Cloudera Runtime 7.2.11

Using Apache Phoenix to Store and Access Data

Date published: 2020-02-29
Date modified: 2023-01-18

Legal Notice

© Cloudera Inc. 2024. 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

Mapping Apache Phoenix schemas to Apache HBase namespaces..... 4
Enable namespace mapping..... 4

Associating tables of a schema to a namespace.....5
Associate table in a customized Kerberos environment..... 5
Associate a table in a non-customized environment without Kerberos.....6

Using secondary indexing in Apache Phoenix.....6
Use strongly consistent indexing with Apache Phoenix.....7
Migrate existing tables to use strongly consistent indexing..... 8

Using transactions in Apache Phoenix..... 9
Configure transaction support..... 9
Create and modify tables with transaction support.....11

Using JDBC API with Apache Phoenix..... 13
Connecting to Apache Phoenix Query Server using the JDBC client.....14
Connect to Phoenix Query Server.....14
Connect to Phoenix Query Server through Apache Knox.....15

Using non-JDBC drivers with Apache Phoenix..... 16

Using Apache Phoenix-Spark connector.....16
Configure Phoenix-Spark connector using Cloudera Manager.....16
Phoenix-Spark connector usage examples..... 18

Using Apache Phoenix-Hive connector..... 20
Configure Phoenix-Hive connector using Cloudera Manager..... 20
Apache Phoenix-Hive usage examples..... 21
Limitations of Phoenix-Hive connector..... 22

Mapping Apache Phoenix schemas to Apache HBase namespaces

You can map a Apache Phoenix schema to an Apache HBase namespace to gain multitenancy features in Apache Phoenix.

Important: You must configure this feature only in a CDP Private Cloud Base deployment. This feature is configured automatically in a CDP Public Cloud deployment.

Apache HBase namespaces are a logical grouping of tables, similar to a database in a relational database system. Apache HBase, the underlying storage engine for Apache Phoenix, has namespaces to support multi-tenancy features. Multitenancy helps an Apache HBase user or administrator to perform access control and quota management tasks. Namespaces enable tighter control of where a particular data set is stored on the Apache HBase RegionsServers.

Enable namespace mapping

You can enable namespace mapping by configuring a set of properties using Cloudera Manager.

About this task

After you set the properties to enable the mapping of Phoenix schemas to HBase namespaces, reverting the property settings renders the Phoenix database unusable. Test or carefully plan the Phoenix to HBase namespace mappings before implementing them.

Important: Cloudera recommends that you enable namespace mapping. If you decide not to enable this feature, you can skip the following steps.

To enable Phoenix schema mapping to a non-default HBase namespace:

Procedure

1. Go to the HBase service.
2. Click the Configuration tab.
3. Select Scope (Service-Wide) .
4. Locate the HBase Service Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property or search for it by typing its name in the Search box.
5. Add the following property values:

   Name: phoenix.schema.isNamespaceMappingEnabled

   Description: Enables mapping of tables of a Phoenix schema to a non-default HBase namespace. To enable mapping of a schema to a non-default namespace, set the value of this property to true. The default setting for this property is false.

   Value: true

   Name: phoenix.schema.mapSystemTablesToNamespace

   Description: With true setting (default): After namespace mapping is enabled with the other property, all system tables, if any, are migrated to a namespace called system. With false setting: System tables are associated with the default namespace.

   Value: true
6. Select Scope Gateway .

7. Locate the HBase Client Advanced Configuration Snippet (Safety Valve) for hbase-site.xml property or search for it by typing its name in the Search box.
8. Add the following property values:

   Name: phoenix.schema.isNamespaceMappingEnabled

   Description: Enables mapping of tables of a Phoenix schema to a non-default HBase namespace. To enable mapping of the schema to a non-default namespace, set the value of this property to true. The default setting for this property is false.

   Value: true

   Name: phoenix.schema.mapSystemTablesToNamespace

   Description: With true setting (default): After namespace mapping is enabled with the other property, all system tables, if any, are migrated to a namespace called system.With false setting: System tables are associated with the default namespace.

   Value: true
9. Enter a Reason for change, and then click Save Changes to commit the changes.
10. Restart the role and service when Cloudera Manager prompts you to restart.

Note: If you do not want to map Phoenix system tables to namespaces because of compatibility issues with your current applications, set the phoenix.schema.mapSystemTablesToNamespace property to false.

Associating tables of a schema to a namespace

Important: You must use this feature only in a CDP Private Cloud Base deployment. This feature is configured automatically in a CDP Public Cloud deployment.

After you enable namespace mapping on a Phoenix schema that already has tables, you can migrate the tables to an HBase namespace. The namespace directory that contains the migrated tables inherits the schema name.

For example, if the schema name is store1, then the full path to the namespace is $hbase.rootdir/data/store1. System tables are migrated to the namespace automatically during the first connection after enabling namespace properties.

Associate table in a customized Kerberos environment

You can run a command to associate a table in a customized environment without Kerberos.

Before you begin

In a Kerberos-secured environment, you must have admin privileges (user hbase) to complete the following task.

Procedure

- Run a command to migrate a table of a schema to a namespace, using the following command syntax for the options that apply to your environment:

```
phoenix-psql
ZooKeeper_hostnames:2181
:zookeeper.znode.parent
:principal_name
:HBase_headless_keytab_location
;TenantId=tenant_Id
;CurrentSCN=current_SCN
-m
schema_name.table_name
```

Associate a table in a non-customized environment without Kerberos

You can run a command to associate a table in a non-customized environment without Kerberos.

Procedure

- Run the following command to associate a table:

```
phoenix-psql ZooKeeper_hostname -m Schema_name.table_name
```

Using secondary indexing in Apache Phoenix

Apache Phoenix uses a secondary index to serve queries. An index table is an Apache Phoenix table that stores the reference copy of some or all the data in the main table.

You can use a secondary index to access data from its primary data access path. When you use a secondary index, the indexed column qualifiers or rows form a unique row key that allows you to do point lookups and range scans.

Secondary index types

Apache Phoenix supports global and local secondary indexes. Global indexes is used for all typical use cases. You can use local indexes for specific use cases where you want the primary and the index table to be present in the same Apache HBase region.

- Use global indexes for read-heavy use cases. Use the covered-global index to save on read-time overheads. Global indexes are used to co-locate related information.
- Use local indexes for write-heavy use cases. Use the functional-local index on arbitrary expressions to query specific combinations of index queries. A local index is an in-partition index that is optimized for writes but requires more data to be read to answer a query.

Note: If your table is large, you can use the ASYNC keyword with CREATE INDEX to create an index asynchronously. ASYNC uses a MapReduce job to distribute index workload, and every single mapper works with each data table region and writes to the index region. Therefore freeing up critical resources during indexing.

The following tables list the index type and index scope with a description and example for each index type:

Table 1:

Index type	Description
Covered	Include the data that you want to access from the primary table in the index rows. The query does not have to access the primary table once the index entry is found. Benefits: Save read-time overhead by only accessing the index entry. In the following example, column v3 is included in the index to avoid the query to access the primary table to retrieve this information. Example The following command creates indexes on the v1 and v2 columns and include the v3 column as well: `CREATE INDEX my_index ON exp_table ( v1,v2) INCLUDE(v3);`

Index type	Description

Functional	Create an index on arbitrary expressions. When your query uses the expression, the index is used to retrieve the results instead of the data table.
	Benefits: Useful for certain combinations of index queries.
	Example
	Run the following command to create a functional index so that you can perform case insensitive searches on the combined first name and last name of a person:
	`CREATE INDEX UPPER_NAME ON EMP (UPPER(FIRST_NAME||' '||LAST_NAME));`
	Search on the combined first name and last name using the following command:
	`SELECT EMP_ID FROM EMP WHERE UPPER(FIRST_NAME||' '||LAST_NAME)='EXP_NAME';`

Table 2:

Index scope	Description
Global	You can use this when you have read-heavy use cases. Each global index is stored in its own table, and therefore it is not co-located with the data table. A Global index is a covered index. It is used for queries only when all columns in that query are included in that index. Example Run the following command to create a global index: `CREATE INDEX my_index ON exp_table (v1);`
Local	You can use this for write-heavy use cases. Each local index is stored within the data table. Example Run the following command to create a local index: `CREATE LOCAL INDEX my_index ON exp_table (v1);`

Use strongly consistent indexing with Apache Phoenix

Strongly consistent indexing is a three-phase indexing algorithm that ensures that the index table is always in sync with the data table.

Strongly consistent indexing ensures that the data read from the index table is consistent at read time for read operations. An index table row is repaired using the corresponding data table row if necessary at read time. When the writes to a table are fast, an additional write phase is used without impacting the write performance.

The following table illustrates the design of the three-phase indexing:

Table 3:

Operation	Strongly consistent index
READ	**1.** Reads the index rows and checks its status **2.** Repairs unverified rows using the underlying data table
WRITE	**1.** Sets the status of the existing index rows as unverified and writes the new index rows with status as unverified **2.** Writes to the underlying data table rows **3.** Deletes the exisiting index rows and sets the status oif the new rows as verified
DELETE	**1.** Sets the status of index table rows as unverified **2.** Deletes the underlying data table rows **3.** Deletes the index table rows

Newly created Apache Phoenix tables in Cloudera Runtime 7.1.6 and higher use strongly consistent indexing by default. You can use the index upgrade tool to upgrade your Apache Phoenix tables to use the strongly consistent indexing.

Migrate existing tables to use strongly consistent indexing

You can use the index upgrade tool to rebuild your indexes. After you run this tool, your existing tables use the strongly consistent global index.

About this task

If you have Apache Phoenix tables that you migrated from Cloudera Runtime 7.1.5 or lower to Cloudera Runtime 7.1.6 or higher, you must convert the tables to use the strongly consistent index using the *index upgrade tool*.

Procedure

1. Perform the following as the HBase user, or as a user with HBase root privileges (global RXWCA).
2. Run the following command for each table that has global indexes.

```
hbase org.apache.phoenix.mapreduce.index.IndexUpgradeTool -o [***UPGRADE /
 ROLLBACK***] -tb [***TABLE NAME***]
-lf [***/tmp/index-upgrade-tool.log***]
```

Parameters:

- -o: Type whether you want to upgrade or rollback.
- -tb: Type the table name that you want to upgrade or rollback.
- -lf: Type the path where you want to store the *index upgrade tool* log file.

3. After the conversion is finished, it is highly recommended to rebuild the indexes. Use the following commands for each index.

```
hbase org.apache.phoenix.mapreduce.index.IndexTool -s [***SCHEMA NAME***]
 -dt [***DATA TABLE NAME***] -it[***INDEX NAME***] -op /tmp/phoenix_index
```

```
hbase org.apache.phoenix.mapreduce.index.IndexTool -fi -s [***SCHEMA
 NAME***] -dt [***DATA TABLE NAME***] -it[***INDEX NAME***] -op /tmp/ph
oenix_index
```

Parameters:

- -s: Type the schema name. Optional for default schema.
- -dt: Type the data table name.
- -it: Type the index name.
- -op: Type the path where you want to store the *index upgrade tool* log file.

Note: These commands start the MapReduce jobs in the background, which might take a longer time and cause significant load on the cluster.

Using transactions in Apache Phoenix

X-row, X-table transaction support enables support for complex distributed transactions. Transaction support enables you to run atomic database operations, meaning your database operations must either be completed or terminated. A transaction may contain a set of database operations where each one is carried out atomically without interfering with others.

You can perform complex transactions that span multiple rows and tables and implement traditional star schema, wide-columns, or both. Transactions are lock-free and handle any write-write conflicts for you without your intervention.

This two-phase commit capability is implemented using ANSI SQL semantics which is the same as used in other databases like MySQL or PostgreSQL, but this implementation benefits from the near-linear scalability of Apache HBase and Apache Phoenix, and do not have upper-bound on the number of transactions, size of rows/table, or size per table like other databases.

Note: You cannot convert a transactional table back to a non-transactional table. Ensure that you only convert those tables that you know will use transactions.

Configure transaction support

Before you can start using transactions, you must first add the Apache Omid service and configure Apache Phoenix to enable transactions. Apache Omid provides the transaction management capability in Apache Phoenix.

Before you begin

- If your cluster is Kerberized, run the kinit command as hbase user before adding the Apache Omid service to the cluster. Ensure you have valid Kerberos credentials. You can list the Kerberos credentials using the klist command.

  or

  If you do not want to kinit as a hbase user, you can use Apache Ranger in the SDX Data Lake to grant the user permissions to CREATE, READ and WRITE on the commit table (default table name OMID_COMMIT_TABLE) and timestamp table (default table name OMID_TIMESTAMP_TABLE).

  Apache Omid creates and writes to OMID_COMMIT_TABLE and OMID_TIMESTAMP_TABLE during a transaction.
- Ensure that the Apache Phoenix client doing transactions should have minimum WRITE permissions on commit table (default table name OMID_COMMIT_TABLE).

About this task

Add the Apache Omid service, and configure Apache Phoenix using Cloudera Manager.

Procedure

1. On the Home Status tab, click to the right of the cluster name and select Add a Service. A list of service types display. You can add one type of service at a time.
2. Select Omid and click Continue.
3. Click Finish to complete adding the Omid service. Return to the Cloudera Manager home page.

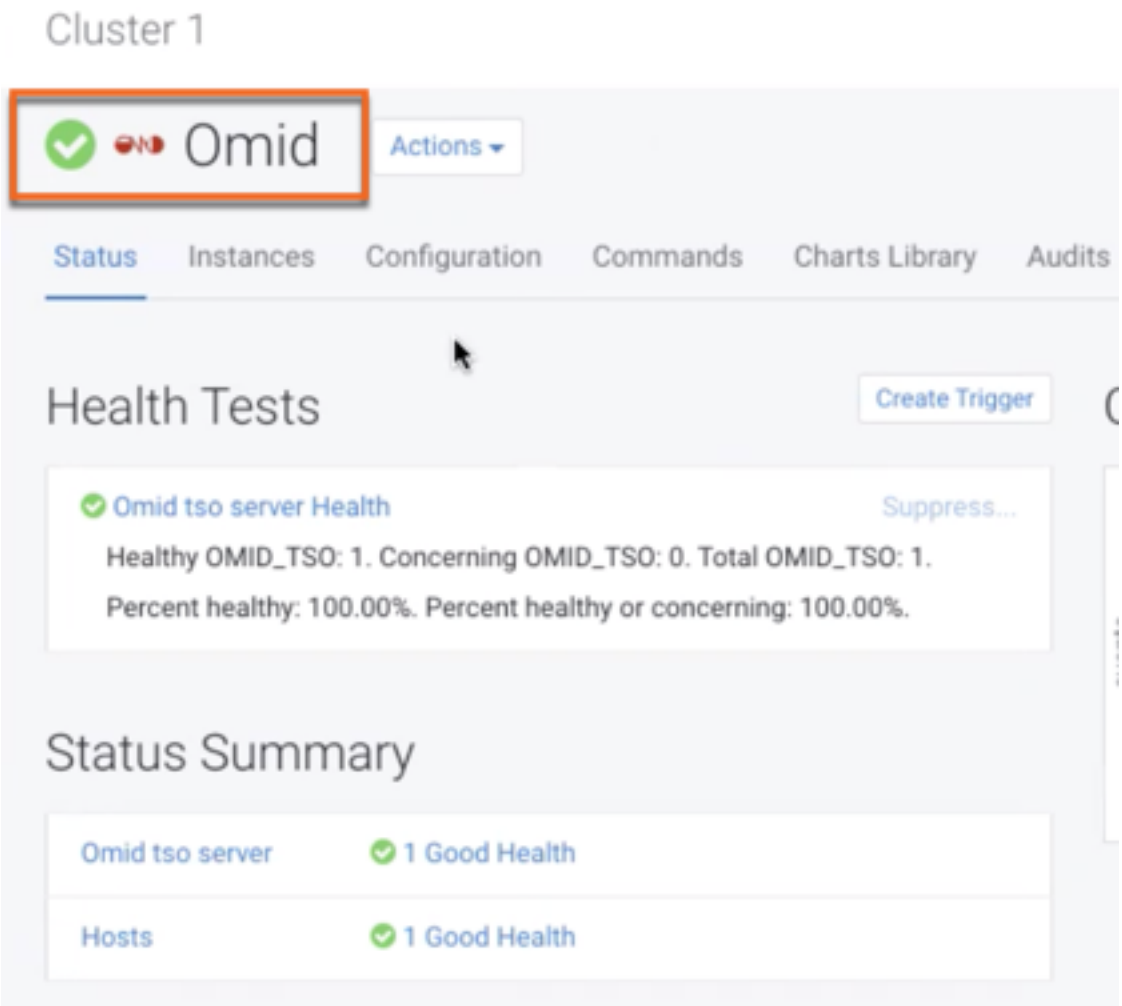
   The Phoenix OMID TSO service listens to the 54758 port by default, which is in the ephemeral port range. Because of this, the service might randomly fail to start if the port is already used.

   To avoid this, change the TSO port to 24758. Follow these steps to change the default port.

   a) Click on the Omid service.
   b) Go to Configuration and search for OMID TSO server port.
   c) Modify the port to 24758.
   d) Enter a Reason for change, and then click Save Changes.
   e) Click (Stale configuration).
   f) Restart the Omid service when Cloudera Manager prompts you to restart.

   Note: You need to change the port manually in all the Omid configuration files that are not managed by Cloudera Manager.

4. Verify the new service is started properly by checking the health status of the new service. If the Health Status is Good, then the service started properly.



5. After the Omid service has started, go to the HBase service.
6. Click the Configuration tab.
7. Select Scope > Gateway.
8. Select Category > Advanced.
9. Locate the HBase Client Advanced Configuration Snippet property or search for HBase Client Advanced Configuration Snippet (Safety Valve) for hbase-site.xml:

   Click View as XML and add the following property value:

```
<property>
  <name>phoenix.transactions.enabled</name>
  <value>true</value>
</property>
```

10. Enter a Reason for change, and then click Save Changes to commit the changes.
11. Restart the role and service when Cloudera Manager prompts you to restart.

Create and modify tables with transaction support

You can create new tables with transaction support enabled or alter an existing table to enable transaction support on that table.

About this task

To enable transaction support, you must use the TRANSACTIONAL=true parameter when creating a table, followed by the COMMIT command at the end of your transaction.

Procedure

- Create a table with transaction support enabled.

```
CREATE TABLE NEW_TABLE (ID INT) TRANSACTIONAL=true; TRANSACTION_PROVIDER
='OMID';
```

> **Note:** TRANSACTION_PROVIDER is an optional parameter. Omid is used as the default transaction provider if this parameter is not specified.

- Set autocommit to off before you start a transaction.

  The following example shows you how to turn off autocommit and then upsert a value into NEW_TABLE.

```
jdbc:phoenix:> !autocommit off
Autocommit status: false
jdbc:phoenix:> upsert into NEW_TABLE values(1,'ph');
1 row affected (0.015 seconds)
jdbc:phoenix:> select * from NEW_TABLE;
+---+----+
| A | B  |
+---+----+
| 1 | ph |
+---+----+
1 row selected (0.144 seconds)
jdbc:phoenix:> upsert into NEW_TABLE values(2,'ph');
1 row affected (0.001 seconds)
jdbc:phoenix:> !commit
```

- Modify a table to enable transactions on the table.

```
ALTER TABLE EXISTING_TABLE SET TRANSACTIONAL=true;
```

Example

The following example shows you how to do a transaction on a table when transaction support is enabled:

```
!autocommit off -- Turn off autocommit
SELECT * FROM example_table; -- Transaction starts here. Other transactions
 cannot see your updates yet
UPSERT INTO example_table VALUES (1,'A');
SELECT count(*) FROM example_table WHERE e=1; -- You can see an uncommited
row
DELETE FROM my_other_table WHERE e=2;
!commit -- Other transactions can now see the updates
```

You will see an exception if you try to commit a row that is conflicting with another transaction. In the following example, you will see an exception when you try and commit the first transaction after committing the second transaction because the second transaction updates the same row.

```
UPSERT INTO NEW_TABLE VALUES (1,'a');
```

Now, when you do a commit on the same row. Row 1 in the following example, you will see an error.

```
UPSERT INTO NEW_TABLE VALUES (1,'b');
!commit
```

You see an exception similar to the following:

```
java.sql.SQLException: ERROR 523 (42900): Transaction aborted due to conflic
t with other mutations. Conflict detected for transaction 155612256680000000
0.
```

Using JDBC API with Apache Phoenix

You can create and interact with Apache HBase tables using Phoenix DDL/DML statements through its standard JDBC API. Apache Phoenix JDBC driver can be easily embedded in any application that supports JDBC.

Apache Phoenix enables you to use the standard JDBC API to create and access Apache HBase tables. You can use JDBC APIs with Apache Phoenix instead of native Apache HBase client APIs to create tables, insert, and query data.

Apache Phoenix tables have a 1:1 relationship with Apache HBase tables. You can choose to create a new table using an Apache Phoenix DDL statement such as CREATE TABLE, or create a view on an existing Apache HBase table using the VIEW statement.

Important: Modifying an Apache Phoenix table using Apache HBase native API is not supported. Doing this leads to errors, inconsistent indexes, incorrect query results, or sometimes to corrupt data.

To use the Apache Phoenix JDBC driver, you must embed the driver in your application that supports JDBC. Apache Phoenix has two kinds of JDBC drivers.

- A *thick driver* communicates directly with Apache HBase. The thick driver, therefore, needs access to all the nodes in the Apache HBase cluster.
- A *thin driver* communicates with Apache HBase through Phoenix Query Server (PQS) and requires access only to PQS. Use the thin driver to connect to PQS through Apache Knox or connect to PQS directly.

In an operational database Data Hub cluster, Data Lake (SDX cluster) provides security dependencies such as Apache Knox. Your JDBC URL string would depend on whether you want to connect directly or through Apache Knox. Before you try connecting to Apache Phoenix, check if you are in the list of allowed users in Apache Ranger allowed to access Apache Phoenix and Apache HBase.

Based on whether you want to use the thick or thin driver, you need the JAR files for the Apache HBase client, the Apache Phoenix client, and the PQS client.

For the thick driver, you need:

- hbase-client-[***VERSION***].jar
- hbase-site.xml

Note: You must add the cluster's current hbase-site.xml to the application classpath. You can get the hbase-site.xml by doing an SSH to the cluster node with the hbase-gateway role. You can copy the hbase-site.xml file from the following path /etc/hbase/hbase-site.xml or /etc/hbase/conf/hbase-site.xml.

For the thin driver, you need:

- phoenix-queryserver-client-[***VERSION***].jar

You can get these JAR files from the following locations:

- Go to /opt/cloudera/parcels/CDH/lib/phoenix/ on an operational database cluster node with the phoenix-gateway role

  or
- Download the JAR files from the Cloudera Repository

When using the thin driver, your applications interact with the Phoenix Query Server using the Avatica API and Google Protocol Buffers serialization format.

JDBC driver location

Use the /opt/cloudera/parcels/CDH/lib/phoenix/[***PHOENIX VERSION***].jar file present in your deployment location. For example, /opt/cloudera/parcels/CDH/lib/phoenix/phoenix-5.0.0.7.2.0.0-128-client.jar

URL syntax for the thick JDBC driver

To connect to Apache Phoenix using the thick JDBC driver, you must use the following JDBC URL syntax:

```
jdbc:phoenix:[***zookeeper_quorum***]:[***zookeeper_port***]:
[***zookeeper_hbase_path***]
```

The zookeeper_quorum and zookeeper_port parameters are optional if you have added the operational database Apache HBase cluster's current hbase-site.xml to the application classpath.

Apart from the JDBC driver, the following drivers are supported:

- ODBC driver
- Python driver for Phoenix

Connecting to Apache Phoenix Query Server using the JDBC client

You can interact with Apache Phoenix using your client and Apache Phoenix Query Server (PQS).

PQS is automatically configured when you create an Operational Database Data Hub cluster. There are two ways in which you can use the thin client to interact with Phoenix:

- Connect to PQS directly
- Connect to PQS using the Apache Knox gateway

Connect to Phoenix Query Server

You can connect to Phoenix Query Server (PQS) using the JDBC thin client without using a gateway such as Apache Knox. You must use the JDBC URL syntax to form the URL and connect to PQS.

Before you begin

Ensure that you have access to Apache Phoenix and Apache HBase, and you have the required permissions configured in Ranger to connect to PQS.

Ensure that you have safely noted down your Kerberos principal and keytab information used to connect to PQS.

About this task

You can use the JDBC URL syntax to form the URL and connect to PQS.

Procedure

- To connect to the PQS directly, you must use the JDBC URL syntax as shown here: jdbc:phoenix:thin:[key=value [;key=value...]]

  You must provide the correct URL, serialization, and authentication key-values to interact with the Phoenix Query Server. For more information about optional parameters, see Client Reference.

Example

```
jdbc:phoenix:thin:url=http://localhost:8765;serialization=PROTOBUF; authenti
cation=SPNEGO;
principal=[***Prinicpal@EXAMPLE.com***];keytab=[***PATH TO THE KEYTAB
 FILE***]
```

Connect to Phoenix Query Server through Apache Knox

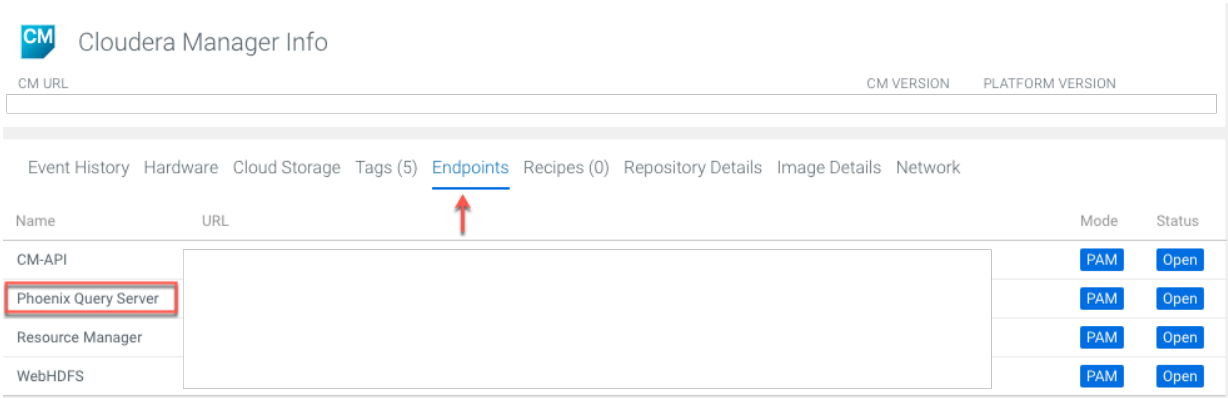
You can connect to Phoenix Query Server (PQS) using the JDBC thin client through the Apache Knox gateway. Apache Knox requires your thin client connection to be over HTTPS.

Before you begin

Ensure that you have access to Apache Phoenix and Apache HBase, and you have the required permissions configured in Ranger to connect to PQS.

Procedure

- Get the PQS Knox endpoint to connect to PQS from the CDP Data Hub user interface. Go to Data Hub cluster Endpoints Cloudera Manager Info Endpoints Phoenix Query Server .



- Use the JDBC URL in the following sytax to connect to PQS through the Apache Knox gateway:

  jdbc:phoenix:thin:url=https://[***KNOX ENDPOINT***]:[***PORT***]/[***CLUSTER NAME***]/cdp-proxy-api/avatica/;serialization=PROTOBUF;authentication=BASIC;avatica_user=[***WORKLOAD USERNAME*];avatica_password=[***WORKLOAD PASSWORD***];truststore=[***PATH TO THE KNOX TRUSTSTORE .jks FILE***];truststore_password=[***TRUSTSTORE PASSWORD***]

  The standard Oracle Java JDK distribution includes a default truststore (cacerts) that contains root certificates for many well-known CAs, including Symantec. Rather than using the default truststore, Cloudera recommends using the alternative truststore, jssecacerts. The alternative truststore is created by copying cacerts to that filename (jssecacerts). Certificates can be added to this truststore when needed for additional roles or services. This alternative truststore is loaded by Hadoop daemons at startup. Password (if there is one for the truststore) stored in a plaintext file readable by all (OS filesystem permissions set to 0440). For more information about truststores, see Understanding Keystores and Truststores.

  Note: The user name and password are for the Apache Knox gateway, and the authentication must always be set to BASIC. The truststore and truststore_password the Knox public certificate.

Example

```
jdbc:phoenix:thin:url=https://[***https://pqs.knox.endpoint:8443/gateway/
cdp-proxy-api/avatica/***]:[***8765***]
/[***clusteropdb***]/cdp-proxy-api/avatica/;serialization=PROTOBUF;
authentication=BASIC;avatica_user=[***WORKLOADUSERNAME***]
;avatica_password=[***WORKLOADPASSWORD***]
;truststore=[***/home/path/truststore.jks***];truststore_password
=[***TRUSTSTOREPASSWORD***]
```

Using non-JDBC drivers with Apache Phoenix

Based on your application development requirements, you can obtain one of the other non-JDBC drivers.

The list of supported programming languages to access data stored in your operational database:

- Apache Groovy
- C
- C++
- Go
- Java
- Jython
- Python
- PHP
- Scala

You can also use REST for API support.

The following drivers are supported:

ODBC driver

Use the Database Open Database Connectivity (ODBC) interface to access the operational database. The ODBC driver is provided by Cloudera as an additional download, but you can also use ODBC drivers from third-party providers.

You can download the Phoenix ODBC Driver from here: Phoenix ODBC Connector for CDP Operational Database.

Note

You must have a Cloudera Enterprise Support Subscription to download the ODBC driver.

Python driver for Phoenix

Download the Python Driver for Apache Phoenix from the Apache Phoenix website. For more information, see Python Driver for Phoenix.

Other drivers that are not supported by Cloudera

You can use other non-JDBC Drivers for Phoenix as add-ons, but they are not supported by Cloudera. You can find compatible client drivers by searching on the web for avatica and the name of an application programming language that you want to use. For example, Apache Phoenix/Avatica SQL driver for Go language. For more information and links to driver download, see Apache Calcite Avatica.

You can use the Phoenix ODBC driver and the C# client library to develop .Net applications that interact with Phoenix Query Server.

The applications that you develop will interact with the Phoenix Query Server using the Avatica API and Google Protocol Buffers serialization format.

Using Apache Phoenix-Spark connector

You can use the Apache Phoenix-Spark connector on your secure clusters to perform READ and WRITE operations. The Phoenix-Spark connector allows Spark to load Phoenix tables as Resilient Distributed Datasets (RDDs) or DataFrames and lets you save them back to Phoenix.

Configure Phoenix-Spark connector using Cloudera Manager

Procedure

1. Follow Step 1 to Step 7(b) as mentioned in *Configure HBase-Spark connector using Cloudera Manager*.
2. Add the required properties to ensure that all required Phoenix and HBase platform dependencies are available on the classpath for the Spark executors and drivers.
   a) Upload the Phoenix-Spark3 connector file:

   ```
   hdfs dfs -put /opt/cloudera/parcels/cdh/lib/phoenix_connectors/phoenix5-
   spark3-shaded.jar /path/hbase_jars_spark3
   ```

   b) Upload the Phoenix-Spark2 connector file:

   ```
   hdfs dfs -put /opt/cloudera/parcels/cdh/lib/phoenix_connectors/phoenix5-
   spark-shaded.jar /path/hbase_jars_spark2
   ```

   c) Add all the Phoenix-Spark connector related files that you have just uploaded in the previous steps to the spark.jars parameter:

   Important: Ensure that you add the Phoenix-Spark connector JARs to the configuration properties and you do not remove any of these files added in Step 1.

   - Spark2:

   ```
   spark.jars=hdfs:///path/hbase_jars_common/hbase-site.xml.jar,hdfs://
   /path/hbase_jars_spark2/hbase-spark-protocol-shaded.jar,hdfs:///path
   /hbase_jars_spark2/hbase-spark.jar,hdfs:///path/hbase_jars_spark2/sc
   ala-library.jar,/path/hbase_jars_common(other common files)...,hdfs:
   ////path/hbase_jars_spark2/phoenix5-spark-shaded.jar
   ```

   - Spark3:

   ```
   spark.jars=hdfs:///path/hbase_jars_common/hbase-site.xml.jar,hdfs://
   /path/hbase_jars_spark3/hbase-spark3.jar,hdfs:///path/hbase_jars_spa
   rk3/hbase-spark3-protocol-shaded.jar,/path/hbase_jars_common(other c
   ommon files)...,hdfs:////path/hbase_jars_spark3/phoenix5-spark3-shad
   ed.jar
   ```

3. Enter a Reason for change, and then click Save Changes to commit the changes.
4. Restart the role and service when Cloudera Manager prompts you to restart.

What to do next

Build a Spark or Spark3 application using the dependencies that you provide when you run your application. If you follow the previous instructions, Cloudera Manager automatically configures the connector for Spark. If you have not:

- Consider the following example while using a Spark2 application:

```
spark-shell --conf spark.jars=hdfs:///path/hbase_jars_common/hbase-site.
xml.jar,hdfs:///path/hbase_jars_spark2/hbase-spark-protocol-shaded.jar,h
dfs:///path/hbase_jars_spark2/hbase-spark.jar,hdfs:///path/hbase_jars_sp
ark2/scala-library.jar,hdfs:///path/hbase_jars_common/hbase-shaded-mapre
duce-***VERSION NUMBER***.jar,hdfs:///path/hbase_jars_common/opentelemet
ry-api-***VERSION NUMBER***.jar,hdfs:///path/hbase_jars_common/opentelem
etry-context-***VERSION NUMBER***.jar,hdfs:////path/hbase_jars_spark2/ph
oenix5-spark-shaded.jar
```

- Consider the following example while using a Spark3 application:

```
spark3-shell --conf spark.jars=hdfs:///path/hbase_jars_common/hbase-site
.xml.jar,hdfs:///path/hbase_jars_spark3/hbase-spark3-protocol-shaded.jar
,hdfs:///path/hbase_jars_spark3/hbase-spark3.jar,hdfs:///path/hbase_jars
_common/hbase-shaded-mapreduce-***VERSION NUMBER***.jar,hdfs:///path/hba
```

```
se_jars_common/opentelemetry-api-***VERSION NUMBER***.jar,hdfs:///path/h
base_jars_common/opentelemetry-context-***VERSION NUMBER***.jar,hdfs:////
path/hbase_jars_spark3/phoenix5-spark3-shaded.jar
```

Related Information

Configure HBase-Spark connector using Cloudera Manager

Phoenix-Spark connector usage examples

Learn about the Phoenix-Spark connector examples which includes reading the Phoenix tables, saving the Phoenix tables, and using the PySpark to READ and WRITE tables.

Reading Phoenix tables

For example, you have a Phoenix table with the following DDL, you can use one of the following methods to load the table:

- As a DataFrame using the Data Source API
- As a DataFrame using a configuration object
- As an RDD using a Zookeeper URL

```
CREATE TABLE TABLE1 (ID BIGINT NOT NULL PRIMARY KEY, COL1 VARCHAR);
UPSERT INTO TABLE1 (ID, COL1) VALUES (1, 'test_row_1');
UPSERT INTO TABLE1 (ID, COL1) VALUES (2, 'test_row_2');
```

Example: Load a DataFrame using the Data Source API

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import org.apache.phoenix.spark._


//delete the following line when running fom spark-shell
val sc = new SparkContext("local", "phoenix-test")
val sqlContext = new SQLContext(sc)
val df = sqlContext.load(
  "org.apache.phoenix.spark",
  //replace "phoenix-server:2181" with the real ZK quorum
  Map("table" -> "TABLE1", "zkUrl" -> "phoenix-server:2181")
)

df
  .filter(df("COL1") === "test_row_1" && df("ID") === 1L)
  .select(df("ID"))
  .show
```

Example: Load as a DataFrame directly using a Configuration object

```
import org.apache.hadoop.conf.Configuration
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import org.apache.phoenix.spark._

val configuration = new Configuration()
// Can set Phoenix-specific settings, requires 'hbase.zookeeper.quorum'

//delete the following line when running fom spark-shell
val sc = new SparkContext("local", "phoenix-test")
val sqlContext = new SQLContext(sc)
// Loads the columns 'ID' and 'COL1' from TABLE1 as a DataFrame
```

```
val df = sqlContext.phoenixTableAsDataFrame(
  "TABLE1", Array("ID", "COL1"), conf = configuration
)

df.show
```

Example: Load as an RDD using a Zookeeper URL

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import org.apache.phoenix.spark._
import org.apache.spark.rdd.RDD

//delete the following line when running fom spark-shell
val sc = new SparkContext("local", "phoenix-test")

// Loads the columns 'ID' and 'COL1' from TABLE1 as an RDD
val rdd: RDD[Map[String, AnyRef]] = sc.phoenixTableAsRDD(
//replace "phoenix-server:2181" with the real ZK quorum
  "TABLE1", Seq("ID", "COL1"), zkUrl = Some("phoenix-server:2181")
)

rdd.count()

val firstId = rdd.first()("ID").asInstanceOf[Long]
val firstCol = rdd.first()("COL1").asInstanceOf[String]
```

Saving Phoenix tables

You can refer to the following examples for saving RDDs and DataFrames.

Example: Saving RDDs

For example, you have a Phoenix table with the following DDL, you can save it as an RDD.

```
CREATE TABLE OUTPUT_TEST_TABLE (id BIGINT NOT NULL PRIMARY KEY, col1 VARCHAR
, col2 INTEGER);
```

The saveToPhoenix method is an implicit method on RDD[Product], or an RDD of Tuples. The data types must correspond to one of the Java types supported by Phoenix.

```
import org.apache.spark.SparkContext
import org.apache.phoenix.spark._

//delete the following line when running fom spark-shell
val sc = new SparkContext("local", "phoenix-test")
val dataSet = List((1L, "1", 1), (2L, "2", 2), (3L, "3", 3))

sc
  .parallelize(dataSet)
  .saveToPhoenix(
    "OUTPUT_TEST_TABLE",
    Seq("ID","COL1","COL2"),
    //replace "phoenix-server:2181" with the real ZK quorum
    zkUrl = Some("phoenix-server:2181")
  )
```

Example: Saving DataFrames

The save is method on DataFrame allows passing in a data source type. You can use org.apache.phoenix.spark, and must also pass in a table and zkUrl parameter to specify which table and server to persist the DataFrame to. The column names are derived from the DataFrame's schema field names, and must match the Phoenix column names.

The save method also takes a SaveMode option, for which only SaveMode.Overwrite is supported. For example, you have a two Phoenix tables with the following DDL, you can save it as a DataFrames.

Using PySpark to READ and WRITE tables

With Spark's DataFrame support, you can use pyspark to READ and WRITE from Phoenix tables.

Example: Load a DataFrame

Given a table TABLE1 and a Zookeeper url of localhost:2181, you can load the table as a DataFrame using the following Python code in pyspark:

```
df = sqlContext.read \
  .format("org.apache.phoenix.spark") \
  .option("table", "TABLE1") \
  .option("zkUrl", "localhost:2181") \
  .load()
```

Example: Save a DataFrame

Given the same table and Zookeeper URLs above, you can save a DataFrame to a Phoenix table using the following code:

```
df.write \
  .format("org.apache.phoenix.spark") \
  .mode("overwrite") \
  .option("table", "TABLE1") \
  .option("zkUrl", "localhost:2181") \
  .save()
```

Note: The functions phoenixTableAsDataFrame, phoenixTableAsRDD and saveToPhoenix all support optionally specifying a conf Hadoop configuration parameter with custom Phoenix client settings, as well as an optional zkUrl parameter for the Phoenix connection URL. If zkUrl isn't specified, it's assumed that the hbase.zookeeper.quorum property has been set in the conf parameter. Similarly, if no configuration is passed in, zkUrl must be specified.

Using Apache Phoenix-Hive connector

This connector enables you to access the Phoenix data from Hive without any data transfer. So the Business Intelligence (BI) logic in Hive can access the operational data available in Phoenix.

Using this connector, you can run a certain type of queries in Phoenix more efficiently than using Hive or other applications, however, this is not a universal tool that can run all types of queries. In some cases, Phoenix can run queries faster than the Phoenix Hive integration and vice versa. In others, you can run this tool to perform operations like many to many joins and aggregations which Phoenix would otherwise struggle to effectively run on its own. This integration is better suited for performing online analytical query processing (OLAP) operations than Phoenix.

Another use case for this connector is transferring the data between these two systems. You can use this connector to simplify the data movement between Hive and Phoenix, since an intermediate form of the data (for example, a .CSV file) is not required. The automatic movement of structured data between these two systems is the major advantage of using this tool. You should be aware that for moving a large amount of data from Hive to Phoenix CSV bulk load is preferable due to performance reasons.

Configure Phoenix-Hive connector using Cloudera Manager

You must configure Phoenix-Hive connector before you can access Phoenix data from Hive.

Before you begin

Ensure that you have the phoenix-hive-version.jar. You can find this JAR file in the /opt/cloudera/parcels/CDH/lib/phoenix_connectors/ path on the Apache Phoenix node, or you can download it from the Cloudera respository. If you use Hive-Tez, you must configure both the Hive and Hive-Tez services.

About this task

To configure the Phoenix-Hive connector using Cloudera Manager:

Procedure

1. Go to the Hive service.
2. Click the Configuration tab.
3. Select Scope Hive Cluster (Service-Wide) .
4. Select Category Advanced .
5. Locate the Hive Auxiliary JARs Directory property or search for it by typing its name in the Search box.
6. Add the following auxiliary path directory: */usr/local/phoenix-hive*

   Important: You must manually create the */usr/local/phoenix-hive* directory, and copy the /opt/cloudera/parcels/CDH/lib/phoenix_connectors/phoenix5-hive-[***VERSION***]-shaded.jar on every node in the cluster that runs Hive-on-Tez Server or Hive Metastore.

   Ensur that you have the required permissions the access and write to the phoenix-hive directory and the JAR file that you copy into it must be accessible by hive:hive user.

   Note: You can use any directory instead of */usr/local/phoenix-hive* that Hive can read or place the JAR file in the existing Hive Auxiliary Directory if one already exists.

7. Enter a Reason for change, and then click Save Changes to commit the changes.
8. Restart the role and service when Cloudera Manager prompts you to restart.

What to do next

If you have installed Hive-Tez, repeat the configuration steps for Hive-Tez as well.

Apache Phoenix-Hive usage examples

You can refer to the following Phoenix-Hive connector examples:

- Creating a table
- Loading data
- Querying data

Creating a table

Creating an external Hive table requires an existing table in Phoenix. Hive manages only the Hive metadata. Dropping an external table from Hive deletes only the Hive metadata, but the Phoenix table is not deleted.

Use the create external table command to create an EXTERNAL Hive table.

```
create external table ext_table (
  i1 int,
  s1 string,
  f1 float,
  d1 decimal
)
STORED BY 'org.apache.phoenix.hive.PhoenixStorageHandler'
TBLPROPERTIES (
```

```
  "phoenix.table.name" = "ext_table",
  "phoenix.zookeeper.quorum" = "localhost",
  "phoenix.zookeeper.znode.parent" = "/hbase",
  "phoenix.zookeeper.client.port" = "2181",
  "phoenix.rowkeys" = "i1",
  "phoenix.column.mapping" = "i1:i1, s1:s1, f1:f1, d1:d1"
);
```

Following are some of the recommended parameters that you could use when creating an external table:

Parameter	Default Value	Description
phoenix.table.name	The same name as the Hive table	Name of the existing Phoenix table
phoenix.zookeeper.quorum	localhost	Specifies the ZooKeeper quorum for HBase
phoenix.zookeeper.znode.parent	/hbase	Specifies the ZooKeeper parent node for HBase
phoenix.zookeeper.client.port	2181	Specifies the ZooKeeper port
phoenix.rowkeys	N/A	The list of columns to be the primary key in a Phoenix table
phoenix.column.mapping	N/A	Mappings between column names for Hive and Phoenix

Loading data

Use insert statement to load data to the Phoenix table through Hive.

```
insert into table T values (....);
insert into table T select c1,c2,c3 from source_table;
```

Querying data

You can use HiveQL for querying data in a Phoenix table.

Following are some of the recommended parameters that you could use when querying the data:

Parameter	Default Value	Description
hbase.scan.cache	100	Read row size for a unit request
hbase.scan.cacheblock	false	Whether or not cache block
split.by.stats	false	If true, mappers use table statistics. One mapper per guide post.
[hive-table-name].reducer.count	1	Number of reducers. In Tez mode, this affects only single-table queries. See Limitations.
[phoenix-table-name].query.hint	N/A	Hint for Phoenix query (for example, NO_INDEX)

Limitations of Phoenix-Hive connector

Following are some of the limitations of Phoenix-Hive connector:

- Only 4K character specification is allowed to specify a full table. If the volume of the data is huge, then there is a possibility to lose the metadata information.
- There is a difference in the way timestamp is saved in Phoenix and Hive. Phoenix uses binary format, whereas Hive uses a text format to store data.
- Hive LLAP is not supported.