APACHE PHOENIX

OLTP and operational analytics for Apache Hadoop

(download.html)
Download latest Apache Phoenix binary and source release artifacts

(issues.html)
Browse through Apache Phoenix JIRAs

(source.html)
Sync and build Apache Phoenix from source code

News (news.html): **Phoenix 5.1.3** has been released and is available for download **here** (download.html)
Follow @ApachePhoenix (https://twitter.com/ApachePhoenix)

Overview

Apache Phoenix enables OLTP and operational analytics in Hadoop for low latency applications by combining the best of both worlds:

- the power of standard SQL and JDBC APIs with full ACID transaction capabilities and
- the flexibility of late-bound, schema-on-read capabilities from the NoSQL world by leveraging HBase as its backing store

Apache Phoenix is fully integrated with other Hadoop products such as Spark, Hive, Pig, Flume, and Map Reduce.

Who is using Apache Phoenix? Read more here... (who_is_using.html)

Mission

Become the trusted data platform for OLTP and operational analytics for Hadoop through well-defined, industry standard APIs.

Quick Start

Tired of reading already and just want to get started? Take a look at our FAQs (faq.html), listen to the Apache Phoenix talk from Hadoop Summit 2015 (https://www.youtube.com/watch?v=XGa0SyJMH94), review the overview presentation (/presentations/OC-HUG-2014-10-4x3.pdf), and jump over to our quick start guide here (Phoenix-in-15-minutes-or-less.html).

SQL Support

Apache Phoenix takes your SQL query, compiles it into a series of HBase scans, and orchestrates the running of those scans to produce regular JDBC result sets. Direct use of the HBase API, along with coprocessors and custom filters, results in performance (performance.html) on the order of milliseconds for small queries, or seconds for tens of millions of rows.

To see a complete list of what is supported, go to our language reference (language/index.html). All standard SQL query constructs are supported, including `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, etc. It also supports a full set of DML commands as well as table creation and versioned incremental alterations through our DDL commands.

Here's a list of what is currently **not** supported:

- **Relational operators**. Intersect, Minus.
- **Miscellaneous built-in functions**. These are easy to add - read this blog (http://phoenix-hbase.blogspot.com/2013/04/how-to-add-your-own-built-in-function.html) for step by step instructions.

Connection

Use JDBC to get a connection to an HBase cluster like this:

```
Connection conn = DriverManager.getConnection("jdbc:phoenix:server1,server2:3333", props);
```

where `props` are optional properties which may include Phoenix and HBase configuration properties, and the connection string which is composed of:

```
jdbc:phoenix [ :<zookeeper quorum> [ :<port number> [ :<root node> [ :<principal> [ :<keytab file> ] ] ] ] ]
```

For any omitted parts, the relevant property value, hbase.zookeeper.quorum, hbase.zookeeper.property.clientPort, and zookeeper.znode.parent will be used from hbase-site.xml configuration file. The optional `principal` and `keytab file` may be used to connect to a Kerberos secured cluster. If only `principal` is specified, then this defines the user name with each distinct user having their own dedicated HBase connection (HConnection). This provides a means of having multiple, different connections each with different configuration properties on the same JVM.

For example, the following connection string might be used for longer running queries, where the `longRunningProps` specifies Phoenix and HBase configuration properties with longer timeouts:

```
Connection conn = DriverManager.getConnection(“jdbc:phoenix:my_server:longRunning”, longRunningProps);
```

while the following connection string might be used for shorter running queries:

```
Connection conn = DriverManager.getConnection("jdbc:phoenix:my_server:shortRunning", shortRunningProps);
```

Please read the relevant FAQ entry (faq.html#What_is_the_Phoenix_JDBC_URL_syntax) for example URLs.

Phoenix now also supports Connecting to HBase without Zookeeper (classpath_and_url.html#The_Phoenix_JDBC_URL)

Transactions

To enable full ACID transactions, a beta feature available in the 4.7.0 release, set the `phoenix.transactions.enabled` property to true. In this case, you'll also need to run the transaction manager that's included in the distribution. Once enabled, a table may optionally be declared as transactional (see here (transactions.html) for directions). Commits over transactional tables will have an all-or-none behavior - either all data will be committed (including any updates to secondary indexes) or none of it will (and an exception will be thrown). Both cross table and cross row transactions are supported. In addition, transactional tables will see their own uncommitted data when querying. An optimistic concurrency model is used to detect row level conflicts with first commit wins semantics. The later commit would produce an exception indicating that a conflict was detected. A transaction is started implicitly when a transactional table is referenced in a statement, at which point you will not see updates from other connections until either a commit or rollback occurs.

Non transactional tables have no guarantees above and beyond the HBase guarantee of row level atomicity (see here (https://hbase.apache.org/acid-semantics.html)). In addition, non transactional tables will not see their updates until after a commit has occurred. The DML commands of Apache Phoenix, UPSERT VALUES, UPSERT SELECT and DELETE, batch pending changes to HBase tables on the client side. The changes are sent to the server when the transaction is committed and discarded when the transaction is rolled back. If auto commit is turned on for a connection, then Phoenix will, whenever possible, execute the entire DML command through a coprocessor on the server-side, so performance will improve.

Timestamps

Most commonly, an application will let HBase manage timestamps. However, under some circumstances, an application needs to control the timestamps itself. In this case, the CurrentSCN (faq.html#Can_phoenix_work_on_tables_with_arbitrary_timestamp_as_flexible_as_HBase_API) property may be specified at connection time to control timestamps for any DDL, DML, or query. This capability may be used to run snapshot queries against prior row values, since Phoenix uses the value of this connection property as the max timestamp of scans.

Timestamps may not be controlled for transactional tables. Instead, the transaction manager assigns timestamps which become the HBase cell timestamps after a commit. Timestamps still correspond to wall clock time, however they are multiplied by 1,000,000 to ensure enough granularity for uniqueness across the cluster.

Schema

Apache Phoenix supports table creation and versioned incremental alterations through DDL commands. The table metadata is stored in an HBase table and versioned, such that snapshot queries over prior versions will automatically use the correct schema.

A Phoenix table is created through the CREATE TABLE (language/index.html#create) command and can either be:

1. **built from scratch**, in which case the HBase table and column families will be created automatically.
2. **mapped to an existing HBase table**, by creating either a read-write TABLE or a read-only VIEW, with the caveat that the binary representation of the row key and key values must match that of the Phoenix data types (see Data Types reference (language/datatypes.html) for the detail on the binary representation).

- For a read-write TABLE, column families will be created automatically if they don't already exist. An empty key value will be added to the first column family of each existing row to minimize the size of the projection for queries.
- For a read-only VIEW, all column families must already exist. The only change made to the HBase table will be the addition of the Phoenix coprocessors used for query processing. The primary use case for a VIEW is to transfer existing data into a Phoenix table, since data modification are not allowed on a VIEW and query performance will likely be less than as with a TABLE.

All schema is versioned (with up to 1000 versions being kept). Snapshot queries over older data will pick up and use the correct schema based on the time at which you've connected (based on the CurrentSCN (faq.html#Can_phoenix_work_on_tables_with_arbitrary_timestamp_as_flexible_as_HBase_API) property).

Altering

A Phoenix table may be altered through the ALTER TABLE (language/index.html#alter) command. When a SQL statement is run which references a table, Phoenix will by default check with the server to ensure it has the most up to date table metadata and statistics. This RPC may not be necessary when you know in advance that the structure of a table may never change. The UPDATE_CACHE_FREQUENCY property was added in Phoenix 4.7 to allow the user to declare how often the server will be checked for meta data updates (for example, the addition or removal of a table column or the updates of table statistics). Possible values are ALWAYS (the default), NEVER, and a millisecond numeric value. An ALWAYS value will cause the client to check with the server each time a statement is executed that references a table (or once per commit for an UPSERT VALUES statement). A millisecond value indicates how long the client will hold on to its cached version of the metadata before checking back with the server for updates.

For example, the following DDL command would create table `FOO` and declare that a client should only check for updates to the table or its statistics every 15 minutes:

```
CREATE TABLE FOO (k BIGINT PRIMARY KEY, v VARCHAR)
UPDATE_CACHE_FREQUENCY=900000;
```

Views

Phoenix supports updatable views on top of tables with the unique feature leveraging the schemaless capabilities of HBase of being able to add columns to them. All views all share the same underlying physical HBase table and may even be indexed independently. For more read here (views.html).

Multi-tenancy

Built on top of view support, Phoenix also supports multi-tenancy (multi-tenancy.html). Just as with views, a multi-tenant view may add columns which are defined solely for that user.

Schema at Read-time

Another schema-related feature allows columns to be defined dynamically at query time. This is useful in situations where you don't know in advance all of the columns at create time. You'll find more details on this feature here (dynamic_columns.html).

Mapping to an Existing HBase Table

Apache Phoenix supports mapping to an existing HBase table through the CREATE TABLE (language/index.html#create) and CREATE VIEW (language/index.html#create) DDL statements. In both cases, the HBase metadata is left as-is, except for with CREATE TABLE the KEEP_DELETED_CELLS (http://hbase.apache.org/book/cf.keep.deleted.html) option is enabled to allow for flashback queries to work correctly. For CREATE TABLE, any HBase metadata (table, column families) that doesn't already exist will be created. Note that the table and column family names are case sensitive, with Phoenix upper-casing all names. To make a name case sensitive in the DDL statement, surround it with double quotes as shown below:

```
CREATE VIEW “MyTable” (“a”.ID VARCHAR PRIMARY KEY)
```

For CREATE TABLE, an empty key value will also be added for each row so that queries behave as expected (without requiring all columns to be projected during scans). For CREATE VIEW, this will not be done, nor will any HBase metadata be created. Instead the existing HBase metadata must match the metadata specified in the DDL statement or a `ERROR 505 (42000): Table is read only` will be thrown.

The other caveat is that the way the bytes were serialized in HBase must match the way the bytes are expected to be serialized by Phoenix. For VARCHAR,CHAR, and UNSIGNED_* types, Phoenix uses the HBase Bytes utility methods to perform serialization. The CHAR type expects only single-byte characters and the UNSIGNED types expect values greater than or equal to zero.

Our composite row keys are formed by simply concatenating the values together, with a zero byte character used as a separator after a variable length type. For more information on our type system, see the Data Type (language/datatypes.html).

Salting

A table could also be declared as salted to prevent HBase region hot spotting. You just need to declare how many salt buckets your table has, and Phoenix will transparently manage the salting for you. You'll find more detail on this feature here (salted.html), along with a nice comparison on write throughput between salted and unsalted tables here (performance.html#salting).

APIs

The catalog of tables, their columns, primary keys, and types may be retrieved via the java.sql metadata interfaces: `DatabaseMetaData`, `ParameterMetaData`, and `ResultSetMetaData`. For retrieving schemas, tables, and columns through the DatabaseMetaData interface, the schema pattern, table pattern, and column pattern are specified as in a LIKE expression (i.e. % and _ are wildcards escaped through the character). The table catalog argument in the metadata APIs is used to filter based on the tenant ID for multi-tenant tables.