Using Apache Phoenix to store and access data

Date of Publish: 2019-08-26

https://docs.hortonworks.com

Contents

What's New in Apache Phoenix

Phoenix in Hortonworks Data Platform (HDP) 3.0 includes the following new features:

- HBase 2.0 support

  This is one of the major release driver.
- Python driver for Phoenix Query Server

  This is a community driver that is brought into the Apache Phoenix project. It Provides Python db 2.0 API implementation.
- Query log

  This is a new system table "SYSTEM.LOG" that captures information about queries that are being run against the cluster (client-driven).
- Column encoding

  This is new to HDP. You can use a custom encoding scheme of data in the HBase table to reduce the amount of space taken. This increases the performance due to less data to read and thereby reduces the storage. The performance gain is 30% and above for the sparse tables.
- Hive 3.0 support for Phoenix

  It provides updated phoenix-hive StorageHandler for the new Hive version.
- Spark 2.3 support for Phoenix

  It provides updated phoenix-spark driver for new the Spark version.
- Supports GRANT and REVOKE commands

  It provides automatic changes to indexes ACLs, if access changed for data table or view.
- This version introduces support for sampling tables.
- Supports atomic update (ON DUPLICATE KEY).
- Supports snapshot scanners for MR-based queries.
- Hardening of both the secondary indexes that includes Local and Global.

Orchestrating SQL and APIs with Apache Phoenix

Apache Phoenix is a SQL abstraction layer for interacting with Apache HBase and other Hadoop components. Phoenix lets you create and interact with tables in the form of typical DDL/DML statements via its standard JDBC API. With the driver APIs, Phoenix translates SQL to native HBase API calls.

Consequently, Phoenix provides a SQL skin for working with data and objects stored in the NoSQL schema of HBase.

This Phoenix documentation focuses on interoperability with HBase. For more information about Phoenix capabilities, see the Apache Phoenix website.

Related Information

Apache Phoenix website

Enable Phoenix and interdependent components

Use Ambari to enable phoenix and its related components.

About this task

If you have a Hortonworks Data Platform installation with Ambari, then no separate installation is required for Phoenix.

To enable Phoenix with Ambari:

Procedure

1. Open Ambari.
2. Select Services tab > HBase > Configs tab.
3. Scroll down to the Phoenix SQL settings.
4. (Optional) Reset the Phoenix Query Timeout.
5. Click the Enable Phoenix slider button.

Note:

Your Phoenix installation must be the same version as the one that is packaged with the distribution of the HDP stack version that is deployed across your cluster.

Thin Client connectivity with Phoenix Query Server

The Phoenix Query Server (PQS) is a component of the Apache Phoenix distribution. PQS provides an alternative means to connect directly. PQS is a stand-alone server that converts custom API calls from "thin clients" to HTTP requests that make use of Phoenix capabilities.

This topology offloads most computation to PQS and requires a smaller client-side footprint. The PQS client protocol is based on the Avatica component of Apache Calcite.

Secure authentication on the Phoenix Query Server

About this task

You can enable Kerberos-based authentication on PQS with Ambari. If you chose to install HDP manually instead, see Configuring Phoenix Query Server to enable the Kerberos protocol.

Related Information

Configuring Phoenix Query Server

Options to obtain a client driver

You have two options to develop an application that works with Phoenix, depending on the client-server architecture. They are without PQS and with PQS.

Without Phoenix Query Server:

If your environment does not have a PQS layer, applications that connnect to Phoenix must use the Phoenix JDBC client driver.

With Phoenix Query Server:

PQS is an abstraction layer that enables other languages such as Python and GoLang to work with Phoenix. The layer provides a protocol buffer as an HTTP wrapper around Phoenix JDBC. You might prefer to use a non-Java client driver for one of various reasons, such as to avoid the JVM footprint on the client or to develop with a different application framework.

Obtaining a driver for application development

To obtain the appropriate driver for application development, visit the specified site and download the driver from appropriate file path.

JDBC Driver	Use the /usr/hdp/current/phoenix-client/phoenix-client.jar file in the Hortonworks Phoenix server-client repository. If you use the repository, download the JAR file corresponding to your installed HDP version. With Ambari, you can determine the HDP version by using the Versions tab. Alternatively, run the hadoop version command to print information displaying the HDP version.
JDBC Driver as a Maven dependency	See Download the HDP Maven Artifacts for Maven artifact repositories that are available for HDP.
Microsoft .NET Driver	Download and install a NuGet package for the Microsoft .NET Driver for Apache Phoenix and Phoenix Query Server. Note: Operability with this driver is a Hortonworks Technical Preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on the Hortonworks Support Portal.
Other non-Java drivers	Other non-JDBC Drivers for Phoenix are available as HDP add-ons and on other websites, but they are not currently supported by Hortonworks. You can find compatible client drivers by constructing a web search string consisting of "avatica" and the name of an application programming language that you want to use. Example: avatica python .

Related Information

Hortonworks Phoenix server-client repository

Versions tab

Microsoft .NET Driver for Apache Phoenix and Phoenix Query Server

Hortonworks Support Portal

Creating and using User-Defined functions (UDFs) in Phoenix

With a user-defined function (UDF), you can extend the functionality of your SQL statements by creating scalar functions that operate on a specific tenant.

For details about creating, dropping, and how to use UDFs for Phoenix, see User-defined functions on the Apache website.

Related Information

User-defined functions

Overview of mapping Phoenix schemas to HBase namespaces

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

HBase, which is often the underlying storage engine for Phoenix, has namespaces to support multitenancy features. Multitenancy helps an HBase user or administrator perform access control and quota management tasks. Also, namespaces enable tighter control of where a particular data set is stored on RegionsServers. See Enabling Multitenancy with Namepaces for further information.

Prior to HDP 2.5, Phoenix tables could not be associated with a namespace other than the default namespace.

Related Information

Enabling Multitenancy with Namepaces

Enable namespace mapping

You can enable namespace mapping by setting an appropriate property in the hbase-site.xml of both the client and the server.

About this task

Note:

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.

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

Procedure

1. Set the phoenix.schema.isNamespaceMappingEnabled property to true in the hbase-site.xml file of both the client and the server.
2. Restart the HBase Master and RegionServer processes.

Note:

You might not want to map Phoenix system tables to namespaces because there are compatibility issues with your current applications. In this case, set the phoenix.schema.mapSystemTablesToNamespace property of the hbase-site.xml file to false.

Namespace mapping properties in the hbase-site.xml file

There are two namespace properties in the hbase-site.xml file. They are phoenix.schema.isNamespaceMappingEnabled and phoenix.schema.mapSystemTablesToNamespace.

phoenix.schema.isNamespaceMappingEnabled	Enables mapping of tables of a Phoenix schema to a non-default HBase namespace. To enable mapping of schema to a non-default namespace, set the value of this property to true. Default setting for this property is false.
phoenix.schema.mapSystemTablesToNamespace	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.

Overview to managing schemas

You can use DDL statements such as CREATE SCHEMA, USE SCHEMA and DROP SCHEMA to manage schemas.

You must have admin privileges in HBase to run CREATE SCHEMA or DROP SCHEMA.

See the Apache Phoenix Grammar reference page for how you can use these DDL statements.

As you create physical tables, views, and indexes, you can associate them with a schema. If the schema already has namespace mapping enabled, the newly created objects automatically become part of the HBase namespace. The directory of the HBase namespace that maps to the Phoenix schema inherits the schema name. For example, if the schema name is store1, then the full path to the namespace is $hbase.rootdir/data/store1. See the "F.A.Q." section of Apache Phoenix Namespace Mapping for more information.

Related Information

Apache Phoenix Grammar

Apache Phoenix Namespace Mapping

Associating tables of a schema to a namespace

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.

Associating table in a noncustomized environment without Kerberos

You can run an appropriate command to associate a table in a noncustomized environment without Kerberos.

Procedure

- Run the following command to associate a table:

```
$bin/psql.py
ZooKeeper_hostname
-m
schema_name.table_name
```

Associating table in a customized Kerberos environment

You can run an appropriate command to associate a table in a customized Kerberos environment.

Before you begin

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

Procedure

1. Navigate to the Phoenix home directory. The default location is /usr/hdp/current/phoenix-client/.

2. 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:

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

Additional information for valid command parameters:

- ZooKeeper_hostnames

  Enter the ZooKeeper hostname or hostnames that compose the ZooKeeper quorum. If you enter multiple hostnames, enter them as comma-separated values. This parameter is required. You must append the colon and ZooKeeper port number if you invoke the other security parameters in the command. The default port number is 2181.
- zookeeper.znode.parent

  This setting is defined in the hbase-site.xml file.
- -m schema_name.table_name

  The -m argument is required. There is a space before and after the -m option.

Understanding Apache Phoenix-spark connector

With Hortonworks Data Platform (HDP), you can use Apache Phoenix-spark plugin on your secured clusters to perform READ and WRITE operations. You can use this tool with HDP 2.5 or later.

Connect to secured cluster

You can connect to a secured cluster using the Phoenix JDBC connector.

Procedure

Enter the following syntax in the shell:

```
jdbc:phoenix:<ZK hostnames>:<ZK port>:<root znode>:<principal name>:<keytab
 file location>
```

```
jdbc:phoenix:h1.hdp.local,h2.hdp.local,h3.hdp.local:2181:/hbase-
secure:user1@HDP.LOCAL:/Users/user1/keytabs/myuser.headless.keytab
```

You need Principal and keytab parameters only if you have not done the kinit before starting the job and want Phoenix to log you in automatically.

Considerations for setting up spark

Set up Spark based on your requirement. Following are some of the considerations that you will have to take into account.

- You should configure the 'spark.executor.extraClassPath' and ‘spark.driver.extraClassPath’ in spark-defaults.conf file to include the ‘phoenix-<version>-client.jar’ to ensure that all required Phoenix and HBase platform dependencies are available on the classpath for the Spark executors and drivers.
-

HDP Version	Spark Version	JARs to add (order dependent)
>=2.6.2 (including 3.0.0)	Spark 2	phoenix-<version>-spark2.jar phoenix-<version>-client.jar
>=2.6.2 (including 3.0.0)	Spark 1	phoenix-<version>-spark.jar phoenix-<version>-client.jar
2.6.0-2.6.1	Spark 2	Unsupported: upgrade to at least HDP-2.6.2
2.6.0-2.6.1	Spark 1	phoenix-<version>-spark.jar phoenix-<version>-client.jar
2.5.x	Spark 1	phoenix-<version>-client-spark.jar

- To enable your IDE, you can add the following provided dependency to your build:

```
<dependency><groupId>org.apache.phoenix</groupId>
<artifactId>phoenix-spark</artifactId>
<version>${phoenix.version}</version>
<scope>provided</scope></dependency>
```

Phoenix Spark connector usage examples

You can refer to the following Phoenix spark connector examples:

- Reading Phoenix tables
- Saving Phoenix tables
- Using PySpark to READ and WRITE tables

Reading Phoenix tables

For example, you have a Phoenix table with the following DDL, you can use 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 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._

val sc = new SparkContext("local", "phoenix-test")
val sqlContext = new SQLContext(sc)

val df = sqlContext.load(
```

```
  "org.apache.phoenix.spark",
  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'

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._

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(
  "TABLE1", Seq("ID", "COL1"), zkUrl = Some("phoenix-server:2181")
)

rdd.count()

val firstId = rdd1.first()("ID").asInstanceOf[Long]
val firstCol = rdd1.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._

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"),
    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.

Limitations of Apache Phoenix-spark connector

You should be aware of the following limitations on using the Apache Phoenix-Spark connector:

- You can use the DataSource API only for basic support for column and predicate pushdown.
- The DataSource API does not support passing custom Phoenix settings in configuration. You must create the DataFrame or RDD directly, if you need a fine-grained configuration.
- There is no support for aggregate or distinct queries, but you can perform any operation on RDDs or DataFrame formed after reading data from Phoenix.

Note:

The Phoenix JDBC driver normalizes column names, but the Phoenix-Spark integration does not perform this operation while loading data from Phoenix Table. so, while creating data frames or RDDs from Phoenix table(sparkContext.phoenixTableAsRDD or sqlContext.phoenixTableAsDataFrame), you must specify column names in the same way as defined when the Phoenix table was created. However, while persisting data frame in Phoenix , it can normalize the column names(which are not double quoted) by default, which can also be turned off by setting the skipNormalizingIdentifier parameter to true.

df.saveToPhoenix(<tableName>, zkUrl = Some(quorumAddress),skipNormalizingIdentifier=true)

Understanding Apache Phoenix-Hive connector

With Hortonworks Data Platform (HDP), you can use the Phoenix-Hive Storage Handler on your secured clusters to handle large joins and large aggregation. You can use this Storage Handler with HDP 2.6 or later.

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.

A change to Hive in HDP 3.0 is that all StorageHandlers must be be marked as "external". There is no such thing as an non-external table created by a StorageHandler. If the corresponding Phoenix table exists when the Hive table is created, it will mimic the HDP 2.x semantics of an "external" table. If the corresponding Phoenix table does not exist when the Hive table is created, it will mimic the HDP 2.x semantics of a non-external table (for example, the Phoenix table is dropped when the Hive table is dropped).

Considerations for setting up Hive

Make phoenix-$VERSION-hive.jar available for Hive:

- Add the phoenix-hive JAR to the HIVE_AUX_JARS_PATH variable in hive-env.sh.

```
HIVE_AUX_JARS_PATH=[..],file:///usr/hdp/current/phoenix-client/phoenix-
$VERSION-hive.jar
```

For HDP 3.0, you can add the following jar to the hive.aux.jars.path parameter as a comma-separated list. The exact file name for HDP-3.0 exists below, but please note that the version will be different for subsequent HDP 3.x.x releases.

```
file:///usr/hdp/current/phoenix-client/phoenix-5.0.0.3.0.0.0-1634-hive.jar
```

- Add a property to hive-site.xml so that Hive MapReduce jobs can also add the phoenix-hive jar to the classpath:

```
<property>
<name>hive.aux.jars.path</name>
<value>file:///usr/hdp/current/phoenix-client/phoenix-$VERSION-hive.jar</
value>
</property>
```

Apache Phoenix-Hive usage examples

You can view the examples of creating a table, loading data, and querying data using Hive. You can use the Apache Phoenix StorageHandler plugin to enable Apache Hive access to Phoenix tables from the Apache Hive command line using HiveQL.

Creating a table

In HDP 3.0, all the Hive tables that are backed by a StorageHandler must use the EXTERNAL keyword. 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 craete 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 the 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

Load 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;
```

Query data

You can use HiveQL for querying data in a Phoenix table. A Hive query on a single table can be as fast as running the query in the Phoenix CLI with the following property settings:

```
hive.fetch.task.conversion=more and hive.exec.parallel=true
```

Following are some of the 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		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 in this integration.
- As of HDP 3.0, the MapReduce engine for Hive is deprecated. Similarly, this tool is not guaranteed to work with the MapReduce engine.

Python library for Apache Phoenix

The Apache Phoenix Python driver is a new addition to the Apache Phoenix. It was originally known as "Python Phoenixdb".

For more information, see the Apache Phoenix site.

The Python driver provides the Python DB2.0 API , which is a generic interface for interacting with databases through Python. This driver requires Phoenix Query Server (PQS) to interact with Phoenix. Using this driver, you can execute queries and load data. All data types are expected to be functional and there are no limitations on the kind of queries that this driver can execute.

Note:

This driver does not support Kerberos authentication.

Example of Phoenix Python library

Following code is an example of Phoenix Python library.

```
db = phoenixdb.connect('http://localhost:8765', autocommit=True)
with db.cursor() as cursor:
cursor.execute("DROP TABLE IF EXISTS test")
cursor.execute("CREATE TABLE test (id INTEGER PRIMARY KEY, text VARCHAR)")
cursor.executemany("UPSERT INTO test VALUES (?, ?)", [[i, 'text
 {}'.format(i)] for i in range(10)])
with db.cursor() as cursor:
cursor.itersize = 4
cursor.execute("SELECT * FROM test WHERE id>1 ORDER BY id")
self.assertEqual(cursor.fetchall(), [[i, 'text {}'.format(i)] for i in
 range(2, 10)])
db.close()
```

Using index in Phoenix

Apache Phoenix automatically uses indexes to service a query.

Phoenix supports global and local indexes. Each is useful in specifc scenarios and has its own performance characteristics.

Global indexes in Phoenix

You can use global indexes for READ-heavy use cases. Each global index is stored in its own table, and thus is not co-located with the data table.

With global indexes, you can disperse the READ load between the main and secondary index table on different RegionServers serving different sets of access patterns. A Global index is a covered index. It is used for queries only when all columns in that query are included in that index.

Local indexes in Phoenix

You can use local indexes for WRITE-heavy use cases. Each local indexes is stored within the data table.

With global indexes, you can use local indexes even when all columns referenced in a query are not contained in the index. This is done by default for local indexes, because the table and index data resides on the same region server and hence it ensures that the lookup is local.

Using Phoenix client to load data

You must use Phoenix client to load data into the HBase database and also to write to the Phoenix tables.

Index updates are automatically generated by the Phoenix client and there is no user intervention or effort required. Whenever a record is written to the Phoenix tables, the client generates updates for the indexes automatically.

Note:

If Phoenix table has indexes, you can use JDBC driver or CSV bulk load table to update or ingest data.

It is highly recommended that you use Phoenix client to load data into the HBase database and also to write to the Phoenix tables. If HBase APIs are used to write data to a Phoenix data table, indexes against that Phoenix data table will not be updated.

Phoenix repair tool

Apache Phoenix depends on the SYSTEM.CATALOG table for metadata information, such as table structure and index location, to function correctly. Use the Phoenix repair tool to validate the data integrity of the SYSTEM.CATALOG table.

Note:

The Phoenix repair tool of HDP is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on the Hortonworks Support Portal.

If a Phoenix client is not functioning as expected and throwing exceptions such as ArrayIndexOutOfBound or TableNotFound, this tool can help identify the problem and fix it.

The repair tool is designed to flag issues that are flagrant trouble spots and to fix SYSTEM.CATALOG problems in a way that does not radically affect your Phoenix system. The tool prompts you to confirm changes before the SYSTEM.CATALOG table is modified.

Do not use the Phoenix repair tool for an upgrade. The tool is designed to function ony with the current version of the system catalog and to use the HBase API directly.

Related Information

Hortonworks Support Portal

Run the Phoenix repair tool

You can run the Phoenix repair tool to gather information about the components of the table. The tool provide capabilities to fix certain problems.

About this task

Note:

Run the HDFS fsck and HBase hbck tools before running the Phoenix repair tool. Checking the condition of HDFS and HBase is highly recommended because the Phoenix repair tool does not run on HDFS and HBase, both of which must be in working order for the repair tool to fix Phoenix problems.

- The Phoenix repair tool looks for table records in the system catalog and collects all corresponding information about columns and indexes. If certain inconsistencies are detected, then the tool prompts you to verify that it should proceed with fixing the problems. The tool can fix the following problems:
- Missing or disabled physical table
- Incorrect number of columns in table metadata information
- Table record has columns with an internal index that is out of range
- The tool performs a cross-reference check between user tables and indexes. If a user table has an index that misses a physical table, the tool offers to delete the link to this index as well as to delete the index

table record from the system catalog. If the physical table is disabled, the tool asks whether it needs to be enabled.

- If you allow the Phoenix repair tool to fix an issue, the tool creates a snapshot of the SYSTEM.CATALOG table. The snapshot is created in case you want to rollback the repair operation.

Prerequisites

Verify that no concurrent execution of the Phoenix repair tool launches or runs while you run the tool. Also, ensure that no other clients modify the system catalog data while the tool runs.

Procedure

1. Run the psl.py utility with the -r option:

```
/usr/hdp/current/phoenix-client/psql.py -r
```

2. If the tool detects previously stored snapshots on the system, respond to the Restore dialogue prompt:
   - Respond whether the tool should delete or retain the previously recorded snapshots.
   - Indicate whether the tool should proceed with the integrity check or restore a table from the one of the snapshots.

Results

After the tool completes the check, you can consider the SYSTEM.CATALOG table as validated. You can proceed with SQL operations in the Phoenix CLI.