Cloudera Data Warehouse on premises 1.5.5

Querying Data

Date published: 2020-08-17
Date modified: 2025-11-08



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Query data **4**
View query history **4**
Create User-defined functions **4**
Set up dev environment 5
Create UDF 6
Build project and upload JAR 7
Register UDF 8
Call UDF in a query 8
Simplify queries with User-defined functions **9**
Creating an Impala user-defined function 9
UDF concepts 10
Runtime environment for UDFs 13
Writing UDFs 13
Writing user-defined aggregate functions (UDAFs) 17
Building and deploying UDFs 18
Performance considerations for UDFs 19
Examples of creating and using UDFs 19
Security considerations for UDFs 26
Limitations and restrictions for Impala UDFs 26
Impala workload management **27**
Impala workload management table format 27
Impala workload management table maintenance 31
Impala workload management use cases 31
Enable Impala workload management 31
Impala coordinator Startup flags 32
Running queries on system tables **33**
Configuring the only coordinators request pool 34
Cloudera Data Explorer (Hue) Storage Browser (Technical Preview) **34**
Enabling the new Cloudera Data Explorer (Hue) Storage Browser (Technical Preview) 35
Managing files and operations in Cloudera Data Explorer (Hue) Storage Browser (Technical Preview) 35
Hive query history service **40**
Query history table 40
Query history table format 41
Query history use case 44
Configuring query history service 45

Submit queries with Hue **45**

Querying data in Cloudera Data Warehouse

This topic describes how to query data in your Virtual Warehouse on Cloudera Data Warehouse.

About this task

The Cloudera Data Warehouse service includes the Hue SQL editor that you can use to submit queries to Virtual Warehouses. For example, you can use Hue to submit queries to an Impala Virtual Warehouse.

Procedure

1. Log in to the Cloudera Data Warehouse service as DWUser.

   The **Overview** page is displayed.
2. Click Hue on the Virtual Warehouse tile.
3. Enter your query into the editor and submit it to the Virtual Warehouse.

Viewing query history in Cloudera Data Warehouse

In Cloudera Data Warehouse, you can view all queries that were run against a Database Catalog from Hue, Beeline, Hive Warehouse Connector (HWC), Tableau, Impala-shell, Impyla, and so on.

About this task

You need to set up Query Processor administrators to view the list of all queries from all users, or to restrict viewing of queries.

Procedure

1. Log in to the Cloudera web interface and navigate to the Cloudera Data Warehouse service.
2. In the Cloudera Data Warehouse service, navigate to the **Overview** page.
3. From a Virtual Warehouse, launch Hue.
4. Click on the Jobs icon on the left-assist panel.

   The **Job Browser** page is displayed.
5. Go to the **Queries** tab to view query history and query details.

Related Information

Viewing Hive query details

Viewing Impala query details

Adding Query Processor Administrator users and groups in Cloudera Data Warehouse

Creating a user-defined function in Cloudera Data Warehouse on premises

You export user-defined functionality (UDF) to a JAR from a Hadoop- and Hive-compatible Java project and store the JAR on your cluster. Using Hive commands, you register the UDF based on the JAR, and call the UDF from a Hive query.

Before you begin

- You must have access rights to upload the JAR to your cluster. Minimum Required Role: Configurator (also provided by Cluster Administrator, Full Administrator).
- Make sure Hive on Tez or Hive LLAP is running on the cluster.
- Make sure that you have installed Java and a Java integrated development environment (IDE) tool on the machine, or virtual machine, where you want to create the UDF.

Setting up the development environment

You can create a Hive UDF in a development environment using IntelliJ, for example, and build the UDF. You define the Cloudera Maven Repository in your POM, which accesses necessary JARS hadoop-common-<version>.jar and hive-exec-<version>.jar.

Procedure

1. Open IntelliJ and create a new Maven-based project. Click Create New Project. Select Maven and the supported Java version as the Project SDK. Click Next.
2. Add archetype information.
   For example:
   - GroupId: com.mycompany.hiveudf
   - ArtifactId: hiveudf
3. Click Next and Finish.
   The generated pom.xml appears in sample-hiveudf.
4. To the pom.xml, add properties to facilitate versioning.
   For example:

```
<properties>
   <hadoop.version>TBD</hadoop.version>
   <hive.version>TBD</hive.version>
</properties>
```

5. In the pom.xml, define the repositories.

   Use internal repositories if you do not have internet access.

```
<repositories>
   <repository>
      <releases>
         <enabled>true</enabled>
         <updatePolicy>always</updatePolicy>
         <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
         <enabled>false</enabled>
         <updatePolicy>never</updatePolicy>
         <checksumPolicy>fail</checksumPolicy>
      </snapshots>
      <id>HDPReleases</id>
      <name>HDP Releases</name>
      <url>http://repo.hortonworks.com/content/repositories/releases/</u
rl>
      <layout>default</layout>
   </repository>
   <repository>
      <id>public.repo.hortonworks.com</id>
      <name>Public Hortonworks Maven Repo</name>
      <url>http://repo.hortonworks.com/content/groups/public/</url>
      <snapshots>
```

```
          <enabled>false</enabled>
       </snapshots>
    </repository>
    <repository>
          <id>repository.cloudera.com</id>
          <url>https://repository.cloudera.com/artifactory/cloudera-repos/<
/url>
    </repository>
 </repositories>
```

6. Define dependencies.
   For example:

```
<dependencies>
  <dependency>
       <groupId>org.apache.hive</groupId>
       <artifactId>hive-exec</artifactId>
       <version>${hive.version}</version>
    </dependency>
    <dependency>
       <groupId>org.apache.hadoop</groupId>
       <artifactId>hadoop-common</artifactId>
       <version>${hadoop.version}</version>
    </dependency>
</dependencies>
```

Creating the UDF class

You define the UDF logic in a new class that returns the data type of a selected column in a table.

Procedure

1. In IntelliJ, click the vertical project tab, and expand hiveudf: hiveudf src main . Select the java directory, and on the context menu, select New Java Class , and name the class, for example, TypeOf.
2. Extend the GenericUDF class to include the logic that identifies the data type of a column.
   For example:

```
package com.mycompany.hiveudf;

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.\
PrimitiveObjectInspectorFactory;
import org.apache.hadoop.io.Text;
public class TypeOf extends GenericUDF {
  private final Text output = new Text();
@Override
  public ObjectInspector initialize(ObjectInspector[] arguments) throws U
DFArgumentException {
    checkArgsSize(arguments, 1, 1);
    checkArgPrimitive(arguments, 0);
    ObjectInspector outputOI = PrimitiveObjectInspectorFactory.writableSt
ringObjectInspector;
    return outputOI;
  }

@Override
```

```
  public Object evaluate(DeferredObject[] arguments) throws HiveException
 {
    Object obj;
    if ((obj = arguments[0].get()) == null) {
      String res = "Type: NULL";
      output.set(res);
      } else {
      String res = "Type: " + obj.getClass().getName();
      output.set(res);
    }
    return output;
  }

@Override
  public String getDisplayString(String[] children) {
    return getStandardDisplayString("TYPEOF", children, ",");
  }
}
```

Building the project and uploading the JAR

You compile the UDF code into a JAR and add the JAR to the classpath on the cluster.

About this task

Use the direct reference method to configure the cluster to find the JAR. It is a straight-forward method, but recommended for development only.

Procedure

1. Build the IntelliJ project.

```
...
[INFO] Building jar: /Users/max/IdeaProjects/hiveudf/target/TypeOf-1.0-S
NAPSHOT.jar
[INFO] ------------------------------------------------------------------
-------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
--------
[INFO] Total time: 14.820 s
[INFO] Finished at: 2019-04-03T16:53:04-07:00
[INFO] Final Memory: 26M/397M
[INFO] ------------------------------------------------------------------
--------

Process finished with exit code 0
```

2. In IntelliJ, navigate to the JAR in the /target directory of the project.
3. Configure the cluster so that Hive can find the JAR using the direct reference method.
   a) Upload the JAR to HDFS.
   b) Move the JAR into the Hive warehouse. For example, in Cloudera Base on premises:

```
$ hdfs dfs -put TypeOf-1.0-SNAPSHOT.jar /warehouse/tablespace/managed/hi
veudf-1.0-SNAPSHOT.jar
```

4. In IntelliJ, click Save.
5. Click Actions Deploy Client Configuration .
6. Restart the Hive service.

Registering the UDF

In Cloudera Data Warehouse, you run a command from Hue to make the UDF functional in Hive queries. The UDF persists between HiveServer restarts.

Before you begin

You need to set up UDF access using a Ranger policy as follows:

1. Log in to the Cloudera Data Warehouse service and open Ranger from the Database Catalog associated with your Hive Virtual Warehouse.
2. On the **Service Manager** page, under the HADOOP SQL section, select the Database Catalog associated with the Hive Virtual Warehouse in which you want to run the UDFs.

   The list of policies is displayed.
3. Select the all - database, udf policy and add the users needing access to Hue. To add all users, you can specify {USER}.

About this task

In this task, the registration command differs depending on the method you choose to configure the cluster for finding the JAR. If you use the Hive aux library directory method that involves a symbolic link, you need to restart the HiveServer pod after registration. If you use the direct JAR reference method, you do not need to restart HiveServer. You must recreate the symbolic link after any patch or maintenance upgrades that deploy a new version of Hive.

Procedure

1. Open Hue from the Hive Virtual Warehouse in Cloudera Data Warehouse.
2. Run the registration command by including the JAR location in the command as follows:

   ```
   CREATE FUNCTION udftypeof AS 'com.mycompany.hiveudf.TypeOf01' USING JAR
   'hdfs:///warehouse/tablespace/managed/TypeOf01-1.0-SNAPSHOT.jar';
   ```
3. Restart the HiveServer.

   You can either delete the hiveserver2-0 pod using Kubernetes, or, you can edit an HS2 related configuration, and Cloudera restarts the HiveServer pod.

   Note: If you plan to run UDFs on LLAP, you must restart the query executor and query coordinator pods after registering the UDF.
4. Verify whether the UDF is registered.

   ```
   SHOW FUNCTIONS;
   ```

   You scroll through the output and find default.typeof.

Calling the UDF in a query

After registration of a UDF, you do not need to restart Hive before using the UDF in a query. In this example, you call the UDF you created in a SELECT statement, and Hive returns the data type of a column you specify.

Before you begin

- For the example query in this task, you need to create a table in Hive and insert some data.

  This task assumes you have the following example table in Hive:

  ```
  +------------------+---------------+---------------+
  |  students.name   | students.age  | students.gpa  |
  ```

```
+------------------+--------------+---------------+
| fred flintstone  | 35           | 1.28          |
| barney rubble    | 32           | 2.32          |
+------------------+--------------+---------------+
```

- As a user, you need to have permission to call a UDF, which a Ranger policy can provide.

Procedure

1. Use the database in which you registered the UDF.

```
USE default;
```

2. Query Hive using the direct reference method:

```
SELECT students.name, udftypeof(students.name) AS type FROM students WHERE
 age=35;
```

You get the data type of the name column in the students table:

```
+------------------+-----------------------------------------------------+
|  students.name   |                        type                         |
+------------------+-----------------------------------------------------+
| fred flintstone  | Type: org.apache.hadoop.hive.serde2.io.HiveVarcharWri
table |
+------------------+-----------------------------------------------------+
```

Simplify queries with User-defined functions

Learn how to use built-in Hive and Impala functions or create custom user-defined functions (UDFs) for specific needs.

Register UDFs using Hive commands and incorporate them into queries. Impala supports UDFs, enabling custom logic for processing column values, complex calculations, and data transformations. These UDFs streamline query logic and enhance flexibility in data processing.

Creating an Impala user-defined function

User-defined functions (frequently abbreviated as UDFs) let you code your own application logic for processing column values during an Impala query. For example, a UDF could perform calculations using an external math library, combine several column values into one, do geospatial calculations, or other kinds of tests and transformations that are outside the scope of the built-in SQL operators and functions.

You can use UDFs to simplify query logic when producing reports, or to transform data in flexible ways when copying from one table to another with the INSERT ... SELECT syntax.

You might be familiar with this feature from other database products, under names such as stored functions or stored routines.

Impala support for UDFs is available in Impala 1.2 and higher:

- In Impala 1.1, using UDFs in a query required using the Hive shell. (Because Impala and Hive share the same metastore database, you could switch to Hive to run just those queries requiring UDFs, then switch back to Impala.)
- Starting in Impala 1.2, Impala can run both high-performance native code UDFs written in C++, and Java-based Hive UDFs that you might already have written.

- Impala can run scalar UDFs that return a single value for each row of the result set, and user-defined aggregate functions (UDAFs) that return a value based on a set of rows. Currently, Impala does not support user-defined table functions (UDTFs) or window functions.

UDF concepts

Depending on your use case, you might write all-new functions, reuse Java UDFs that you have already written for Hive, or port Hive Java UDF code to higher-performance native Impala UDFs in C++. You can code either scalar functions for producing results one row at a time, or more complex aggregate functions for doing analysis across. The following sections discuss these different aspects of working with UDFs.

UDFs and UDAFs

Depending on your use case, the user-defined functions (UDFs) you write might accept or produce different numbers of input and output values:

- The most general kind of user-defined function (the one typically referred to by the abbreviation UDF) takes a single input value and produces a single output value. When used in a query, it is called once for each row in the result set. For example:

```
select customer_name, is_frequent_customer(customer_id) from
 customers;
select obfuscate(sensitive_column) from sensitive_data;
```

- A user-defined aggregate function (UDAF) accepts a group of values and returns a single value. You use UDAFs to summarize and condense sets of rows, in the same style as the built-in COUNT, MAX(), SUM(), and AVG() functions. When called in a query that uses the GROU P BY clause, the function is called once for each combination of GROUP BY values. For example:

```
-- Evaluates multiple rows but returns a single value.
select closest_restaurant(latitude, longitude) from places;

-- Evaluates batches of rows and returns a separate value for
 each batch.
select most_profitable_location(store_id, sales, expenses, tax
_rate, depreciation) from franchise_data group by year;
```

- Currently, Impala does not support other categories of user-defined functions, such as user-defined table functions (UDTFs) or window functions.

Native Impala UDFs

Impala supports UDFs written in C++, in addition to supporting existing Hive UDFs written in Java. Cloudera recommends using C++ UDFs because the compiled native code can yield higher performance, with UDF execution time often 10x faster for a C++ UDF than the equivalent Java UDF.

Using Hive UDFs with Impala

Impala can run Java-based user-defined functions (UDFs), originally written for Hive, with no changes, subject to the following conditions:

- The parameters and return value must all use scalar data types supported by Impala. For example, complex or nested types are not supported.
- Hive/Java UDFs must extend org.apache.hadoop.hive.ql.exec.UDF class.
- Currently, Hive UDFs that accept or return the TIMESTAMP type are not supported.
- Prior to Impala 2.5 the return type must be a “Writable” type such as Text or IntWritable, rather than a Java primitive type such as String or int. Otherwise, the UDF returns NULL. In Impala 2.5 and higher, this restriction is lifted, and both UDF arguments and return values can be Java primitive types.

- Hive UDAFs and UDTFs are not supported.
- Typically, a Java UDF will run several times slower in Impala than the equivalent native UDF written in C++.
- In Impala 2.5 and higher, you can transparently call Hive Java UDFs through Impala, or call Impala Java UDFs through Hive. This feature does not apply to built-in Hive functions. Any Impala Java UDFs created with older versions must be re-created using new CREATE FUNCT ION syntax, without any signature for arguments or the return value.

To take full advantage of the Impala architecture and performance features, you can also write Impala-specific UDFs in C++.

For background about Java-based Hive UDFs, see the Hive documentation for UDF. For examples or tutorials for writing such UDFs, search the web for related blog posts.

The ideal way to understand how to reuse Java-based UDFs (originally written for Hive) with Impala is to take some of the Hive built-in functions (implemented as Java UDFs) and take the applicable JAR files through the UDF deployment process for Impala, creating new UDFs with different names:

1. Take a copy of the Hive JAR file containing the Hive built-in functions.
2. Use jar tf *JAR_FILE* to see a list of the classes inside the JAR. You will see names like org/apac he/hadoop/hive/ql/udf/UDFLower.class and org/apache/hadoop/hive/ql/udf/UDFOPNegative. class. Make a note of the names of the functions you want to experiment with. When you specify the entry points for the Impala CREATE FUNCTION statement, change the slash characters to dots and strip off the .class suffix, for example org.apache.hadoop.hive.ql.udf.UDFLower and org.apache.hadoop.hive.ql.udf.UDFOPNegative.
3. Copy that file to an HDFS location that Impala can read. (In the examples here, we renamed the file to hive-builtins.jar in HDFS for simplicity.)
4. For each Java-based UDF that you want to call through Impala, issue a CREATE FUNCT ION statement, with a LOCATION clause containing the full HDFS path of the JAR file, and a SYMBOL clause with the fully qualified name of the class, using dots as separators and without the .class extension. Remember that user-defined functions are associated with a particular database, so issue a USE statement for the appropriate database first, or specify the SQL function name as *DB_NAME.FUNCTION_NAME*. Use completely new names for the SQL functions, because Impala UDFs cannot have the same name as Impala built-in functions.
5. Call the function from your queries, passing arguments of the correct type to match the function signature. These arguments could be references to columns, arithmetic or other kinds of expressions, the results of CAST functions to ensure correct data types, and so on.



Note:

In Impala 2.9 and higher, you can refresh the user-defined functions (UDFs) that Impala recognizes, at the database level, by running the REFRESH FUNCTIONS statement with the database name as an argument. Java-based UDFs can be added to the metastore database through Hive CREATE FUNCTION statements, and made visible to Impala by subsequently running REFRESH FUNCTIONS. For example:

```
CREATE DATABASE shared_udfs;
USE shared_udfs;
...use CREATE FUNCTION statements in Hive to create so
me Java-based UDFs
   that Impala is not initially aware of...
REFRESH FUNCTIONS shared_udfs;
SELECT udf_created_by_hive(c1) FROM ...
```

Java UDF example: Reusing lower() function

For example, the following `impala-shell` session creates an Impala UDF my_lower() that reuses the Java code for the Hive lower(): built-in function. We cannot call it lower() because Impala does not allow UDFs to have the same name as built-in functions. From SQL, we call the

function in a basic way (in a query with no WHERE clause), directly on a column, and on the results of a string expression:

```
[localhost:21000] > create database udfs;
[localhost:21000] > use udfs;
localhost:21000] > create function lower(string) returns string
location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hi
ve.ql.udf.UDFLower';
ERROR: AnalysisException: Function cannot have the same name as a
 builtin: lower
[localhost:21000] > create function my_lower(string) returns s
tring location '/user/hive/udfs/hive.jar' symbol='org.apache.had
oop.hive.ql.udf.UDFLower';
[localhost:21000] > select my_lower('Some String NOT ALREADY LOWE
RCASE');
+----------------------------------------------------+
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
| some string not already lowercase                  |
+----------------------------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create table t2 (s string);
[localhost:21000] > insert into t2 values ('lower'),('UPPER'),('I
nit cap'),('CamelCase');
Inserted 4 rows in 2.28s
[localhost:21000] > select * from t2;
+-----------+
| s         |
+-----------+
| lower     |
| UPPER     |
| Init cap  |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[localhost:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower            |
| upper            |
| init cap         |
| camelcase        |
+------------------+
Returned 4 row(s) in 0.54s
[localhost:21000] > select my_lower(concat('ABC ',s,' XYZ')) f
rom t2;
+----------------------------------------+
| udfs.my_lower(concat('abc ', s, ' xyz')) |
+----------------------------------------+
| abc lower xyz                          |
| abc upper xyz                          |
| abc init cap xyz                       |
| abc camelcase xyz                      |
+----------------------------------------+
Returned 4 row(s) in 0.22s
```

Java UDF example: Reusing negative() function

Here is an example that reuses the Hive Java code for the negative() built-in function. This example demonstrates how the data types of the arguments must match precisely with the function signature. At first, we create an Impala SQL function that can only accept an integer argument. Impala cannot find a matching function when the query passes a floating-point argument, although we can call the

integer version of the function by casting the argument. Then we overload the same function name to also accept a floating-point argument.

```
[localhost:21000] > create table t (x int);
[localhost:21000] > insert into t values (1), (2), (4), (100);
Inserted 4 rows in 1.43s
[localhost:21000] > create function my_neg(bigint) returns bigin
t location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.
hive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4);
+----------------+
| udfs.my_neg(4) |
+----------------+
| -4             |
+----------------+
[localhost:21000] > select my_neg(x) from t;
+----------------+
| udfs.my_neg(x) |
+----------------+
| -2             |
| -4             |
| -100           |
+----------------+
Returned 3 row(s) in 0.60s
[localhost:21000] > select my_neg(4.0);
ERROR: AnalysisException: No matching function with signature:
 udfs.my_neg(FLOAT).
[localhost:21000] > select my_neg(cast(4.0 as int));
+------------------------------+
| udfs.my_neg(cast(4.0 as int)) |
+------------------------------+
| -4                           |
+------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create function my_neg(double) returns double
 location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.h
ive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4.0);
+------------------+
| udfs.my_neg(4.0) |
+------------------+
| -4               |
+------------------+
Returned 1 row(s) in 0.11s
```

You can find the sample files mentioned here in the Impala github repo.

Runtime environment for UDFs

By default, Impala copies UDFs into /tmp, and you can configure this location through the --local_library_dir startup flag for the `impalad` daemon.

Writing UDFs

Before starting UDF development, make sure to install the development package and download the UDF code samples.

When writing UDFs:

- Keep in mind the data type differences as you transfer values from the high-level SQL to your lower-level UDF code. For example, in the UDF code you might be much more aware of how many bytes different kinds of integers require.

- Use best practices for function-oriented programming: choose arguments carefully, avoid side effects, make each function do a single thing, and so on.

Getting started with UDF coding

To understand the layout and member variables and functions of the predefined UDF data types, examine the header file /usr/include/impala_udf/udf.h:

```
// This is the only Impala header required to develop UDFs and U
DAs. This header
// contains the types that need to be used and the FunctionCont
ext object. The context
// object serves as the interface object between the UDF/UDA and
 the impala process.
```

For the basic declarations needed to write a scalar UDF, see the header file udf-sample.h within the sample build environment, which defines a simple function named AddUdf():

```
#ifndef IMPALA_UDF_SAMPLE_UDF_H
#define IMPALA_UDF_SAMPLE_UDF_H
#include <impala_udf/udf.h>

using namespace impala_udf;

IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const
 IntVal& arg2);
#endif
```

For sample C++ code for a simple function named AddUdf(), see the source file udf-sample.cc within the sample build environment:

```
#include "udf-sample.h"
// In this sample we are declaring a UDF that adds two ints and
 returns an int.
IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const
 IntVal& arg2) {
  if (arg1.is_null || arg2.is_null) return IntVal::null();
  return IntVal(arg1.val + arg2.val);
}

// Multiple UDFs can be defined in the same file
```

Data types for function arguments and return values

Each value that a user-defined function can accept as an argument or return as a result value must map to a SQL data type that you could specify for a table column.

Currently, Impala UDFs cannot accept arguments or return values of the Impala complex types (STRUCT, ARRAY, or MAP).

Each data type has a corresponding structure defined in the C++ and Java header files, with two member fields and some predefined comparison operators and constructors:

- is_null indicates whether the value is NULL or not. val holds the actual argument or return value when it is non-NULL.
- Each struct also defines a null() member function that constructs an instance of the struct with the is_null flag set.
- The built-in SQL comparison operators and clauses such as <, >=, BETWEEN, and ORDER BY all work automatically based on the SQL return type of each UDF. For example, Impala knows how to evaluate BETWEEN 1 AND udf_returning_int(col1) or ORDER BY udf_returning_string(col2) without you declaring any comparison operators within the UDF itself.

For convenience within your UDF code, each struct defines == and != operators for comparisons with other structs of the same type. These are for typical C++ comparisons within your own code, not necessarily reproducing SQL semantics. For example, if the is_null flag is set in both structs, they compare as equal. That behavior of null comparisons is different from SQL (where NULL == NULL is NULL rather than true), but more in line with typical C++ behavior.

- Each kind of struct has one or more constructors that define a filled-in instance of the struct, optionally with default values.
- Impala cannot process UDFs that accept the composite or nested types as arguments or return them as result values. This limitation applies both to Impala UDFs written in C++ and Java-based Hive UDFs.
- You can overload functions by creating multiple functions with the same SQL name but different argument types. For overloaded functions, you must use different C++ or Java entry point names in the underlying functions.

The data types defined on the C++ side (in /usr/include/impala_udf/udf.h) are:

- IntVal represents an INT column.
- BigIntVal represents a BIGINT column. Even if you do not need the full range of a BIGINT value, it can be useful to code your function arguments as BigIntVal to make it convenient to call the function with different kinds of integer columns and expressions as arguments. Impala automatically casts smaller integer types to larger ones when appropriate, but does not implicitly cast large integer types to smaller ones.
- SmallIntVal represents a SMALLINT column.
- TinyIntVal represents a TINYINT column.
- StringVal represents a STRING column. It has a len field representing the length of the string, and a ptr field pointing to the string data. It has constructors that create a new StringVal struct based on a null-terminated C-style string, or a pointer plus a length; these new structs still refer to the original string data rather than allocating a new buffer for the data. It also has a constructor that takes a pointer to a FunctionContext struct and a length, that does allocate space for a new copy of the string data, for use in UDFs that return string values.
- BooleanVal represents a BOOLEAN column.
- FloatVal represents a FLOAT column.
- DoubleVal represents a DOUBLE column.
- TimestampVal represents a TIMESTAMP column. It has a date field, a 32-bit integer representing the Gregorian date, that is, the days past the epoch date. It also has a time_of_day field, a 64-bit integer representing the current time of day in nanoseconds.

Variable-length argument lists

UDFs typically take a fixed number of arguments, with each one named explicitly in the signature of your C++ function. Your function can also accept additional optional arguments, all of the same type. For example, you can concatenate two strings, three strings, four strings, and so on. Or you can compare two numbers, three numbers, four numbers, and so on.

To accept a variable-length argument list, code the signature of your function like this:

```
StringVal Concat(FunctionContext* context, const StringVal& sepa
rator,
  int num_var_args, const StringVal* args);
```

In the CREATE FUNCTION statement, after the type of the first optional argument, include ... to indicate it could be followed by more arguments of the same type. For example, the following function accepts a STRING argument, followed by one or more additional STRING arguments:

```
[localhost:21000] > create function my_concat(string, string ...
) returns string location '/user/test_user/udfs/sample.so' symbo
l='Concat';
```

The call from the SQL query must pass at least one argument to the variable-length portion of the argument list.

When Impala calls the function, it fills in the initial set of required arguments, then passes the number of extra arguments and a pointer to the first of those optional arguments.

Handling NULL values

For correctness, performance, and reliability, it is important for each UDF to handle all situations where any NULL values are passed to your function. For example, when passed a NULL, UDFs typically also return NULL. In an aggregate function, which could be passed a combination of real and NULL values, you might make the final value into a NULL (as in CONCAT()), ignore the NULL value (as in AVG()), or treat it the same as a numeric zero or empty string.

Each parameter type, such as IntVal or StringVal, has an is_null Boolean member. Test this flag immediately for each argument to your function, and if it is set, do not refer to the val field of the argument structure. The val field is undefined when the argument is NULL, so your function could go into an infinite loop or produce incorrect results if you skip the special handling for NULL.

If your function returns NULL when passed a NULL value, or in other cases such as when a search string is not found, you can construct a null instance of the return type by using its null() member function.

Memory allocation for UDFs

By default, memory allocated within a UDF is deallocated when the function exits, which could be before the query is finished. The input arguments remain allocated for the lifetime of the function, so you can refer to them in the expressions for your return values. If you use temporary variables to construct all-new string values, use the StringVal() constructor that takes an initial FunctionCont ext* argument followed by a length, and copy the data into the newly allocated memory buffer.

Thread-safe work area for UDFs

One way to improve performance of UDFs is to specify the optional PREPARE_FN and CLOS E_FN clauses on the CREATE FUNCTION statement. The “prepare” function sets up a thread-safe data structure in memory that you can use as a work area. The “close” function deallocates that memory. Each subsequent call to the UDF within the same thread can access that same memory area. There might be several such memory areas allocated on the same host, as UDFs are parallelized using multiple threads.

Within this work area, you can set up predefined lookup tables, or record the results of complex operations on data types such as STRING or TIMESTAMP. Saving the results of previous computations rather than repeating the computation each time is an optimization known as Memoization. For example, if your UDF performs a regular expression match or date manipulation on a column that repeats the same value over and over, you could store the last-computed value or a hash table of already-computed values, and do a fast lookup to find the result for subsequent iterations of the UDF.

Each such function must have the signature:

```
void FUNCTION_NAME(impala_udf::FunctionContext*, impala_udf::F
unctionContext::FunctionScope)
```

Currently, only THREAD_SCOPE is implemented, not FRAGMENT_SCOPE. See udf.h for details about the scope values.

Error handling for UDFs

To handle errors in UDFs, you call functions that are members of the initial FunctionContext* argument passed to your function.

A UDF can record one or more warnings, for conditions that indicate minor, recoverable problems that do not cause the query to stop. The signature for this function is:

```
bool AddWarning(const char* warning_msg);
```

For a serious problem that requires cancelling the query, a UDF can set an error flag that prevents the query from returning any results. The signature for this function is:

```
void SetError(const char* error_msg);
```

Writing user-defined aggregate functions (UDAFs)

User-defined aggregate functions (UDAFs or UDAs) are a powerful and flexible category of user-defined functions. If a query processes N rows, calling a UDAF during the query condenses the result set, anywhere from a single value (such as with the SUM or MAX functions), or some number less than or equal to N (as in queries using the GROUP BY or HAVING clause).

The underlying functions for a UDA

A UDAF must maintain a state value across subsequent calls, so that it can accumulate a result across a set of calls, rather than derive it purely from one set of arguments. For that reason, a UDAF is represented by multiple underlying functions:

- An initialization function that sets any counters to zero, creates empty buffers, and does any other one-time setup for a query.
- An update function that processes the arguments for each row in the query result set and accumulates an intermediate result for each node. For example, this function might increment a counter, append to a string buffer, or set flags.
- A merge function that combines the intermediate results from two different nodes.
- A serialize function that flattens any intermediate values containing pointers, and frees any memory allocated during the init, update, and merge phases.
- A finalize function that either passes through the combined result unchanged, or does one final transformation.

In the SQL syntax, you create a UDAF by using the statement CREATE AGGREGATE FUN CTION. You specify the entry points of the underlying C++ functions using the clauses INIT_FN, UPDATE_FN, MERGE_FN, SERIALIZE_FN, and FINALIZE_FN.

For convenience, you can use a naming convention for the underlying functions and Impala automatically recognizes those entry points. Specify the UPDATE_FN clause, using an entry point name containing the string update or Update. When you omit the other _FN clauses from the SQL statement, Impala looks for entry points with names formed by substituting the update or Update portion of the specified name.

uda-sample.h:

See this file online at: uda-sample.h

uda-sample.cc:

See this file online at: uda-sample.cc

Intermediate results for UDAs

A user-defined aggregate function might produce and combine intermediate results during some phases of processing, using a different data type than the final return value. For example, if you implement a function similar to the built-in AVG() function, it must keep track of two values, the number of values counted and the sum of those values. Or, you might accumulate a string value over the course of a UDA, then in the end return a numeric or Boolean result.

In such a case, specify the data type of the intermediate results using the optional INTERMEDIATE *TYPE_NAME* clause of the CREATE AGGREGATE FUNCTION statement. If the intermediate data is a typeless byte array (for example, to represent a C++ struct or array), specify the type name as CHAR(*N*), with *N* representing the number of bytes in the intermediate result buffer.

For an example of this technique, see the trunc_sum() aggregate function, which accumulates intermediate results of type DOUBLE and returns BIGINT at the end. View the appropriate CREATE FUNCTION statement and the implementation of the underlying TruncSum*() functions on Github.

- test_udfs.py
- test-udas.cc

Building and deploying UDFs

This section explains the steps to compile Impala UDFs from C++ source code, and deploy the resulting libraries for use in Impala queries.

Impala UDF development package ships with a sample build environment for UDFs, that you can study, experiment with, and adapt for your own use.

The `cmake` configuration command reads the file CMakeLists.txt and generates a Makefile customized for your particular directory paths. Then the `make` command runs the actual build steps based on the rules in the Makefile.

Impala loads the shared library from an HDFS location. After building a shared library containing one or more UDFs, use hdfs dfs or hadoop fs commands to copy the binary file to an HDFS location readable by Impala.

The final step in deployment is to issue a CREATE FUNCTION statement in the `impala-shell` interpreter to make Impala aware of the new function. Because each function is associated with a particular database, always issue a USE statement to the appropriate database before creating a function, or specify a fully qualified name, that is, CREATE FUNCTION *DB_NAME.FUNCTION_NAME*.

As you update the UDF code and redeploy updated versions of a shared library, use DROP FUNCTION and CREATE FUNCTION to let Impala pick up the latest version of the code.



Note:

In Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the `catalogd` daemon. Prior to Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

See CREATE FUNCTION statement and DROP FUCNTION statement for the new syntax for the persistent Java UDFs.

Prerequisites for the build environment are:

1. Install the packages using the appropriate package installation command for your Linux distribution.

```
sudo yum install gcc-c++ cmake boost-devel
sudo yum install impala-udf-devel
# The package name on Ubuntu and Debian is impala-udf-dev.
```

2. Download the UDF sample code:

```
git clone https://github.com/cloudera/impala-udf-samples
```

```
cd impala-udf-samples &amp;amp;&amp;amp; cmake . &amp;amp;&amp;amp; make
```

3. Unpack the sample code in udf_samples.tar.gz and use that as a template to set up your build environment.

To build the original samples:

```
# Process CMakeLists.txt and set up appropriate Makefiles.
cmake .
# Generate shared libraries from UDF and UDAF sample code,
# udf_samples/libudfsample.so and udf_samples/libudasample.so
make
```

The sample code to examine, experiment with, and adapt is in these files:

- udf-sample.h: Header file that declares the signature for a scalar UDF (AddUDF).
- udf-sample.cc: Sample source for a simple UDF that adds two integers. Because Impala can reference multiple function entry points from the same shared library, you could add other UDF functions in this file and add their signatures to the corresponding header file.
- udf-sample-test.cc: Basic unit tests for the sample UDF.
- uda-sample.h: Header file that declares the signature for sample aggregate functions. The SQL functions will be called COUNT, AVG, and STRINGCONCAT. Because aggregate functions require more elaborate coding to handle the processing for multiple phases, there are several underlying C++ functions such as CountInit, AvgU pdate, and StringConcatFinalize.
- uda-sample.cc: Sample source for simple UDAFs that demonstrate how to manage the state transitions as the underlying functions are called during the different phases of query processing.
  - The UDAF that imitates the COUNT function keeps track of a single incrementing number; the merge functions combine the intermediate count values from each Impala node, and the combined number is returned verbatim by the finalize function.
  - The UDAF that imitates the AVG function keeps track of two numbers, a count of rows processed and the sum of values for a column. These numbers are updated and merged as with COUNT, then the finalize function divides them to produce and return the final average value.
  - The UDAF that concatenates string values into a comma-separated list demonstrates how to manage storage for a string that increases in length as the function is called for multiple rows.
- uda-sample-test.cc: basic unit tests for the sample UDAFs.

Performance considerations for UDFs

Because a UDF typically processes each row of a table, potentially being called billions of times, the performance of each UDF is a critical factor in the speed of the overall ETL or ELT pipeline. Tiny optimizations you can make within the function body can pay off in a big way when the function is called over and over when processing a huge result set.

Examples of creating and using UDFs

This section demonstrates how to create and use all kinds of user-defined functions (UDFs).

For downloadable examples that you can experiment with, adapt, and use as templates for your own functions, see the Cloudera sample UDF github. You must have already installed the appropriate header files, as explained in *Building and deploying UDFs*.

Sample C++ UDFs: HasVowels, CountVowels, StripVowels

This example shows 3 separate UDFs that operate on strings and return different data types. In the C++ code, the functions are HasVowels() (checks if a string contains any vowels), CountVowels() (returns the number of vowels in a string), and StripVowels() (returns a new string with vowels removed).

First, we add the signatures for these functions to udf-sample.h in the demo build environment:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal&
input);
IntVal CountVowels(FunctionContext* context, const StringVal& ar
g1);
StringVal StripVowels(FunctionContext* context, const StringVal&
 arg1);
```

Then, we add the bodies of these functions to udf-sample.cc:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal&
input)
{
        if (input.is_null) return BooleanVal::null();

        int index;
        uint8_t *ptr;

        for (ptr = input.ptr, index = 0; index <= input.len; i
ndex++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o'
 || c == 'u')
                {
                        return BooleanVal(true);
                }
        }
        return BooleanVal(false);
}

IntVal CountVowels(FunctionContext* context, const StringVal&
arg1)
{
        if (arg1.is_null) return IntVal::null();

        int count;
        int index;
        uint8_t *ptr;

        for (ptr = arg1.ptr, count = 0, index = 0; index <= arg1.
len; index++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o'
 || c == 'u')
                {
                        count++;
                }
        }
        return IntVal(count);
}

StringVal StripVowels(FunctionContext* context, const StringVal&
 arg1)
{
        if (arg1.is_null) return StringVal::null();

        int index;
        std::string original((const char *)arg1.ptr,arg1.len);
        std::string shorter("");
```

```
        for (index = 0; index < original.length(); index++)
        {
                uint8_t c = original[index];
                uint8_t l = tolower(c);

                if (l == 'a' || l == 'e' || l == 'i' || l == 'o'
 || l == 'u')
                {
                        ;
                }
                else
                {
                    shorter.append(1, (char)c);
                }
        }
// The modified string is stored in 'shorter', which is destroyed
 when this function ends. We need to make a string val
// and copy the contents.
        StringVal result(context, shorter.size()); // Only the ve
rsion of the ctor that takes a context object allocates new memo
ry
        memcpy(result.ptr, shorter.c_str(), shorter.size());
        return result;
}
```

We build a shared library, libudfsample.so, and put the library file into HDFS where Impala can read it:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
[ 33%] Built target udasample
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
Scanning dependencies of target udfsample
[ 83%] Building CXX object CMakeFiles/udfsample.dir/udf-sample.o
Linking CXX shared library udf_samples/libudfsample.so
[ 83%] Built target udfsample
Linking CXX executable udf_samples/udf-sample-test
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudfsample.so /user/hive/udfs/li
budfsample.so
```

Finally, we go into the `impala-shell` interpreter where we set up some sample data, issue CREATE FUNCTION statements to set up the SQL function names, and call the functions in some queries:

```
[localhost:21000] > create database udf_testing;
[localhost:21000] > use udf_testing;

[localhost:21000] > create function has_vowels (string) returns b
oolean location '/user/hive/udfs/libudfsample.so' symbol='HasVow
els';
[localhost:21000] > select has_vowels('abc');
+------------------------+
| udfs.has_vowels('abc') |
+------------------------+
| true                   |
+------------------------+
Returned 1 row(s) in 0.13s
[localhost:21000] > select has_vowels('zxcvbnm');
```

```
+---------------------------+
| udfs.has_vowels('zxcvbnm') |
+---------------------------+
| false                     |
+---------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select has_vowels(null);
+-----------------------+
| udfs.has_vowels(null) |
+-----------------------+
| NULL                  |
+-----------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select s, has_vowels(s) from t2;
+-----------+--------------------+
| s         | udfs.has_vowels(s) |
+-----------+--------------------+
| lower     | true               |
| UPPER     | true               |
| Init cap  | true               |
| CamelCase | true               |
+-----------+--------------------+
Returned 4 row(s) in 0.24s
[localhost:21000] > create function count_vowels (string) retur
ns int location '/user/hive/udfs/libudfsample.so' symbol='CountV
owels';
[localhost:21000] > select count_vowels('cat in the hat');
+-------------------------------------+
| udfs.count_vowels('cat in the hat') |
+-------------------------------------+
| 4                                   |
+-------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select s, count_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.count_vowels(s) |
+-----------+----------------------+
| lower     | 2                    |
| UPPER     | 2                    |
| Init cap  | 3                    |
| CamelCase | 4                    |
+-----------+----------------------+
Returned 4 row(s) in 0.23s
[localhost:21000] > select count_vowels(null);
+-------------------------+
| udfs.count_vowels(null) |
+-------------------------+
| NULL                    |
+-------------------------+
Returned 1 row(s) in 0.12s

[localhost:21000] > create function strip_vowels (string) returns
 string location '/user/hive/udfs/libudfsample.so' symbol='Strip
Vowels';
[localhost:21000] > select strip_vowels('abcdefg');
+-----------------------------+
| udfs.strip_vowels('abcdefg') |
+-----------------------------+
| bcdfg                       |
+-----------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select strip_vowels('ABCDEFG');
+-----------------------------+
| udfs.strip_vowels('abcdefg') |
```

```
+-----------------------------+
| BCDFG                       |
+-----------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select strip_vowels(null);
+-------------------------+
| udfs.strip_vowels(null) |
+-------------------------+
| NULL                    |
+-------------------------+
Returned 1 row(s) in 0.16s
[localhost:21000] > select s, strip_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.strip_vowels(s) |
+-----------+----------------------+
| lower     | lwr                  |
| UPPER     | PPR                  |
| Init cap  | nt cp                |
| CamelCase | CmlCs                |
+-----------+----------------------+
Returned 4 row(s) in 0.24s
```

Sample C++ UDA: SumOfSquares

```
[localhost:21000] > insert overwrite sos values (1, 1), (2, 0),
(3, 1), (4, 0);
Inserted 4 rows in 1.24s

[localhost:21000] > -- Compute 1 squared + 3 squared, and 2 sq
uared + 4 squared;
[localhost:21000] > select y, sum_of_squares(x) from sos group by
 y;
+---+------------------------+
| y | udfs.sum_of_squares(x) |
+---+------------------------+
| 1 | 10                     |
| 0 | 20                     |
+---+------------------------+
Returned 2 row(s) in 0.43s
```

This example demonstrates a user-defined aggregate function (UDA) that produces the sum of the squares of its input values.

The coding for a UDA is a little more involved than a scalar UDF, because the processing is split into several phases, each implemented by a different function. Each phase is relatively straightforward: the “update” and “merge” phases, where most of the work is done, read an input value and combine it with some accumulated intermediate value.

As in our sample UDF from the previous example, we add function signatures to a header file (in this case, uda-sample.h). Because this is a math-oriented UDA, we make two versions of each function, one accepting an integer value and the other accepting a floating-point value.

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val);
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val);

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal
& input, BigIntVal* val);
void SumOfSquaresUpdate(FunctionContext* context, const Double
Val& input, DoubleVal* val);

void SumOfSquaresMerge(FunctionContext* context, const BigIntV
al& src, BigIntVal* dst);
```

```
void SumOfSquaresMerge(FunctionContext* context, const DoubleV
al& src, DoubleVal* dst);

BigIntVal SumOfSquaresFinalize(FunctionContext* context, const Bi
gIntVal& val);
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const Do
ubleVal& val);
```

We add the function bodies to a C++ source file (in this case, uda-sample.cc):

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val) {
  val->is_null = false;
  val->val = 0;
}
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val) {
  val->is_null = false;
  val->val = 0.0;
}

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal
& input, BigIntVal* val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal
& input, DoubleVal* val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}

void SumOfSquaresMerge(FunctionContext* context, const BigIntVal&
 src, BigIntVal* dst) {
  dst->val += src.val;
}
void SumOfSquaresMerge(FunctionContext* context, const DoubleV
al& src, DoubleVal* dst) {
  dst->val += src.val;
}
BigIntVal SumOfSquaresFinalize(FunctionContext* context, const
BigIntVal& val) {
  return val;
}
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const
 DoubleVal& val) {
  return val;
}
```

As with the sample UDF, we build a shared library and put it into HDFS:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
Scanning dependencies of target udasample
[ 33%] Building CXX object CMakeFiles/udasample.dir/uda-sample.o
Linking CXX shared library udf_samples/libudasample.so
[ 33%] Built target udasample
Scanning dependencies of target uda-sample-test
[ 50%] Building CXX object CMakeFiles/uda-sample-test.dir/uda-s
ample-test.o
Linking CXX executable udf_samples/uda-sample-test
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
```

```
[ 83%] Built target udfsample
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudasample.so /user/hive/udfs/li
budasample.so
```

To create the SQL function, we issue a CREATE AGGREGATE FUNCTION statement and specify the underlying C++ function names for the different phases:

```
[localhost:21000] > use udf_testing;

[localhost:21000] > create table sos (x bigint, y double);
[localhost:21000] > insert into sos values (1, 1.1), (2, 2.2),
(3, 3.3), (4, 4.4);
Inserted 4 rows in 1.10s

[localhost:21000] > create aggregate function sum_of_squares(b
igint) returns bigint
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';
[localhost:21000] > -- Compute the same value using literals or
 the UDA;
[localhost:21000] > select 1*1 + 2*2 + 3*3 + 4*4;
+-------------------------------+
| 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 |
+-------------------------------+
| 30                            |
+-------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(x) from sos;
+------------------------+
| udfs.sum_of_squares(x) |
+------------------------+
| 30                     |
+------------------------+
Returned 1 row(s) in 0.35s
```

Until we create the overloaded version of the UDA, it can only handle a single data type. To allow it to handle DOUBLE as well as BIGINT, we issue another CREATE AGGREGATE FUNCTION statement:

```
[localhost:21000] > select sum_of_squares(y) from sos;
ERROR: AnalysisException: No matching function with signature: ud
fs.sum_of_squares(DOUBLE).

[localhost:21000] > create aggregate function sum_of_squares(dou
ble) returns double
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';

[localhost:21000] > -- Compute the same value using literals or t
he UDA;
[localhost:21000] > select 1.1*1.1 + 2.2*2.2 + 3.3*3.3 + 4.4*4.4;
+-----------------------------------------------+
| 1.1 * 1.1 + 2.2 * 2.2 + 3.3 * 3.3 + 4.4 * 4.4 |
+-----------------------------------------------+
| 36.3                                          |
```

```
+-------------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(y) from sos;
+------------------------+
| udfs.sum_of_squares(y) |
+------------------------+
| 36.3                   |
+------------------------+
Returned 1 row(s) in 0.35s
```

Typically, you use a UDA in queries with GROUP BY clauses, to produce a result set with a separate aggregate value for each combination of values from the GROUP BY clause. Let's change our sample table to use 0 to indicate rows containing even values, and 1 to flag rows containing odd values. Then the GROUP BY query can return two values, the sum of the squares for the even values, and the sum of the squares for the odd values:

Security considerations for UDFs

When the Impala authorization feature is enabled:

- To call a UDF in a query, you must have the required read privilege for any databases and tables used in the query.
- The CREATE FUNCTION statement requires:
  - The CREATE privilege on the database.
  - The ALL privilege on two URIs where the URIs are:
    - The JAR file on the file system. For example:

      ```
      GRANT ALL ON URI 'file:///PATH_TO_MY.JAR' TO ROLE MY_ROLE;
      ```

    - The JAR on HDFS. For example:

      ```
      GRANT ALL ON URI 'hdfs:///PATH/TO/JAR' TO ROLE MY_ROLE
      ```

Limitations and restrictions for Impala UDFs

The following limitations and restrictions apply to Impala UDFs in the current release.

Limited support for Hive Generic UDFs

Hive has 2 types of UDFs. This release contains limited support for the second generation UDFs called GenericUDFs. The main limitations are as follows:

- Decimal types are not supported
- Complex types are not supported
- Functions are not extracted from the jar file

GenericUDFs cannot be made permanent. They will need to be recreated every time the server is restarted.

Other limitations

- Impala does not support Hive UDFs that accept or return composite or nested types, or other types not available in Impala tables.
- The Hive current_user() function cannot be called from a Java UDF through Impala.
- All Impala UDFs must be deterministic, that is, produce the same output each time when passed the same argument values. For example, an Impala UDF must not call functions such as rand() to produce different values for each invocation. It must not retrieve data from external sources, such as from disk or over the network.

- An Impala UDF must not spawn other threads or processes.
- Prior to Impala 2.5 when the `catalogd` process is restarted, all UDFs become undefined and must be reloaded. In Impala 2.5 and higher, this limitation only applies to older Java UDFs. Re-create those UDFs using the new CREATE FUNCTION syntax for Java UDFs, which excludes the function signature, to remove the limitation entirely.
- Impala currently does not support user-defined table functions (UDTFs).
- The CHAR and VARCHAR types cannot be used as input arguments or return values for UDFs.

Impala workload management

Learn how to enable Impala query logging in Cloudera Data Warehouse to track queries, analyze performance, and retain execution data for better insights.

Cloudera Data Warehouse provides you the option to enable logging Impala queries on an existing Virtual Warehouse or while creating a new Impala Virtual Warehouse. By logging the Impala queries in Cloudera Data Warehouse, you gain increased observability of the workloads running on Impala, which you can use to improve the performance of your Impala Virtual Warehouses.

This feature represents a significant enhancement to query profiling capabilities. You can have Impala archive crucial data from each query's profile into dedicated database tables known as the query history table and live query table. These tables are part of the sys database and are designed to store valuable information that can later be queried using any Impala client, providing a consolidated view of both actively running and previously executed queries.

The query history table, sys.impala_query_log proves particularly useful when dissecting workloads for in-depth analysis of query performance. Unlike the limitations associated with query profiles, which are only available to the client that initiated the query, the query history table offers a comprehensive solution for querying completed queries without the need to parse the text of each query profile. Additionally, the query history table provides a comprehensive view across all Impala coordinators.

The Impala query information is stored indefinitely in the sys.impala_query_log table whereas the sys.impala_query_live table reflects the in-memory state of all Impala coordinators. Actively running and recently completed queries are stored in this table. Data is removed from this table once the query finishes and is persisted in the sys.impala_query_log table or if the coordinator is restarted. Therefore, there is a possibility that some of the records could momentarily be duplicated in both these tables.

Since the sys.impala_query_live table is stored only in-memory, recently completed queries that are not yet persisted to the sys.impala_query_log table are lost if the coordinator crashes. However, if the coordinator is shut down gracefully, then the recently completed queries are stored in the sys.impala_query_log table and are not lost.

The <onlyCoordinators> element in Impala’s Admission Control restricts a request pool to coordinators only, excluding executors. This is mainly used for querying the sys.impala_query_live table. However, these pools can still run any query, potentially exhausting coordinator resources. Proper naming is important to avoid unintended query routing. For more information, see *Apache Impala: onlyCoordinators*.

Related Information

Hive query history service

Impala workload management table format

Learn about the available columns in the query history and live query system tables.

Table Format

The following columns are available as part of the query history and live query system tables:

Column Name	Description	Data Type	Sample Value
cluster_id	String specified through the Impala startup flag to uniquely identify an instance.	string	cluster-123
query_id	Impala assigned query identifier.	string	214d08bef0831e7a:3c65392400000000
session_id	Impala assigned session identifier.	string	ea4f661af43993d8:587839553a41adb8
session_type	Client session type.	string	HIVESERVER2
hiveserver2_protocol_version	Version of the HiveServer (HS2) protocol that was used by the client when connecting.	string	HIVE_CLI_SERVICE_PROTOCOL_V6
db_user	Effective user on the cluster.	string	csso_name
db_user_connection	Username from an authenticated client.	string	csso_name
db_name	Name of the database being queried.	string	default
impala_coordinator	Name of the coordinator for the query.	string	coord-22899:27000
query_status	Status of the query when it completes.	string	OK
query_state	Final state of the query.	string	FINISHED
impala_query_end_state	Final Impala state of the query.	string	FINISHED
query_type	Type of the query.	string	QUERY
network_address	Client IP address and port.	string	127.0.0.1:40120
start_time_utc	Time when the query started. Time zone is in UTC.	timestamp	2024-07-17 17:13:46.414316000
total_time_ms	Difference between the query end time and start time, in milliseconds (digits after the decimal point represent milliseconds).	decimal(18,3)	136.121
query_opts_config	List of query options stored as a single string containing comma-separated values of key-value pairs.	string	TIMEZONE=America/Los_Angeles,CLIENT_IDENTIFIER=Impala Shell v4.4.0a1 (04bdb4d) built on Mon Nov 20 10:49:35 PST 2023
resource_pool	Name of the resource pool for the query.	string	default-pool
per_host_mem_estimate	Size, in bytes of the per-host memory estimate.	bigint	5
dedicated_coord_mem_estimate	Size, in bytes of the dedicated coordinator memory estimate.	bigint	4
per_host_fragment_instances	Comma-separated string listing each host and its fragment instances.	string	myhost-1:27000=1,myhost-2:27001=2
backends_count	Count of the number of backends used by this query.	integer	2
admission_result	Result of the admission (not applicable to DDLs).	string	Admitted immediately
cluster_memory_admitted	Cluster memory, in bytes that was admitted.	integer	4
executor_group	Name of the executor group.	string	executor_group

Column Name	Description	Data Type	Sample Value
executor_groups	List of all executor groups including the groups that were considered and rejected as part of Workload Aware Auto Scaling.	string	executor_group1, executor_group2…
exec_summary	Full text of the executor summary.	string	
num_rows_fetched	Number of rows fetched by the query.	bigint	6001215
row_materialization_rows_per_sec	Count of the number of rows materialized per second.	bigint	3780
row_materialization_time_ms	Time spent materializing rows converted to milliseconds.	decimal(18,3)	1.58
compressed_bytes_spilled	Count of bytes that were written (or spilled) to scratch disk space.	bigint	241515
event_planning_finished	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	27.253
event_submit_for_admission	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	30.204
event_completed_admission	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	30.986
event_all_backends_started	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	31.969
event_rows_available	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	31.969
event_first_row_fetched	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	135.175
event_last_row_fetched	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	135.181
event_unregister_query	Event from the timeline.The value represents the number of milliseconds since the query was received.	decimal(18,3)	141.435
read_io_wait_total_ms	Total read I/O wait time converted to milliseconds.	bigint	15.091
read_io_wait_mean_ms	Average read I/O wait time across executors converted to milliseconds	bigint	35.515
bytes_read_cache_total	Total bytes read from the data cache	bigint	45823
bytes_read_total	Total bytes read	bigint	745227
pernode_peak_mem_min	Minimum value of all the per-node peak memory usages	bigint	5552846

Column Name	Description	Data Type	Sample Value
pernode_peak_mem_max	Maximum value of all the per-node peak memory usages	bigint	5552846
pernode_peak_mem_mean	Mean value of all the per-node peak memory usages	bigint	5552846
sql	SQL statement as provided by the user	string	SELECT db_user, total_time_ms from impala_query_log where query_state = 'EXCEPTION';
plan	Full text of the query plan	string	
tables_queried	Comma-separated string containing all the tables queried in the SQL statement. Aliased tables are resolved to their actual table names.	string	db.tbl,db.tbl
select_columns	Comma-separated string containing all columns from the select list of the sql. Aliased columns are resolved to their actual column names. Each column is in the format database.table.column_name.	string	db.tbl.col1,db.tbl.col2
where_columns	Comma-separated string containing all columns from the where list of the sql. Aliased columns are resolved to their actual column names. Each column is in the format database.table.column_name.	string	db.tbl.col1,db.tbl.col2
join_columns	Comma-separated string containing all columns from the sql used in a join. Aliased columns are resolved to their actual column names. Each column is in the format database.table.column_name.	string	db.tbl.col1,db.tbl.col2
aggregate_columns	Comma-separated string containing all columns from the group by and having lists of the sql. Aliased columns are resolved to their actual column names. Each column is in the format database.table.column_name.	string	db.tbl.col1,db.tbl.col2
orderby_columns	Comma-separated string containing all columns from the order by list of the sql. Aliased columns are resolved to their actual column names. Each column is in the format database.table.column_name.	string	db.tbl.col1,db.tbl.col2
coordinator_slots	Number of query slots used by the query on the coordinator.	bigint	1
executor_slots	Numberof query slots used by the query on the executors. The value in this column represents the slots used by a single executor, not the total number of slots across all executors.	bigint	1

Impala workload management table maintenance

Understand the maintenance requirements for the sys.impala_query_live and sys.impala_query_log tables.

For efficient query performance, different maintenance needs apply to the sys.impala_query_live and sys.impala_query_log tables.

- Sys.impala_query_live:
  - No maintenance is required because it resides entirely in memory.
- Sys.impala_query_log:
  - As an Iceberg table, it requires periodic maintenance, such as:
    - Computing statistics.
    - Optimizing the table structure.
    - Performing snapshot expiration or cleanup.

Since Impala workloads are unique, no automatic maintenance is performed on the sys.impala_query_log table. You should schedule maintenance tasks according to your workload needs.

To optimize the Impala query log, run the query OPTIMIZE TABLE sys.impala_query_log (FILE_SIZE_THRESHOLD_MB=128). Cloudera recommends testing this query in the development or test environments to evaluate its impact on your workload. For best results, run the query during low cluster activity times.

Impala workload management use cases

Learn how to use query history to track executed queries by user, identify frequently queried statements, and report long-running queries for analysis.

A consolidated view of reports from previously executed queries can be useful in the following use cases:

- Collecting history of all queries run and reporting by user, start/end time, and execution duration. For instance:

```
SELECT db_user, start_time, end_time, total_time_ms, sql FROM impala_query_log ORDER BY db_user;
```

- Collecting the top five most frequently executed queries. For instance:

```
SELECT lower(sql) sql, count(sql) count FROM sys.impala_query_log GROUP BY lower(sql) ORDER BY count desc LIMIT 5;
```

- Reporting on queries that ran over 10 minutes. For instance:

```
SELECT db_user, total_time_ms, sql FROM sys.impala_query_log WHERE total_time_ms > 600000;
```

- Reporting on queries that are actively running and have been running for over 10 minutes. For instance:

```
SELECT db_user, total_time_ms, sql FROM sys.impala_query_live WHERE total_time_ms > 600000;
```

Enable Impala workload management

Learn how to enable and configure Impala query logging to store and analyze query history.

To use this feature, enable Impala query logging while creating a new Virtual Warehouse or by editing an existing one by selecting the Log Impala Queries option. By default, the Log Impala Queries option is off.

You can then configure the Impala coordinator using specific startup flags to store query history. Impala manages the table serving as a centralized repository for all query histories across databases. Completed queries are periodically inserted into this table based on a preconfigured interval.

This feature streamlines the process of query history management, providing a more accessible and comprehensive way to analyze and retrieve information about completed queries.

Note: This feature is available from Cloudera Data Warehouse versions 2024.0.18.0 (tech preview), 2025.0.19.0 and higher (GA).

Important: The following query types are not written into the query logging tables:

- SET
- SHOW
- USE
- DESCRIBE

Impala coordinator Startup flags

Learn about Impala coordinator startup flags used to create and configure query logging tables during startup.

On startup, each Impala coordinator runs an SQL statement to create the query logging tables. The following table lists the Impala startup coordinator flags that you can configure:

Name	Data Type	Default	Description
cluster_id	string		Specifies an identifier string that uniquely represents this cluster. This identifier is included in both the tables and is used as a table partition for the sys.impala_query _log table.
query_log_shutdown_timeout_s	number (seconds)	30	Hidden flag. Number of seconds to wait for the queue of completed queries to be carried into the query history table before timing out and continuing the shutdown process.The query history table drain process runs after the shutdown process completes, therefore the max shutdown time is extended by the value specified in this flag.
workload_mgmt_user	string	impala	Specifies the user that will be used to insert records into the query history table.
query_log_write_interval_s	number (seconds)	300	Number of seconds to wait before inserting completed queries into the query history table. Allows for batching inserts to help avoid small files.

Name	Data Type	Default	Description
query_log_max_queued	number	3000	Maximum number of completed queries that can be queued before they are written to the query history table.This flag operates independently of the query_log_write_interval_m flag. If the number of queued records reaches this value, the records will be written to the query log table no matter how much time has passed since the last write.A value of 0 indicates no maximum number of queued records.
query_log_max_sql_length	number	16777216	Maximum length of a SQL statement that will be recorded in the completed queries table. If a SQL statement with a length longer than this specified value is executed, the SQL inserted into the completed queries table will be trimmed to this length. Any characters that require escaping will have their backslash character counted towards this limit.
query_log_max_plan_length	number	16777216	Maximum length of the SQL query plan that will be recorded in the completed queries table. If a query plan has a length longer than this value, the plan inserted into the completed queries table will be trimmed to this length. Any characters that require escaping will have their backslash character counted towards this limit.
query_log_request_pool	string		Specifies a pool or queue used by the queries that insert into the query log table. Empty value causes no pool to be set.
query_log_dml_exec_timeout_s	number	120	Specifies the value of the EXEC_TIME_LIMIT_S query option on the DML that inserts records into the sys.impala_query_log table.

Running queries on system tables

Learn how to configure system table queries in Impala to use only the necessary coordinator resources.

Queries against Impala system tables, such as sys.impala_query_live, experienced delays due to admission control constraints. These queries, which only require only coordinator resources, were blocked by other queries competing for executor resources.

To address this, Impala introduces an "only coordinators" request pool. This allows system table queries to run without waiting for executor resources, even when those resources are fully used. In Cloudera Data Warehouse, queries submitted to an only coordinators request pool continue to run even if no executors are running.

Note:

Running queries on system tables feature for Impala is in technical preview and not recommended for production deployments. Cloudera recommends that you try this feature in test or development environments.

Coordinator-only request pools

You can configure coordinator-only request pools in Impala by setting the <onlyCoordinators> option to true in the fair-scheduler.xml file. When enabled, the request pool runs queries only on coordinators. No executors are required, and all fragment instances are executed exclusively on the coordinators.

Warning: Be cautious when submitting queries to a coordinator-only request pool. If a large or non-system table query is submitted, it will run entirely on the coordinators. This can lead to memory or CPU exhaustion.

To prevent resource issues, Cloudera recommends limiting access to coordinator-only request pools to a small group of users who understand the risks of running complex queries in this configuration.

Configuring the only coordinators request pool

Follow these steps to configure a coordinator-only request pool in Impala to optimize system table queries.

About this task

By configuring a coordinator-only request pool, you can run queries that do not require executors. This reduces resource usage and enhances system performance.

Before you begin

Ensure you have administrator access to modify Impala configurations.

Procedure

1. Log in to the Cloudera web interface and navigate to the Cloudera Data Warehouse service.
2. In the Cloudera Data Warehouse service, click Virtual Warehouses in the left navigation panel.
3. Select the Impala Virtual Warehouse, click options for the warehouse you want to include the onlyCoordinators setting for a request pool.
4. Click Details and navigate to Impala Coordinator under the Configurations tab.
5. Select the fair-scheduler.xm under Configuration files.
6. Add <onlyCoordinators>true</onlyCoordinators>.

```
<queue name="coords">
  <maxResources>10000 mb,0 vcores</maxResources>
  <onlyCoordinators>true</onlyCoordinators>
</queue>
```

7. Click Apply Changes and restart Impala.

Cloudera Data Explorer (Hue) Storage Browser (Technical Preview)

The Data Explorer Storage Browser is a web-based interface designed to provide seamless interaction with multiple file systems, including HDFS, Amazon S3, Azure Blob Storage (ABFS), and Google Cloud Storage (GS). With

enhanced usability and functionality, the Storage Browser improves data management within the Hadoop ecosystem, offering a streamlined experience.

Technical Preview: This is a technical preview feature and considered under development. Do not use this in your production systems. To share your feedback, contact Cloudera Support by logging a case on the Cloudera Support Portal. Technical preview features are not guaranteed troubleshooting guidance and fixes.

Key Advantages

Data Explorer Storage Browser has the following key advantages:

- Seamless Data Management – Provides unified access to multiple file systems from a single interface.
- Enhanced Usability – Provides intuitive navigation, search, and organization features.
- Optimized for Productivity – Provides bulk actions, permission management, and advanced file operations.
- Improved Security & Compliance – Provides role-based access control and granular permission settings.

Supported File Systems

The Data Explorer Storage Browser supports the following file storage platforms:

- HDFS
- GS (Google Cloud Storage)
- S3A (Amazon S3)
- ABFS (Azure Blob Storage)

Enabling the new Cloudera Data Explorer (Hue) Storage Browser (Technical Preview)

The legacy version of the Storage Browser is enabled by default. To utilize the enhanced features and improved interface of the new Data Explorer Storage Browser, you must enable the Data Explorer Storage Browser manually through configuration settings.

Procedure

1. Go to Cloudera Manager Clusters Data Explorer (Hue) service Configuration .
2. In the search, enter hue_safety_valve.ini to locate the Hue Service Advanced Configuration Snippet (Safety Valve) for hue_safety_valve.ini advanced configuration snippet.
3. Set the configuration in the Hue Service Advanced Configuration Snippet (Safety Valve) for the hue_safety_valve.ini advanced configuration snippet as follows:

   ```
   [desktop]
   enable_new_storage_browser=true
   ```

4. Click the Apply Changes button.

   The new Data Explorer Storage Browser is enabled.

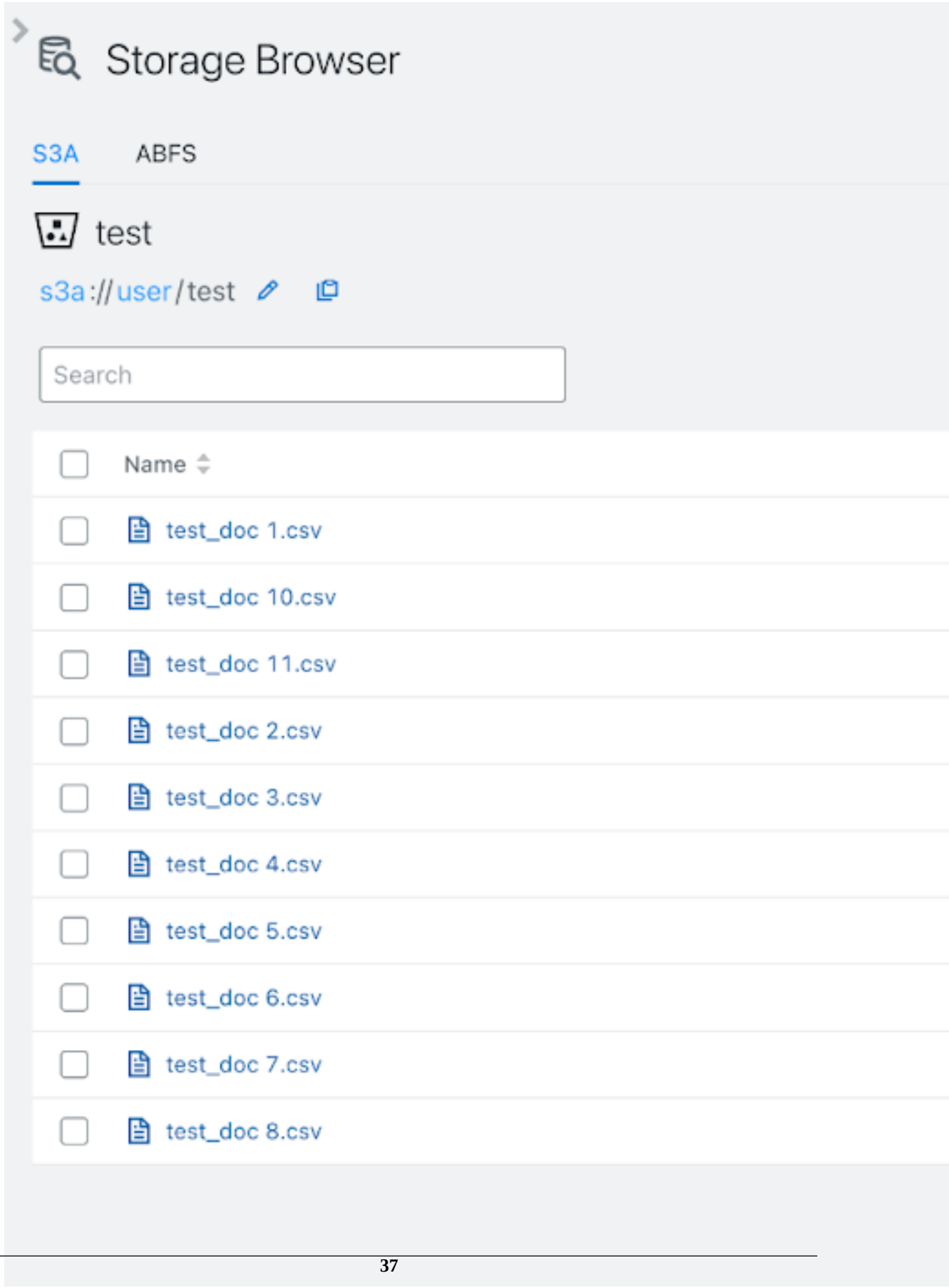
Managing files and operations in Cloudera Data Explorer (Hue) Storage Browser (Technical Preview)

The Data Explorer Storage Browser provides an intuitive interface for managing files across various storage platforms. You can create, upload, move, copy, delete, and preview files efficiently. With enhanced navigation, streamlined file actions, and an improved experience, Data Explorer simplifies data handling while maintaining security and accessibility.

Enhanced navigation and controls

The Data Explorer Storage Browser introduces the following significant improvements in navigation and controls:

- Unified file access – You can view and manage all enabled file types on a single page, categorized into intuitive tabs.
- Breadcrumb navigation – Improved visibility and usability with larger text, editable paths, and a Copy Path button for quick reference.
- Actionable UI elements
  - The Search field – Enables quick keyword-based file searches.
  - The Home icon – Instantly navigates you to your designated home directory.
  - The Refresh icon – Reloads the current view to reflect the latest changes.
- Pagination and Display options – Clean interface with the ability to display up to 1000 items per page for improved readability.

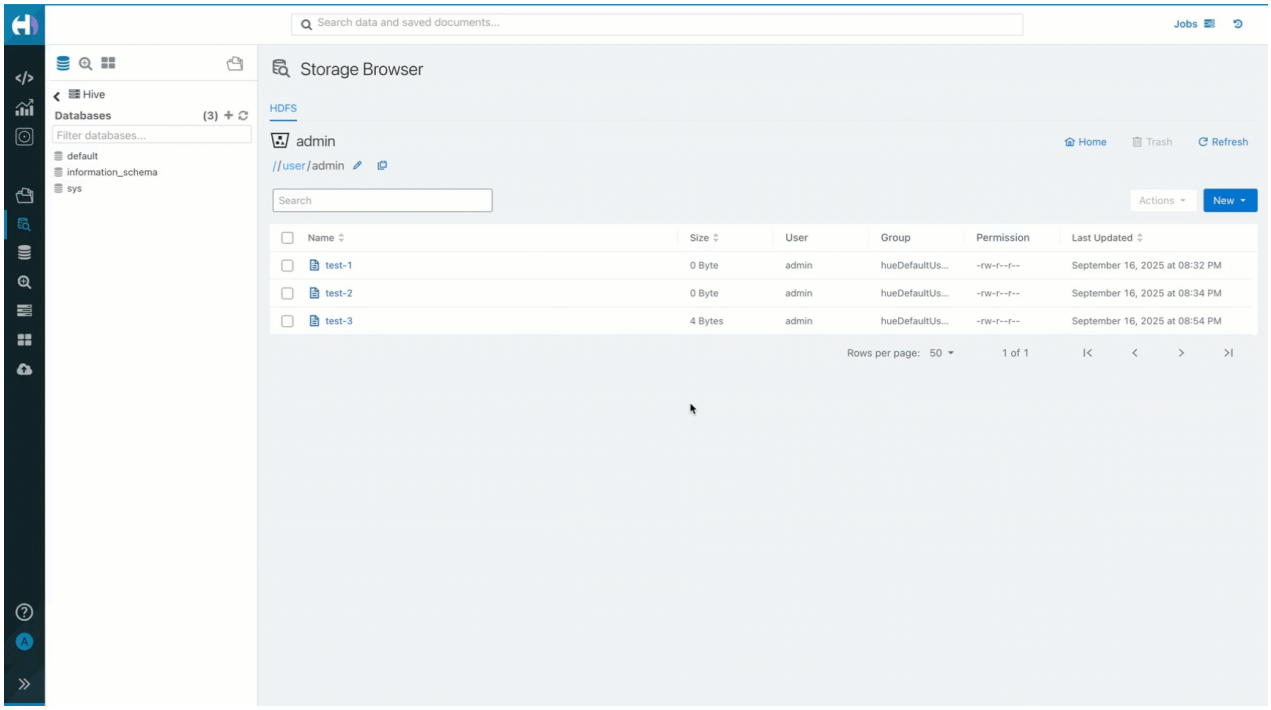

Storage Browser
S3A
ABFS
test
s3a://user/test
Search
Name
test_doc 1.csv
test_doc 10.csv
test_doc 11.csv
test_doc 2.csv
test_doc 3.csv
test_doc 4.csv
test_doc 5.csv
test_doc 6.csv
test_doc 7.csv
test_doc 8.csv

Improved actions and file operations

You can now perform multiple file operations through an intuitive New menu and Action menu.

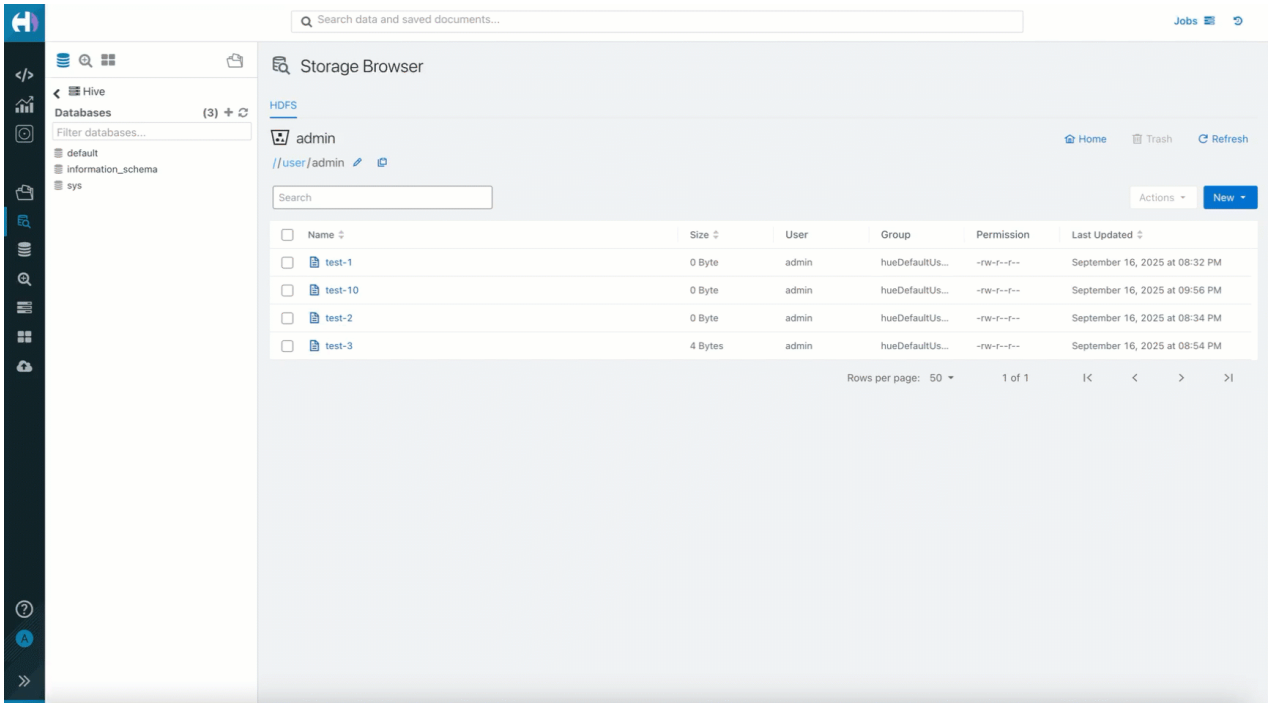
Clicking the New button provides a contextual menu with the following options:

- New File – Create a new file within the selected directory.
- New Folder – Create a new directory.
- Upload File – Directly upload files to the selected location. Drag files from a local system into the Storage Browser for quick uploads.



Click on any file to open the Action menu, which includes the following options:

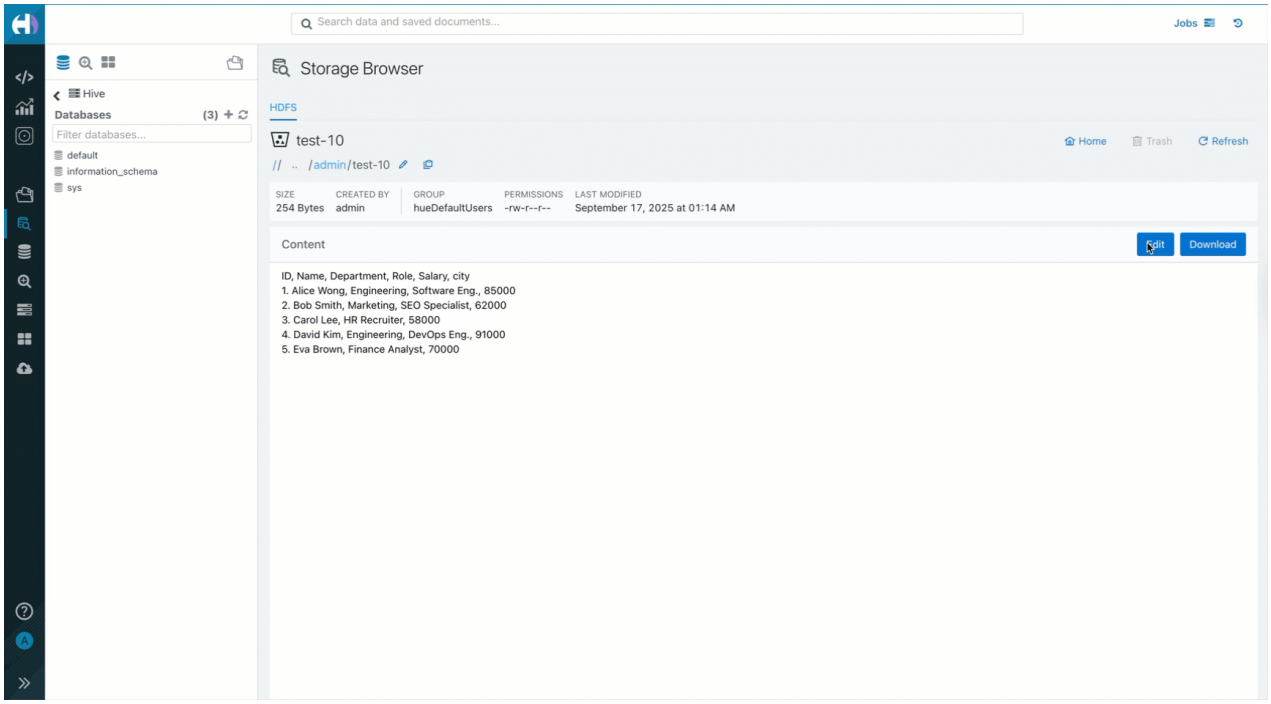
- Rename – You can rename files and directories directly from the Storage Browser, allowing for better organization and easier identification of data.
- Move and Copy – You can move or duplicate files while browsing subdirectories or creating new folders within the selection process using the directory tree view.
- Download – You can quickly download files from storage using the Download button.
- Summary – You can display detailed information about the selected files or directories, including disk space consumed, bytes used, namespace and disk space quotas, replication factor, and counts of files and directories. This overview helps you monitor storage usage and understand the attributes of your data.
- Set Replication – You can configure the replication factor for selected files or directories to define how many copies are stored across the cluster.
- Compress – You can archive the selected files or directories into a compressed format, such as .zip, to reduce storage space and facilitate easier file transfer.
- Delete – You can permanently remove files.
- Compress – You can archive selected files into a compressed format.



File preview and Editing

The Data Explorer Storage Browser includes an integrated file preview and editing system with the following possible actions:

- You can view metadata and content within the browser, with paginated previews for easier navigation.
- If permissions allow, you can Edit and Save files directly from the preview page.
- You can quickly retrieve stored files using the Download button.



Hive query history service

The query history service in Hive provides a scalable solution for storing and querying historical query information in a structured and performant manner, enabling long-term analysis and monitoring.

Cloudera Data Warehouse provides you the option to enable logging Hive queries on an existing Virtual Warehouse or while creating a new Hive Virtual Warehouse.The query history service in Hive is a feature that stores a long-term record of finished queries and their associated metrics. It is designed to support auditing, debugging, and performance monitoring at scale by persisting historical query data in a modern table format.

Hive already offers several ways to inspect query activity, including:

- Hive history .txt files.
- Protobuf logging hook.
- Live queries on the HiveServer2 Web UI.
- SHOW PROCESSLIST command.
- In-development query history service.

While these options allow inspection of active or recent queries, none provide a scalable solution for storing and querying historical query information in a structured and performant manner.

The query history service addresses this limitation by persisting structured records for completed queries, enabling long-term analysis using standard SQL.

Purpose of the query history table

The query history table stores the following information for each finished query:

- Submitting user
- Query runtime
- Tables accessed
- Errors
- Additional metadata fields

This information is stored in a structured format using the Iceberg table format, allowing efficient querying and future integration with tools such as Apache Hue or custom dashboards.

Scope and Limitations

The query history service runs as part of HiveServer2 and writes query data to an Iceberg table in batches, using configurable memory buffering and flushing strategies. However, it is not intended for real-time query inspection, query debugging or recommendations, or for creating user interfaces or visualizations.

Related Information

Impala workload management

Query history table

Learn how the query history service in Hive stores past query data in structured tables using Iceberg and ORC for efficient querying and long-term data storage.

The query history service in Hive is designed to store historical query data in a scalable and queryable format. It uses an Iceberg table backed by the ORC file format, enabling efficient SQL queries for analyzing past workloads.

Why Iceberg and ORC were chosen?

To support scalability and queryability, the service avoids storing one file per query or session. Instead, it uses the Iceberg table format, which provides:

1. Schema evolution
2. Partitioning support
3. Compatibility with SQL engines
4. Efficient metadata management

The ORC format is used for physical storage, offering compact, columnar storage and fast performance for Hive workloads.

The query history service is designed to be pluggable, allowing flexibility to support other storage formats in the future. Hive handles all writing operations, so the service itself does not need to manage storage-level complexity.

Benefits of using Iceberg

The query history service stores data in an Iceberg table instead of traditional file-based logs. Iceberg offers the following advantages:

- Efficient storage and query performance.
- Schema evolution support.
- Partitioning and metadata handling for faster analysis.
- Integration with query engines like Hive and Impala.

Schema structure

The schema of the query history table is organized into fields and partitioning to efficiently store and query historical data.

1. Fields:

   The query history service gathers a wide range of query-related data, categorized into basic and runtime fields. These fields address challenges related to data extraction and integration within HiveServer2. Basic fields are extracted during the service's integration into HiveServer2, leveraging encapsulating objects such as QueryPro perties, QueryInfo, QueryDisplay, and DriverContext. Each of these objects contains overlapping fields relevant to query tracking.While these objects provide valuable query-related information, the service needed to identify a central access point for all necessary data.DriverContext was ultimately chosen as it offers comprehensive access to all required query details, ensuring efficient data extraction and storage.
2. Partitioning:

   The table is partitioned by cluster_id, enabling users to filter queries based on the cluster or compute group that executed them. This partitioning strategy improves query performance and ensures data separation across different environments. For more information, see *HIVE-28324*

Related Information

HIVE-28324

Query history table format

Learn about the query history table format, which provides detailed information on query execution, including column names, data types, and descriptions.

Table Format

The following table provides an overview of the columns in the query history table, including their names, data types, and descriptions:

Column Name	Data type	Description
query_history_schema_version	int	Schema version of the query history record.

Column Name	Data type	Description
hive_version	string	Hive version running in HiveServer2 when the query was executed.
query_id	string	Hive-assigned identifier for the query.
session_id	string	Hive-assigned identifier for the session.
operation_id	string	Hive-assigned identifier for the operation.
execution_engine	string	Execution engine used to run the query (typically Tez).
execution_mode	string	Indicates whether the query ran in LLAP or Tez container mode.
tez_dag_id	string	Tez DAG ID used for the query, if applicable.
tez_app_id	string	Tez application ID used for the query, if applicable.
tez_session_id	string	Tez session ID used for the query, if applicable.
cluster_id	string	Unique identifier for the cluster instance.
sql	string	SQL query string submitted by the user
session_type	string	Session type (HIVESERVER2 or OTHER).
hiveserver2_protocol_version	int	Protocol version used by the client, as defined in TCLIService.
cluster_user	string	Effective user on the cluster (typically hive if doAs is disabled).
end_user	string	Authenticated client username.
db_name	string	Database selected when the query was run.
tez_coordinator	string	Address of the Tez coordinator used for the query.
query_state	string	Query state as an OperationState value.
query_type	string	Query type identified by the semantic analyzer (DQL, DDL, DML, DCL, STATS, or empty).
operation	string	Hive operation name based on syntax or semantic analysis.
server_address	string	Address of the HiveServer2 instance the client connected to.
server_port	int	TCP port of the HiveServer2 instance.
client_address	string	IP address of the client that initiated the connection.
start_time_utc	timestamp	UTC timestamp when the query started.
end_time_utc	timestamp	UTC timestamp when the query finished.
start_time	timestamp	Server-local timestamp when the query started.
end_time	timestamp	Server-local timestamp when the query finished.
total_time_ms	bigint	Total execution time in milliseconds.

* The SQL column in the sys.query_history table captures the full submitted text, including any comments, to ensure no query data is lost.*

Column Name	Data type	Description
planning_duration	bigint	Time spent in query compilation and planning (ms).
planning_start_time	timestamp	Timestamp when query planning started.
prepare_plan_duration	bigint	Time spent preparing the DAG for execution (ms).
prepare_plan_start_time	timestamp	Timestamp when the DAG preparation started.
get_session_duration	bigint	Time taken to acquire a Tez session (ms).
get_session_start_time	timestamp	Timestamp when session acquisition started.
execution_duration	bigint	Duration of DAG execution (ms).
execution_start_time	timestamp	Timestamp when DAG execution started.
failure_reason	string	Error message for query failure, if any.
num_rows_fetched	int	Number of rows fetched by the query.
plan	string	Full text of the query plan.
used_tables	string	Comma-separated list of tables used by the query.
exec_summary	string	Full text of the execution summary.
configuration_options_changed	string	Configuration options changed during the query session.
total_tasks	int	Total number of Tez tasks started for the query.
succeeded_tasks	int	Number of successful tasks.
killed_tasks	int	Number of killed tasks.
failed_tasks	int	Number of failed task attempts.
task_duration_millis	bigint	Total task execution time in milliseconds.
node_used_count	int	Number of nodes used to run the query.
node_total_count	int	Total number of nodes visible during query execution.
reduce_input_groups	bigint	Number of reducer input groups.
reduce_input_records	bigint	Number of input records seen by reducers.
spilled_records	bigint	Number of records spilled during shuffle.
num_shuffled_inputs	bigint	Number of physical inputs used during shuffle.
num_failed_shuffle_inputs	bigint	Number of failed attempts to fetch shuffle inputs.
input_records_processed	bigint	Number of input records actually processed.
input_split_length_bytes	bigint	Total size (bytes) of input splits.
output_records	bigint	Number of output records from all vertices.
output_bytes_physical	bigint	Actual bytes written, including compression.
shuffle_chunk_count	bigint	Number of shuffled files processed.
shuffle_bytes	bigint	Total shuffled data size in bytes.
shuffle_bytes_disk_direct	bigint	Bytes shuffled using direct disk access.
shuffle_phase_time	bigint	Time spent in the shuffle phase (ms).
merge_phase_time	bigint	Time spent merging shuffled data (ms).

Query history use case

The query history service and the sys.query_history table provide developers, support engineers, and customers with tools for detailed investigations and large-scale insights.

The query history service and the sys.query_history table are designed for developers, support engineers, and customers. With a wide range of available fields, you can use the service to:

- Find a specific query and view its details.
- Analyze trends across multiple queries for performance monitoring.

This supports both detailed investigations and large-scale insights.

Example queries

Basic query information

Get key details such as SQL text, DAG ID, execution plan, and summary.

```
SELECT sql, tez_dag_id, plan, exec_summary
FROM sys.query_history
WHERE query_id = 'abc123';
```

Query count by user and database

Find the number of queries run by each user on each database.

```
SELECT db_name, end_user, COUNT(*) AS query_count
FROM sys.query_history
GROUP BY db_name, end_user
ORDER BY db_name;
```

Top five longest-running queries

Identify the five queries with the highest execution time.

```
SELECT end_user, query_id, total_time_ms
FROM sys.query_history
ORDER BY total_time_ms DESC
LIMIT 5;
```

Concurrent query detection

Determine how many queries overlapped with each query.

```
SELECT a.query_id, COUNT(b.query_id) AS concurrent_queries
FROM sys.query_history a
LEFT JOIN sys.query_history b
  ON a.query_id != b.query_id
  AND a.start_time <= b.end_time
  AND b.start_time <= a.end_time
GROUP BY a.query_id
ORDER BY concurrent_queries DESC
LIMIT 5;
```

Extracting counters from execution summary

Use regular expressions to extract specific counters that are not stored in separate fields.

```
SELECT query_id, regexp_extract(exec_summary, 'AM_CPU_MILLISECON
DS: (\\d+)', 1) AS am_cpu_milliseconds
FROM sys.query_history
WHERE tez_dag_id IS NOT NULL;
```

Configuring query history service

Learn how to modify the default configuration of the query history service in Cloudera Hive to manage query history settings more effectively.

Before you begin

Make sure the **Log Hive queries** option is enabled.

Procedure

1. Log in to the Cloudera Data Warehouse service
2. Go to the Virtual Warehouses tab. Select the Hive Virtual Warehouse, click ⋮ Edit
3. Go to the Configurations tab. Select hive-site from the Configuration files drop-down menu, and add the following properties to modify the values as required:
   a) hive.query.history.explain.plan.enabled (Default: true)

      Determines whether to collect and store the explain plan in the query history.
   b) hive.query.history.exec.summary.enabled (Default: true)

      Specifies whether to collect and store the execution summary in the query history.
   c) hive.query.history.batch.size (Default: 200)

      Specifies the maximum number of records that can be held in memory before the query history service persists them to the target table. Smaller values (e.g., 1-5) enable more real-time behavior but result in smaller files. Setting this property to 0 forces synchronous persistence of records, Cloudera does not recommend for production environments
   d) hive.query.history.max.memory.bytes (Default: 20mb)

      Defines the maximum memory size, in bytes, that the query history queue can occupy before the service persists records to the target table. Setting this property to 0 disables the memory limit, Cloudera does not recommend in production environments to prevent excessive memory usage in HiveServer2.
   e) hive.query.history.flush.interval.seconds (Default: 1h)

      Specifies the time interval, in seconds, for flushing query history records from memory to the Iceberg table, regardless of the batch size. This ensures timely access to query history records. The default value is 1 hour, balancing file size and record availability. Setting this property to 0 disables the interval-based flush, relying solely on batch size for persistence.
   f) hive.query.history.repository.class (Default: org.apache.hadoop.hive.ql.queryhistory.repository.IcebergRepository)

      Indicates the class that implements QueryHistoryRepository, which is responsible for persisting query history records.
4. Click **Apply Changes** and restart Hive.

Submitting queries with Hue

You can write and edit queries for Hive, Impala, or Trino Virtual Warehouses in the Cloudera Data Warehouse service by using Hue.

About this task

Note: Trino is available as a Technical Preview offering starting from the Cloudera Data Warehouse on premises 1.5.5 SP1 release. It is available as a GA offering in the Cloudera Data Warehouse on premises 1.5.5 SP2 release.

For detailed information about using Hue, see Using Hue.

Before you begin

Hue uses your LDAP credentials that you have configured for the Cloudera cluster.

Procedure

1. Log into the Cloudera Data Warehouse service as DWUser.
2. Go to the **Virtual Warehouses** tab, locate the Virtual Warehouse using which you want to run queries, and click HUE.

   The Hue query editor opens in a new browser tab.
3. To run a query:
   a) Click a database to view the tables it contains.

      When you click a database, it sets it as the target of your query in the main query editor panel.
   b)

      Type a query in the editor panel and click  to run the query.

      You can also run multiple queries by selecting them and clicking  .

      Note: Use the language reference to get information about syntax in addition to the SQL auto-complete feature that is built in. To view the language reference, click the book icon  to the right of the query editor panel.

Related Information

Advanced Hue configurations (safety valves) in Cloudera Data Warehouse