Cloudera Runtime 7.2.11

Apache Phoenix

Date published: 2020-02-29
Date modified: 2021-09-09

https://docs.cloudera.com/

Legal Notice

Contents

Introduction to Apache Phoenix

Apache Phoenix is is a SQL layer for Apache HBase that provides a programmatic ANSI SQL interface.

Apache Phoenix implements best-practice optimizations to enable software engineers to develop HBase based next-generation applications that operationalize big data. Using Phoenix, you can create and interact with tables in the form of typical DDL/DML statements using the Phoenix standard JDBC API.

Apache HBase timestamps are the core part of the Apache Phoenix functionality. It is used as part of the *Travel in time* functionality and by the Transactional engines. Due to this, we expect that the data must have valid timestamps that signifies the time when the record is created. Invalid timestamps cause incorrect behavior for most Apache Phoenix functionalities, such as secondary indexes or CRUD operations. If your data is relying on the special behavior of the record's timestamps or future timestamps, you must not use such data with Apache Phoenix.

Apache Phoenix and SQL

You can use the Apache Phoenix SQL commands to create, drop, or modify an Apache HBase table. You can also create Apache Phoenix views that are virtual tables that share the same Apache HBase table.

The Apache Phoenix commands enable you to use standard SQL data definition language (DDL) and data manipulation language (DML) commands to interact with Apache HBase tables. You can create a new Apache HBase table using the standard CREATE TABLE SQL command or create a view on an existing Apache HBase table using the VIEW command. View enables you to have multiple virtual tables that share the same physical Apache HBase table.

Apache HBase tables and Apache Phoenix tables have a one-to-one relationship. This means each Apache HBase table is associated with a corresponding Apache Phoenix table.

Warning: Although you can modify Apache Phoenix tables using the Apache HBase native APIs, this is not supported and results in errors, inconsistent indexes, incorrect query results, and sometimes corrupt data.

Using Apache Phoenix commands you can do the following tasks:

- Create or alter Apache HBase tables using DDL commands like CREATE TABLE
- Modify contents in an Apache HBase table using DML commands like UPSERT VALUES

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);`

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 (UPPE R(FIRST_NAME||' '||LAST_NAME));`
	Search on the combined first name and last name using the following command:
	`SELECT EMP_ID FROM EMP WHERE UPPER(F IRST_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);`