING:
                    String strV = (String) record.get(i);
                    break;
                case DATE:
                    Date dtV = (Date) record.get(i);
                    break;
            }
        }
    }
}
```

 Copyright © Huawei Technologies Co., Ltd.

Parameters

Table 2-51 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
databaseName	Yes	None	Default LakeFormation database name.
catalogName	Yes	None	Default LakeFormation catalog name.
sqlStatement	Yes	None	SQL statement to be submitted.

2.2.2.3 Submitting a Spark Job

Description

This section describes how to submit a Spark job using AI DataLake SDK.

Prerequisites

- The compute resources and endpoints in use are available.
- Permissions related to APIs and SQL jobs have been configured. For details, see **Permissions and Supported Actions**.

Functions

- SparkJarJob

```
# Synchronous submission function
submit();

# Asynchronous submission function
asyncSubmit();
```

- SparkSqlScriptingJob

```
# Synchronous submission function
submit();

# Asynchronous submission function
asyncSubmit();
```

- SparkPythonJob

```
# Synchronous submission function
submit();

# Asynchronous submission function
asyncSubmit();
```

Sample Code

- Synchronously submitting a SparkJar job

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
```

 Copyright © Huawei Technologies Co., Ltd.

```
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";
    // set spark configs
    Map<String, String> config = new HashMap<>();
    config.put("key", "value");

    SparkJarJob sparkJob = sparkJobClient.newSparkJarJobBuilder(jobName, "com.example.Main",
"obs://app.jar")
                    .endpointName(endpointName)
                    .jobAgency("agency")
                    .catalogName(catalogName)
                    .mainArgs(Arrays.asList("arg1"))
                    .dependencyJars(Arrays.asList("obs://lib.jar"))
                    .sparkConfig(config)
                    .driverCpu(2000)
                    .driverMemory(4096)
                    .executorCpu(4000)
                    .executorMemory(8192)
                    .executorNumber(4)
                    .build();
    sparkJob.submit();
}
```

Table 2-52 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
catalogName	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
endpointName	Yes	None	SparkSQL endpoint name.
mainArgs	No	None	Arguments of the **Main** class.
dependencyJars	No	None	Dependent JAR files.
jobAgency	No	None	IAM agency for the job, which is used by users to access OBS or Lakeformation.
sparkConfig	No	None	Spark parameters.
driverCpu	No	None	Number of millicores of the Driver pod.
driverMemory	No	None	Memory size of the Driver pod, in MB.
executorCpu	No	None	Number of millicores of the Executor pod.
executorMemory	No	None	Memory size of the Executor pod, in MB.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Description
executorNumber	No	None	Number of Executor instances.

- Asynchronously submitting a SparkSqlScript job

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";
    // set spark configs
    Map<String, String> config = new HashMap<>();
    config.put("key", "value");

    SparkSqlScriptingJob sparkScriptingJob = sparkJobClient.newSparkSqlScriptingJobBuilder(jobName)
                .endpointName(endpointName)
                .sqlScriptingFileUri("obs://xxx")
                .catalogName(catalogName)
                .jobAgency("agency")
                .sparkConfig(config)
                .driverCpu(2000)
                .driverMemory(4096)
                .executorCpu(4000)
                .executorMemory(8192)
                .executorNumber(4)
                .sqlScriptingResultToObs(true)
                .build();
    sparkJob.asyncSubmit();
}
```

Table 2-53 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
catalogName	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
endpointName	Yes	None	SparkSQL endpoint name.
jobAgency	No	None	IAM agency for the job, which is used by users to access OBS or Lakeformation.
sqlScriptingFileUri	Yes	None	OBS path to the SQL scripting file.
sparkConfig	No	None	Spark parameters.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Description
driverCpu	No	None	Number of millicores of the Driver pod.
driverMemory	No	None	Memory size of the Driver pod, in MB.
executorCpu	No	None	Number of millicores of the Executor pod.
executorMemory	No	None	Memory size of the Executor pod, in MB.
executorNumber	No	None	Number of Executor instances.

- Synchronously submitting a SparkPython job

```
public static void submit(AIDataLakeSparkJobClient sparkJobClient) throws AIDataLakeException {
    String jobName = "JOB_NAME";
    String endpointName = "ENDPOINT_NAME";
    String catalogName = "CATALOG_NAME";

    return sparkJobClient.newSparkPythonJobBuilder(jobName, "obs://xxx/job.py")
            .endpointName(endpointName)
            .catalogName(catalogName)
            .jobAgency("agency")
            .sparkConfig(config)
            .driverCpu(1000)
            .driverMemory(1024)
            .driverDisk(20)
            .executorCpu(1000)
            .executorMemory(1024)
            .build();
    sparkJob.submit();
}
```

Table 2-54 Parameter description

Parameter	Mandatory (Yes/No)	Default Value	Description
catalogName	No	None	Default LakeFormation catalog name. This parameter is optional if **Spark.sql** is executed within the job.
endpointName	Yes	None	SparkSQL endpoint name.

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Description
jobAgency	No	None	IAM agency for the job, which is used by users to access OBS or LakeFormation.
sparkConfig	No	None	Spark parameters.
driverCpu	No	None	Number of millicores of the Driver pod.
driverMemory	No	None	Memory size of the Driver pod, in MB.
executorCpu	No	None	Number of millicores of the Executor pod.
executorMemory	No	None	Memory size of the Executor pod, in MB.
executorNumber	No	None	Number of Executor instances.

2.2.2.4 Querying the Metadata of the Spark Databases and Tables

Description

This section describes how to use AI DataLake SDK to query metadata of databases and tables.

Prerequisites

LakeFormation APIs and related permissions have been configured. For details about the permissions, see **Permissions and Supported Actions**.

Functions

AIDataLakeLakeFormationClient

```
# List catalogs.
listCatalogs();

# List databases.
listDatabases("yourCatalogName");

# List tables.
listTables("yourCatalogName", "yourDBName");
```

 Copyright © Huawei Technologies Co., Ltd.

Parameter	Mandatory (Yes/No)	Default Value	Description
yourCatalogName	No	None	Data catalog name.
yourDBName	Yes	None	Database name.

Sample Code

- Querying catalogs

```
public static void listCatalogs(AIDataLakeLakeFormationClient lakeformationClient) {
    Catalogs catalogs = lakeformationClient.listCatalogs();
    for (Catalog item : catalogs) {
        System.out.println(item.getCatalogName());
    }
    Catalog catalog = catalogs.getCatalog("yourCatalogName")
}
```

- Querying databases

```
public static void listDatabases(AIDataLakeLakeFormationClient lakeformationClient) {
    Databases databases = lakeformationClient.listDatabases("yourCatalogName");
    for (Database item : databases) {
        System.out.println(item.getDatabaseName());
    }
    Database database = databases.getDatabase("yourDBName")
}
```

- Querying tables

```
public static void listTables(AIDataLakeLakeFormationClient lakeformationClient) throws
AIDataLakeException {
    Tables tables = lakeformationClient.listTables("yourCatalogName", "yourDBName");
    for (Table item : tables) {
        System.out.println(item.getTableName());
    }
    // get table
    Table table = tables.getTable("yourTableName");
    // filter table
    TableFilter tableFilter = new TableFilter();
    tableFilter.setTblPattern("table_name_xxx");
    tableFilter.setTblPattern("table_type_xxx");
    Iterable<Table> tableA = tables.iterableWithDatabaseAndFilter("yourDBName", tableFilter);
    // get table meta
    TableMeta tableMeta = tables.getTableMeta("yourTableName");
}
```

 Copyright © Huawei Technologies Co., Ltd.