Apache Phoenix使用文档(英文)

书栈(BookStack.CN)

目　录

致谢

F.A.Q.

Quick Start

Building

Tuning

Explain Plan

Configuration

Backward Compatibility

Release Notes

Performance Testing

Apache Spark Integration

Phoenix Storage Handler for Apache Hive

Apache Pig Integration

Map Reduce Integration

Apache Flume Plugin

Apache Kafka Plugin

Python Driver

致谢

F.A.Q.

F.A.Q.

I want to get started. Is there a Phoenix Hello World?

Pre-requisite: Download latest Phoenix from here and copy phoenix-*.jar to HBase lib folder and restart HBase.

1. Using console

- Start Sqlline: $ sqlline.py [zookeeper]
- Execute the following statements when Sqlline connects:

```
create table test (mykey integer not null primary key, mycolumn varchar);
upsert into test values (1,'Hello');
upsert into test values (2,'World!');
select * from test;
```

- You should get the following output

```
+-------+------------+
| MYKEY |  MYCOLUMN  |
+-------+------------+
| 1     | Hello      |
| 2     | World!     |
+-------+------------+
```

2. Using java

Create test.java file with the following content:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Statement;

public class test {

    public static void main(String[] args) throws SQLException {
        Statement stmt = null;
        ResultSet rset = null;

        Connection con = DriverManager.getConnection("jdbc:phoenix:[zookeeper]");
        stmt = con.createStatement();

        stmt.executeUpdate("create table test (mykey integer not null primary key, mycolumn varchar)");
        stmt.executeUpdate("upsert into test values (1,'Hello')");
        stmt.executeUpdate("upsert into test values (2,'World!')");
        con.commit();

        PreparedStatement statement = con.prepareStatement("select * from test");
        rset = statement.executeQuery();
        while (rset.next()) {
            System.out.println(rset.getString("mycolumn"));
        }
        statement.close();
        con.close();
    }
}
```

Compile and execute on command line

$ javac test.java $ java -cp "../phoenix-[version]-client.jar:." test

You should get the following output

HelloWorld!

What is the Phoenix JDBC URL syntax?

Thick Driver

The Phoenix (Thick) Driver JDBC URL syntax is as follows (where elements in square brackets are optional):

jdbc:phoenix:[comma-separated ZooKeeper Quorum [:port [:hbase root znode [:kerberos_principal [:path to kerberos keytab]]]]

The simplest URL is:

jdbc:phoenix:localhost

Whereas the most complicated URL is:

jdbc:phoenix:zookeeper1.domain,zookeeper2.domain,zookeeper3.domain:2181:/hbase-1:phoenix@EXAMPLE.COM:/etc/security/keytabs/phoenix.keytab

Please note that each optional element in the URL requires all previous optional elements. For example, to specify the HBase root ZNode, the ZooKeeper port *must* also be specified.

This information is initially covered on the index page.

Thin Driver

The Phoenix Thin Driver (used with the Phoenix Query Server) JDBC URL syntax is as follows:

jdbc:phoenix:thin:[key=value[;key=value…]]

There are a number of keys exposed for client-use. The most commonly-used keys are: url and serialization. The url key is required to interact with the Phoenix Query Server.

The simplest URL is:

jdbc:phoenix:thin:url=http://localhost:8765

Where as very complicated URL is:

jdbc:phoenix:thin:url=http://queryserver.domain:8765;serialization=PROTOBUF;authentication=SPENGO;principal=phoenix@EXAMPLE.COM;keytab=/etc/security/keytabs/phoenix.keytab

Please refer to the Apache Avatica documentation for a full list of supported options in the Thin client JDBC URL, or see the Query Server documentation

Is there a way to bulk load in Phoenix?

Map Reduce

See the example here

CSV

CSV data can be bulk loaded with built in utility named psql. Typical upsert rates are 20K - 50K rows per second (depends on how wide are the rows).

Usage example:Create table using psql $ psql.py [zookeeper] ../examples/web_stat.sql

Upsert CSV bulk data $ psql.py [zookeeper] ../examples/web_stat.csv

How I map Phoenix table to an existing HBase table?

You can create both a Phoenix table or view through the CREATE TABLE/CREATE VIEW DDL statement on a pre-existing HBase table. In both cases, we'll leave the HBase metadata as-is. For CREATE TABLE, we'll create any metadata (table, column families) that doesn't already exist. We'll also add an empty key value for each row so that queries behave as expected (without requiring all columns to be projected during scans).

The other caveat is that the way the bytes were serialized must match the way the bytes are serialized by Phoenix. For VARCHAR,CHAR, and UNSIGNED_* types, we use the HBase Bytes methods. The CHAR type expects only single-byte characters and the UNSIGNED types expect values greater than or equal to zero. For signed types(TINYINT, SMALLINT, INTEGER and BIGINT), Phoenix will flip the first bit so that negative values will sort before positive values. Because HBase sorts row keys in lexicographical order and negative value’s first bit is 1 while positive 0 so that negative value is ‘greater than’ positive value if we don’t flip the first bit. So if you stored integers by HBase native API and want to access them by Phoenix, make sure that all your data types are UNSIGNED types.

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.

If you create an HBase table like this:

```
create 't1', {NAME => 'f1', VERSIONS => 5}
```

then you have an HBase table with a name of ‘t1’ and a column family with a name of ‘f1’. Remember, in HBase, you don’t model the possible KeyValues or the structure of the row key. This is the information you specify in Phoenix above and beyond the table and column family.

So in Phoenix, you’d create a view like this:

```
CREATE VIEW "t1" ( pk VARCHAR PRIMARY KEY, "f1".val VARCHAR )
```

The “pk” column declares that your row key is a VARCHAR (i.e. a string) while the “f1”.val column declares that your HBase table will contain KeyValues with a column family and column qualifier of “f1”:VAL and that their value will be a VARCHAR.

Note that you don’t need the double quotes if you create your HBase table with all caps names (since this is how Phoenix normalizes strings, by upper casing them). For example, with:

```
create 'T1', {NAME => 'F1', VERSIONS => 5}
```

you could create this Phoenix view:

```
CREATE VIEW t1 ( pk VARCHAR PRIMARY KEY, f1.val VARCHAR )
```

Or if you’re creating new HBase tables, just let Phoenix do everything for you like this (No need to use the HBase shell at all.):

```
CREATE TABLE t1 ( pk VARCHAR PRIMARY KEY, val VARCHAR )
```

Are there any tips for optimizing Phoenix?

- Use **Salting** to increase read/write performance Salting can significantly increase read/write performance by pre-splitting the data into multiple regions. Although Salting will yield better performance in most scenarios.
  Example:

```
CREATE TABLE TEST (HOST VARCHAR NOT NULL PRIMARY KEY, DESCRIPTION VARCHAR)
SALT_BUCKETS=16
```

Note: Ideally for a 16 region server cluster with quad-core CPUs, choose salt buckets between 32-64 for optimal performance.

- **Per-split** table Salting does automatic table splitting but in case you want to exactly control where table split occurs with out adding extra byte or change row key order then you can pre-split a table.
  Example:

```
CREATE TABLE TEST (HOST VARCHAR NOT NULL PRIMARY KEY, DESCRIPTION VARCHAR)
SPLIT ON ('CS','EU','NA')
```

- Use **multiple column families**
  Column family contains related data in separate files. If you query use selected columns then it make sense to group those columns together in a column family to improve read performance.

Example:

Following create table DDL will create two column faimiles A and B.

```
CREATE TABLE TEST (MYKEY VARCHAR NOT NULL PRIMARY KEY, A.COL1 VARCHAR,
A.COL2 VARCHAR, B.COL3 VARCHAR)
```

- Use **compression** On disk compression improves performance on large tables
  Example:

```
CREATE TABLE TEST (HOST VARCHAR NOT NULL PRIMARY KEY, DESCRIPTION VARCHAR)
COMPRESSION='GZ'
```

- Create **indexes** See faq.html#/How_do_I_create_Secondary_Index_on_a_table

- **Optimize cluster** parameters See http://hbase.apache.org/book/performance.html

- **Optimize Phoenix** parameters See tuning.html

How do I create Secondary Index on a table?

Starting with Phoenix version 2.1, Phoenix supports index over mutable and immutable data. Note that Phoenix 2.0.x only supports Index over immutable data. Index write performance index with immutable table is slightly faster than mutable table however data in immutable table cannot be updated.

Example

- Create table
  Immutable table: `create table test (mykey varchar primary key, col1 varchar, col2 varchar) IMMUTABLE_ROWS=true;`

Mutable table: `create table test (mykey varchar primary key, col1 varchar, col2 varchar);`

- Creating index on col2

  `create index idx on test (col2)`

- Creating index on col1 and a covered index on col2

  `create index idx on test (col1) include (col2)`

Upsert rows in this test table and Phoenix query optimizer will choose correct index to use. You can see in explain plan if Phoenix is using the index table. You can also give a hint in Phoenix query to use a specific index.

Why isn’t my secondary index being used?

The secondary index won’t be used unless all columns used in the query are in it (as indexed or covered columns). All columns making up the primary key of the data table will automatically be included in the index.

Example: DDL `create table usertable (id varchar primary key, firstname varchar, lastname varchar); create index idx_name on usertable (firstname);`

Query: DDL `select id, firstname, lastname from usertable where firstname = 'foo';`

Index would not be used in this case as lastname is not part of indexed or covered column. This can be verified by looking at the explain plan. To fix this create index that has either lastname part of index or covered column. Example: `create idx_name on usertable (firstname) include (lastname);`

How fast is Phoenix? Why is it so fast?

Phoenix is fast. Full table scan of 100M rows usually completes in 20 seconds (narrow table on a medium sized cluster). This time come down to few milliseconds if query contains filter on key columns. For filters on non-key columns or non-leading key columns, you can add index on these columns which leads to performance equivalent to filtering on key column by making copy of table with indexed column(s) part of key.

Why is Phoenix fast even when doing full scan:

- Phoenix chunks up your query using the region boundaries and runs them in parallel on the client using a configurable number of threads
- The aggregation will be done in a coprocessor on the server-side, collapsing the amount of data that gets returned back to the client rather than returning it all.

How do I connect to secure HBase cluster?

Check out excellent post by Anil Gupta
http://bigdatanoob.blogspot.com/2013/09/connect-phoenix-to-secure-hbase-cluster.html

How do I connect with HBase running on Hadoop-2?

Hadoop-2 profile exists in Phoenix pom.xml.

Can phoenix work on tables with arbitrary timestamp as flexible as HBase API?

By default, Phoenix let's HBase manage the timestamps and just shows you the latest values for everything. However, Phoenix also allows arbitrary timestamps to be supplied by the user. To do that you'd specify a "CurrentSCN" at connection time, like this:

```
Properties props = new Properties();
props.setProperty("CurrentSCN", Long.toString(ts));
Connection conn = DriverManager.connect(myUrl, props);

conn.createStatement().execute("UPSERT INTO myTable VALUES ('a')");
conn.commit();
```

The above is equivalent to doing this with the HBase API:

```
myTable.put(Bytes.toBytes('a'),ts);
```

By specifying a CurrentSCN, you're telling Phoenix that you want everything for that connection to be done at that timestamp. Note that this applies to queries done on the connection as well - for example, a query over myTable above would not see the data it just upserted, since it only sees data that was created before its CurrentSCN property. This provides a way of doing snapshot, flashback, or point-in-time queries.

Keep in mind that creating a new connection is *not* an expensive operation. The same underlying HConnection is used for all connections to the same cluster, so it's more or less like instantiating a few objects.

Why isn't my query doing a RANGE SCAN?

DDL: CREATE TABLE TEST (pk1 char(1) not null, pk2 char(1) not null, pk3 char(1) not null, non-pk varchar CONSTRAINT PK PRIMARY KEY(pk1, pk2, pk3));
RANGE SCAN means that only a subset of the rows in your table will be scanned over. This occurs if you use one or more leading columns from your primary key constraint. Query that is not filtering on leading PK columns ex. select *from test where pk2='x' and pk3='y'; will result in full scan whereas the following query will result in range scan* select from test where pk1='x' and pk2='y';. Note that you

can add a secondary index on your “pk2” and “pk3” columns and that would cause a range scan to be done for the first query (over the index table).

DEGENERATE SCAN means that a query can’t possibly return any rows. If we can determine that at compile time, then we don’t bother to even run the scan.

FULL SCAN means that all rows of the table will be scanned over (potentially with a filter applied if you have a WHERE clause)

SKIP SCAN means that either a subset or all rows in your table will be scanned over, however it will skip large groups of rows depending on the conditions in your filter. See this blog for more detail. We don’t do a SKIP SCAN if you have no filter on the leading primary key columns, but you can force a SKIP SCAN by using the /+ *SKIP_SCAN* / hint. Under some conditions, namely when the cardinality of your leading primary key columns is low, it will be more efficient than a FULL SCAN.

Should I pool Phoenix JDBC Connections?

No, it is not necessary to pool Phoenix JDBC Connections.

Phoenix’s Connection objects are different from most other JDBC Connections due to the underlying HBase connection. The Phoenix Connection object is designed to be a thin object that is inexpensive to create. If Phoenix Connections are reused, it is possible that the underlying HBase connection is not always left in a healthy state by the previous user. It is better to create new Phoenix Connections to ensure that you avoid any potential issues.

Implementing pooling for Phoenix could be done simply by creating a delegate Connection that instantiates a new Phoenix connection when retrieved from the pool and then closes the connection when returning it to the pool (see PHOENIX-2388).

Why does Phoenix add an empty/dummy KeyValue when doing an upsert?

The empty or dummy KeyValue (with a column qualifier of _0) is needed to ensure that a given column is available for all rows.

As you may know, data is stored in HBase as KeyValues, meaning that the full row key is stored for each column value. This also implies that the row key is not stored at all unless there is at least one column stored.

Now consider JDBC row which has an integer primary key, and several columns which are all null. In order to be able to store the primary key, a KeyValue needs to be stored to show that the row is present at all. This column is represented by the empty column that you’ve noticed. This allows doing a “SELECT * FROM TABLE” and receiving records for all rows, even those whose non-pk columns are null.

The same issue comes up even if only one column is null for some (or all) records. A scan over Phoenix will include the empty column to ensure that rows that only consist of the primary key (and have null for all non-key columns) will be included in a scan result.

> 原文: *http://phoenix.apache.org/faq.html*

Quick Start

Phoenix in 15 minutes or less

What is this new Phoenix thing I've been hearing about? Phoenix is an open source SQL skin for HBase. You use the standard JDBC APIs instead of the regular HBase client APIs to create tables, insert data, and query your HBase data.

Doesn't putting an extra layer between my application and HBase just slow things down? Actually, no. Phoenix achieves as good or likely better performance than if you hand-coded it yourself (not to mention with a heck of a lot less code) by:

- compiling your SQL queries to native HBase scans
- determining the optimal start and stop for your scan key
- orchestrating the parallel execution of your scans
- bringing the computation to the data by
- pushing the predicates in your where clause to a server-side filter

- executing aggregate queries through server-side hooks (called co-processors)
  In addition to these items, we've got some interesting enhancements in the works to further optimize performance:

- secondary indexes to improve performance for queries on non row key columns

- stats gathering to improve parallelization and guide choices between optimizations
- skip scan filter to optimize IN, LIKE, and OR queries

- optional salting of row keys to evenly distribute write load
  Ok, so it's fast. But why SQL? It's so 1970s Well, that's kind of the point: give folks something with which they're already familiar. What better way to spur the adoption of HBase? On top of that, using JDBC and SQL:

- Reduces the amount of code users need to write

- Allows for performance optimizations transparent to the user
- Opens the door for leveraging and integrating lots of existing tooling
  But how can SQL support my favorite HBase technique of x,y,z Didn't make it to the last HBase Meetup did you? SQL is just a way of expressing ***what you want to get*** not ***how you want to get it***. Check out my presentation for various existing and to-be-done Phoenix features to support your favorite HBase trick. Have ideas of your own? We'd love to hear about them: file an issue for us and/or join our mailing list.

Blah, blah, blah - I just want to get started! Ok, great! Just follow our install instructions:

- download and expand our installation tar
- copy the phoenix server jar that is compatible with your HBase installation into the lib directory of every region server
- restart the region servers
- add the phoenix client jar to the classpath of your HBase client

- download and setup SQuirrel as your SQL client so you can issue adhoc SQL against your HBase cluster
  I don't want to download and setup anything else! Ok, fair enough - you can create your own SQL scripts and execute them using our command line tool instead. Let's walk through an example now. Begin by navigating to the `bin/` directory of your Phoenix install location.

- First, let's create a us_population.sql file, containing a table definition:

```
CREATE TABLE IF NOT EXISTS us_population (
      state CHAR(2) NOT NULL,
      city VARCHAR NOT NULL,
      population BIGINT
      CONSTRAINT my_pk PRIMARY KEY (state, city));
```

- Now let's create a us_population.csv file containing some data to put in that table:

```
NY,New York,8143197
CA,Los Angeles,3844829
IL,Chicago,2842518
TX,Houston,2016582
PA,Philadelphia,1463281
AZ,Phoenix,1461575
TX,San Antonio,1256509
CA,San Diego,1255540
TX,Dallas,1213825
CA,San Jose,912332
```

- And finally, let's create a us_population_queries.sql file containing a query we'd like to run on that data.

```
SELECT state as "State",count(city) as "City Count",sum(population) as "Population Sum"
FROM us_population
GROUP BY state
ORDER BY sum(population) DESC;
```

- Execute the following command from a command terminal

```
./psql.py <your_zookeeper_quorum> us_population.sql us_population.csv us_population_queries.sql
```

Congratulations! You've just created your first Phoenix table, inserted data into it, and executed an aggregate query with just a few lines of code in 15 minutes or less!

Big deal - 10 rows! What else you got? Ok, ok - tough crowd. Check out our `bin/performance.py` script to create as many rows as you want, for any schema you come up with, and run timed queries against it.

Why is it called Phoenix anyway? Did some other project crash and burn and this is the next generation? I'm sorry, but we're out of time and space, so we'll have to answer that next time!

> 原文: *http://phoenix.apache.org/Phoenix-in-15-minutes-or-less.html*

Building

Building Phoenix Project

Phoenix is a fully mavenized project. Download source and build simply by doing:

```
$ mvn package
```

builds, runs fast unit tests and package Phoenix and put the resulting jars (phoenix-[version].jar and phoenix-[version]-client.jar) in the generated phoenix-core/target/ and phoenix-assembly/target/ directories respectively.

To build, but skip running the fast unit tests, you can do:

```
$ mvn package -DskipTests
```

To build against hadoop2, you can do:

```
$ mvn package -DskipTests -Dhadoop.profile=2
```

To run all tests including long running integration tests

```
$ mvn install
```

To only build the generated parser (i.e. PhoenixSQLLexer and PhoenixSQLParser), you can do:

```
$ mvn install -DskipTests
$ mvn process-sources
```

To build an Eclipse project, install the m2e plugin and do an File->Import…->Import Existing Maven Projects selecting the root directory of Phoenix.

Maven

Phoenix is also hosted at Apache Maven Repository. You can add it to your mavenized project by adding the following to your pom:

```
<repositories>
  ...
   <repository>
     <id>apache release</id>
     <url>https://repository.apache.org/content/repositories/releases/</url>
```

```
    </repository>
    ...
  </repositories>

  <dependencies>
    ...
    <dependency>
        <groupId>org.apache.phoenix</groupId>
        <artifactId>phoenix-core</artifactId>
        <version>[version]</version>
    </dependency>
    ...
  </dependencies>
```

Note: [version] can be replaced by 3.1.0, 4.1.0, 3.0.0-incubating, 4.0.0-incubating, etc.

Branches

Phoenix 3.0 is running against hbase0.94+, Phoenix 4.0 is running against hbase0.98.1+ and Phoenix master branch is running against hbase trunk branch.

See also:

Building Project Web Site

How to do a release

> 原文: *http://phoenix.apache.org/building.html*

Tuning

Tuning Guide

Tuning Phoenix can be complex, but with a little knowledge of how it works you can make significant changes to the performance of your reads and writes. The most important factor in performance is the design of your schema, especially as it affects the underlying HBase row keys. Look in “General Tips” below to find design advice for different anticipated data access patterns. Subsequent sections describe how to use secondary indexes, hints, and explain plans.

Note: Phoenix and HBase work well when your application does point lookups and small range scans. This can be achieved by good primary key design (see below). If you find that your application requires many full table scans, then Phoenix and HBase are likely not the best tool for the job. Instead, look at using other tools that write to HDFS directly using columnar representations such as Parquet.

Primary Keys

The underlying row key design is the single most important factor in Phoenix performance, and it’s important to get it right at design time because you cannot change it later without re-writing the data and index tables.

The Phoenix primary keys are concatenated to create the underlying row key in Apache HBase. The columns for the primary key constraint should be chosen and ordered in a way that aligns with the common query patterns—choose the most frequently queried columns as primary keys. The key that you place in the leading position is the most performant one. For example, if you lead off with a column containing org ID values, it is easy to select all rows pertaining to a specific org. You can add the HBase row timestamp to the primary key to improve scan efficiency by skipping rows outside the queried time range.

Every primary key imposes a cost because the entire row key is appended to every piece of data in memory and on disk. The larger the row key, the greater the storage overhead. Find ways to store information compactly in columns you plan to use for primary keys—store deltas instead of complete time stamps, for example.

To sum up, the best practice is to design primary keys to add up to a row key that lets you scan the smallest amount of data.

Tip: When choosing primary keys, lead with the column you filter most frequently across the queries that are most important to optimize. If you will use `ORDER BY` in your query, make sure your PK columns match the expressions in your `ORDER BY` clause.

Monotonically increasing Primary keys

If your primary keys are monotonically increasing, use salting to help distribute writes across the cluster and improve parallelization. Example:

CREATE TABLE … (…) SALT_BUCKETS = N

For optimal performance the number of salt buckets should approximately equal the number of region servers. Do not salt automatically. Use salting only when experiencing hotspotting. The downside of salting is that it imposes a cost on read because when you want to query the data you have to run multiple queries to do a range scan.

General Tips

The following sections provide a few general tips for different access scenarios.

Is the Data Random-Access?

- As with any random read workloads, SSDs can improve performance because of their faster random seek time.

Is the data read-heavy or write-heavy?

- For read-heavy data:
  - Create global indexes. This will affect write speed depending on the number of columns included in an index because each index writes to its own separate table.
  - Use multiple indexes to provide fast access to common queries.
  - When specifying machines for HBase, do not skimp on cores; HBase needs them.
- For write-heavy data:
  - Pre-split the table. It can be helpful to split the table into pre-defined regions, or if the keys are monotonically increasing use salting to to avoid creating write hotspots on a small number of nodes. Use real data types rather than raw byte data.
  - Create local indexes. Reads from local indexes have a performance penalty, so it’s important to do performance testing. See the Pherf tool.

Which columns will be accessed often?

- Choose commonly-queried columns as primary keys. For more information, see “Primary Keys” below.
  - Create additional indexes to support common query patterns, including heavily accessed fields that are not in the primary key.

Can the data be append-only (immutable)?

- If the data is immutable or append-only, declare the table and its indexes as immutable using the IMMUTABLE_ROWSoption at creation time to reduce the write-time cost. If you need to make an existing table immutable, you can do so with ALTER TABLE trans.event SET IMMUTABLE_ROWS=true after creation time.
    - If speed is more important than data integrity, you can use the DISABLE_WALoption. Note: it is possible to lose data with DISABLE_WAL if a region server fails.
- Set the UPDATE_CACHE_FREQUENCYoption to 15 minutes or so if your metadata doesn't change very often. This property determines how often an RPC is done to ensure you're seeing the latest schema.
- If the data is not sparse (over 50% of the cells have values), use the SINGLE_CELL_ARRAY_WITH_OFFSETS data encoding scheme introduced in Phoenix 4.10, which obtains faster performance by reducing the size of the data. For more information, see "Column Mapping and Immutable Data Encoding" on the Apache Phoenix blog.

Is the table very large?

- Use the ASYNC keyword with your CREATE INDEX call to create the index asynchronously via MapReduce job. You'll need to manually start the job; see https://phoenix.apache.org/secondary_indexing.html#Index_Population for details.
- If the data is too large to scan the table completely, use primary keys to create an underlying composite row key that makes it easy to return a subset of the data or facilitates skip-scanning—Phoenix can jump directly to matching keys when the query includes key sets in the predicate.

Is transactionality required?

A transaction is a data operation that is atomic—that is, guaranteed to succeed completely or not at all. For example, if you need to make cross-row updates to a data table, then you should consider your data transactional.

- If you need transactionality, use the TRANSACTIONALoption. (See also http://phoenix.apache.org/transactions.html.)

Block Encoding

Using compression or encoding is a must. Both SNAPPY and FAST_DIFF are good all around options.

FAST_DIFF encoding is automatically enabled on all Phoenix tables by default, and almost always improves overall read latencies and throughput by allowing more data to fit into blockcache. Note: FAST_DIFF encoding can increase garbage produced during request processing.

Set encoding at table creation time. Example: CREATE TABLE … (…)

DATA_BLOCK_ENCODING='FAST_DIFF'

Schema Design

Because the schema affects the way the data is written to the underlying HBase layer, Phoenix performance relies on the design of your tables, indexes, and primary keys.

Phoenix and the HBase data model

HBase stores data in tables, which in turn contain columns grouped in column families. A row in an HBase table consists of versioned cells associated with one or more columns. An HBase row is a collection of many key-value pairs in which the rowkey attribute of the keys are equal. Data in an HBase table is sorted by the rowkey, and all access is via the rowkey. Phoenix creates a relational data model on top of HBase, enforcing a PRIMARY KEY constraint whose columns are concatenated to form the row key for the underlying HBase table. For this reason, it's important to be cognizant of the size and number of the columns you include in the PK constraint, because a copy of the row key is included with every cell in the underlying HBase table.

Column Families

If some columns are accessed more frequently than others, create multiple column families to separate the frequently-accessed columns from rarely-accessed columns. This improves performance because HBase reads only the column families specified in the query.

Columns

Here are a few tips that apply to columns in general, whether they are indexed or not:

- Keep VARCHAR columns under 1MB or so due to I/O costs. When processing queries, HBase materializes cells in full before sending them over to the client, and the client receives them in full before handing them off to the application code.
- For structured objects, don't use JSON, which is not very compact. Use a format such as protobuf, Avro, msgpack, or BSON.
- Consider compressing data before storage using a fast LZ variant to cut latency and I/O costs.
- Use the column mapping feature (added in Phoenix 4.10), which uses numerical HBase column qualifiers for non-PK columns instead of directly using column names. This improves performance when looking for a cell in the sorted list of cells returned by HBase, adds further across-the-board performance by reducing the disk size used by tables, and speeds up DDL operations like column rename and metadata-level column drops. For more information, see "Column Mapping and Immutable Data Encoding" on the Apache Phoenix blog.

Indexes

A Phoenix index is a physical table that stores a pivoted copy of some or all of the data in the main table, to serve specific kinds of queries. When you issue a query, Phoenix selects the best index for the query automatically. The primary index is created automatically based on the primary keys you select. You can create secondary indexes, specifying which columns are included based on the anticipated queries the index will support.

See also: Secondary Indexing

Secondary indexes

Secondary indexes can improve read performance by turning what would normally be a full table scan into a point lookup (at the cost of storage space and write speed). Secondary indexes can be added or removed after table creation and don’t require changes to existing queries – queries simply run faster. A small number of secondary indexes is often sufficient. Depending on your needs, consider creating *covered* indexes or *functional* indexes, or both.

If your table is large, use the `ASYNC` keyword with `CREATE INDEX` to create the index asynchronously. In this case, the index will be built through MapReduce, which means that the client going up or down won’t impact index creation and the job is retried automatically if necessary. You’ll need to manually start the job, which you can then monitor just as you would any other MapReduce job.

Example: `create index if not exists event_object_id_idx_b on trans.event (object_id) ASYNC UPDATE_CACHE_FREQUENCY=60000;`

See Index Population for details.

If you can’t create the index asynchronously for some reason, then increase the query timeout (`phoenix.query.timeoutMs`) to be larger than the time it’ll take to build the index. If the `CREATE INDEX` call times out or the client goes down before it’s finished, then the index build will stop and must be run again. You can monitor the index table as it is created—you’ll see new regions created as splits occur. You can query the `SYSTEM.STATS` table, which gets populated as splits and compactions happen. You can also run a `count(*)` query directly against the index table, though that puts more load on your system because requires a full table scan.

Tips:

- Create local indexes for write-heavy use cases.
- Create global indexes for read-heavy use cases. To save read-time overhead, consider creating covered indexes.
- If the primary key is monotonically increasing, create salt buckets. The salt

buckets can’t be changed later, so design them to handle future growth. Salt buckets help avoid write hotspots, but can decrease overall throughput due to the additional scans needed on read.
- Set up a cron job to build indexes. Use ASYNC with CREATE INDEX to avoid blocking.
- Only create the indexes you need.
- Limit the number of indexes on frequently updated tables.
- Use covered indexes to convert table scans into efficient point lookups or range queries over the index table instead of the primary table: CREATE INDEX index ON table(…)INCLUDE(…)

Queries

It’s important to know which queries are executed on the server side versus the client side, because this can impact performace due to network I/O and other bottlenecks. If you’re querying a billion-row table, you want to do as much computation as possible on the server side rather than transmitting a billion rows to the client for processing. Some queries, on the other hand, must be executed on the client. Sorting data that lives on multiple region servers, for example, requires that you aggregate and re-sort on the client.

Reading

- Avoid joins unless one side is small, especially on frequent queries. For larger joins, see “Hints,” below.
- In the WHERE clause, filter leading columns in the primary key constraint.
- Filtering the first leading column with IN or OR in the WHERE clause enables skip scan optimizations.
- Equality or comparisions (< or >) in the WHERE clause enables range scan optimizations.
- Let Phoenix optimize query parallelism using statistics. This provides an automatic benefit if using Phoenix 4.2 or greater in production.
  See also: https://phoenix.apache.org/joins.html

Range Queries

If you regularly scan large data sets from spinning disk, you’re best off with GZIP (but watch write speed). Use a lot of cores for a scan to utilize the available memory bandwidth. Apache Phoenix makes it easy to utilize many cores to increase scan performance.

For range queries, the HBase block cache does not provide much advantage.

Large Range Queries

For large range queries, consider setting `Scan.setCacheBlocks(false)` even if the whole scan could fit into the block cache.

If you mostly perform large range queries you might even want to consider running HBase with a much smaller heap and size the block cache down, to only rely on the OS Cache. This will alleviate some garbage collection related issues.

Point Lookups

For point lookups it is quite important to have your data set cached, and you should use the HBase block cache.

Hints

Hints let you override default query processing behavior and specify such factors as which index to use, what type of scan to perform, and what type of join to use.

- During the query, Hint global index if you want to force it when query includes a column not in the index.
- If necessary, you can do bigger joins with the /+ *USE_SORT_MERGE_JOIN* / hint, but a big join will be an expensive operation over huge numbers of rows.
- If the overall size of all right-hand-side tables would exceed the memory size limit, use the /+ *NO_STAR_JOIN* /hint.
  See also: Hint.

Explain Plans

An `EXPLAIN` plan tells you a lot about how a query will be run. To generate an explain plan run this query and to interpret the plan, see this reference.

Parallelization

You can improve parallelization with the UPDATE STATISTICS command. This command subdivides each region by determining keys called *guideposts* that are equidistant from each other, then uses these guideposts to break up queries into multiple parallel scans. Statistics are turned on by default. With Phoenix 4.9, the user can set guidepost width for each table. Optimal guidepost width depends on a number of factors such as cluster size, cluster usage, number of cores per node, table size, and disk I/O.

In Phoenix 4.12, we have added a new configuration `phoenix.use.stats.parallelization` that controls whether statistics should be used for driving parallelization. Note that one can still run stats collection. The information collected is used to surface estimates on number of bytes and rows a query will scan when an EXPLAIN is generated for it.

Writing

Updating data with UPSERT VALUES

When using UPSERT VALUES to write a large number of records, turn off autocommit and batch records in reasonably small batches (try 100 rows and adjust from there to fine-tune performance).

Note: With the default fat driver, executeBatch() will not provide any benefit. Instead update mutliple rows by executing UPSERT VALUES mutliple times and then use commit() to submit the batch to the cluster. With the thin driver, however, it's important to use executeBatch() as this will minimize the number of RPCs between the client and query server.

```
try (Connection conn = DriverManager.getConnection(url)) {
  conn.setAutoCommit(false);
  int batchSize = 0;
  int commitSize = 1000; // number of rows you want to commit per batch.
  try (Statement stmt = conn.prepareStatement(upsert)) {
    stmt.set ... while (there are records to upsert) {
      stmt.executeUpdate();
      batchSize++;
      if (batchSize % commitSize == 0) {
        conn.commit();
      }
   }
 conn.commit(); // commit the last batch of records
 }
```

Note: Because the Phoenix client keeps uncommitted rows in memory, be careful not to set commitSize too high.

Updating data with UPSERT SELECT

When using UPSERT SELECT to write many rows in a single statement, turn on autocommit and the rows will be automatically batched according to the phoenix.mutate.batchSize. This will minimize the amount of data returned back to the client and is the most efficient means of updating many rows.

Deleting data

When deleting a large data set, turn on autoCommit before issuing the DELETE query so that the client does not need to remember the row keys of all the keys as they are deleted. This prevents the client from buffering the rows affected by the DELETE so that Phoenix can delete them directly on the region servers without the expense of returning them to the client.

Reducing RPC traffic

To reduce RPC traffic, set the `UPDATE_CACHE_FREQUENCY` (4.7 or above) on your table and indexes when you create them (or issue an `ALTER TABLE`/`INDEX` call. See https://phoenix.apache.org/#Altering.

Using local indexes

If using 4.8, consider using local indexes to minimize the write time. In this case, the writes for the secondary index will be to the same region server as your base table. This approach does involve a performance hit on the read side, though, so make sure to quantify both write speed improvement and read speed reduction.

Further Tuning

For advice about tuning the underlying HBase and JVM layers, see Operational and Performance Configuration Options in the Apache HBase™ Reference Guide.

Special Cases

The following sections provide Phoenix-specific additions to the tuning recommendations in the Apache HBase™ Reference Guide section referenced above.

For applications where failing quickly is better than waiting

In addition to the HBase tuning referenced above, set `phoenix.query.timeoutMs` in `hbase-site.xml` on the client side to the maximum tolerable wait time in milliseconds.

For applications that can tolerate slightly out of date information

In addition to the HBase tuning referenced above, set `phoenix.connection.consistency = timeline` in `hbase-site.xml` on the client side for all connections.

> 原文: *http://phoenix.apache.org/tuning_guide.html*

Explain Plan

Explain Plan

An EXPLAIN plan tells you a lot about how a query will be run:

- All the HBase range queries that will be executed
- An estimate of the number of bytes that will be scanned
- An estimate of the number of rows that will be traversed
- Time at which the above estimate information was collected
- Which HBase table will be used for each scan

- Which operations (sort, merge, scan, limit) are executed on the client versus the server
  Use an EXPLAIN plan to check how a query will run, and consider rewriting queries to meet the following goals:

- Emphasize operations on the server rather than the client. Server operations are distributed across the cluster and operate in parallel, while client operations execute within the single client JDBC driver.

- Use RANGE SCAN or SKIP SCAN whenever possible rather than TABLE SCAN.
- Filter against leading columns in the primary key constraint. This assumes you have designed the primary key to lead with frequently-accessed or frequently-filtered columns as described in “Primary Keys,” above.
- If necessary, introduce a local index or a global index that covers your query.
- If you have an index that covers your query but the optimizer is not detecting it, try hinting the query: SELECT /+ *INDEX() / …*
  See also: http://phoenix.apache.org/language/index.html#explain

Anatomy of an Explain Plan

An explain plan consists of lines of text that describe operations that Phoenix will perform during a query, using the following terms:

- AGGREGATE INTO ORDERED DISTINCT ROWS—aggregates the returned rows using an operation such as addition. When ORDERED is used, the GROUP BY operation is applied to the leading part of the primary key constraint, which allows the aggregation to be done in place rather than keeping all distinct groups in memory on the server side.
- AGGREGATE INTO SINGLE ROW—aggregates the results into a single row using an aggregate function with no GROUP BY clause. For example, the count() statement returns one row with the total number of rows that match the query.
- CLIENT—the operation will be performed on the client side. It’s faster to perform most operations on the server side, so you should consider whether there’s a way

to rewrite the query to give the server more of the work to do.
- FILTER BY expression—returns only results that match the expression.
- FULL SCAN OVER tableName—the operation will scan every row in the specified table.
- INNER-JOIN—the operation will join multiple tables on rows where the join condition is met.
- MERGE SORT—performs a merge sort on the results.
- RANGE SCAN OVER tableName […]—The information in the square brackets indicates the start and stop for each primary key that's used in the query.
- ROUND ROBIN—when the query doesn't contain ORDER BY and therefore the rows can be returned in any order, ROUND ROBIN order maximizes parallelization on the client side.
- x-CHUNK—describes how many threads will be used for the operation. The maximum parallelism is limited to the number of threads in thread pool. The minimum parallelization corresponds to the number of regions the table has between the start and stop rows of the scan. The number of chunks will increase with a lower guidepost width, as there is more than one chunk per region.
- PARALLELx-WAY—describes how many parallel scans will be merge sorted during the operation.
- SERIAL—some queries run serially. For example, a single row lookup or a query that filters on the leading part of the primary key and limits the results below a configurable threshold.
- EST_BYTES_READ - provides an estimate of the total number of bytes that will be scanned as part of executing the query
- EST_ROWS_READ - provides an estimate of the total number of rows that will be scanned as part of executing the query
- EST_INFO_TS - epoch time in milliseconds at which the estimate information was collected

Example

```
+----------------------------------------------------------------------------------------------------------------------------------+
|                                          PLAN                                         | EST_BYTES_READ  | EST_ROWS_READ  | EST_INFO_TS  |
+----------------------------------------------------------------------------------------------------------------------------------+
| CLIENT 36-CHUNK 237878 ROWS 6787437019 BYTES PARALLEL 36-WAY FULL SCAN
| OVER exDocStoreb                                                                      |     237878      | 6787437019    | 1510353318102|
|   PARALLEL INNER-JOIN TABLE 0 (SKIP MERGE)                                           |     237878      | 6787437019    | 1510353318102|
|     CLIENT 36-CHUNK PARALLEL 36-WAY RANGE SCAN OVER indx_exdocb
|      [0,' 42ecf4abd4bd7e7606025dc8eee3de 6a3cc04418cbc2619ddc01f54d88d7 c3bf']
|      - [0,' 42ecf4abd4bd7e7606025dc8eee3de 6a3cc04418cbc2619ddc01f54d88d7 c3bg' |     237878      | 6787437019    | 1510353318102|
|       SERVER FILTER BY FIRST KEY ONLY                                                 |     237878      | 6787437019    | 1510353318102|
|       SERVER AGGREGATE INTO ORDERED DISTINCT ROWS BY ["ID"]                          |     237878      | 6787437019    | 1510353318102|
|     CLIENT MERGE SORT                                                                 |     237878      |
6787437019    | 1510353318102|
|   DYNAMIC SERVER FILTER BY (A.CURRENT_TIMESTAMP, [A.ID](http://a.id/))
    IN ((TMP.MCT, TMP.TID))                                                      |     237878      |
6787437019    | 1510353318102|
+------------------------------------------------------------------------------------------------------------
----------------------
```

JDBC Explain Plan API and the estimates information

The information displayed in the explain plan API can also be accessed programmatically through the standard JDBC interfaces. When statistics collection is enabled for a table, the explain plan also gives an estimate of number of rows and bytes a query is going to scan. To get hold of the info, you can use corresponding columns in the result set returned by the explain plan statement. When stats collection is not enabled or if for some reason Phoenix cannot provide the estimate information, the columns return null. Below is an example:

```
String explainSql = "EXPLAIN SELECT * FROM T";
Long estimatedBytes = null;
Long estimatedRows = null;
Long estimateInfoTs = null;
try (Statement statement = conn.createStatement(explainSql)) {
        int paramIdx = 1;
        ResultSet rs = statement.executeQuery(explainSql);
        rs.next();
        estimatedBytes =
                (Long) rs.getObject(PhoenixRuntime.EXPLAIN_PLAN_ESTIMATED_BYTES_READ_COLUMN);
        estimatedRows =
                (Long) rs.getObject(PhoenixRuntime.EXPLAIN_PLAN_ESTIMATED_ROWS_READ_COLUMN);
        estimateInfoTs =
                (Long) rs.getObject(PhoenixRuntime.EXPLAIN_PLAN_ESTIMATE_INFO_TS_COLUMN);
}
```

> 原文: *http://phoenix.apache.org/explainplan.html*

Configuration

Configuration

Phoenix provides many different knobs and dials to configure and tune the system to run more optimally on your cluster. The configuration is done through a series of Phoenix-specific properties specified both on client and server-side `hbase-site.xml` files. In addition to these properties, there are of course all the HBase configuration properties with the most important ones documented here. The table below outlines the full set of Phoenix-specific configuration properties and their defaults.

|**Property**|**Description**|**Default**
|data.tx.snapshot.dir|Server-side property specifying the HDFS directory used to store snapshots of the transaction state. No default value.|None
|data.tx.timeout|Server-side property specifying the timeout in seconds for a transaction to complete. Default is 30 seconds.|30
|phoenix.query.timeoutMs|Client-side property specifying the number of milliseconds after which a query will timeout on the client. Default is 10 min.|600000
|phoenix.query.keepAliveMs| Maximum time in milliseconds that excess idle threads will wait for a new tasks before terminating when the number of threads is greater than the cores in the client side thread pool executor. Default is 60 sec.|60000
|phoenix.query.threadPoolSize|Number of threads in client side thread pool executor. As the number of machines/cores in the cluster grows, this value should be increased.|128
|phoenix.query.queueSize|Max queue depth of the bounded round robin backing the client side thread pool executor, beyond which an attempt to queue additional work is rejected. If zero, a SynchronousQueue is used instead of the bounded round robin queue. The default value is 5000.|5000
|phoenix.stats.guidepost.width| Server-side parameter that specifies the number of bytes between guideposts. A smaller amount increases parallelization, but also increases the number of chunks which must be merged on the client side. The default value is 100 MB. |104857600
|phoenix.stats.guidepost.per.region| Server-side parameter that specifies the number of guideposts per region. If set to a value greater than zero, then the guidepost width is determiend by `MAX_FILE_SIZE of table / phoenix.stats.guidepost.per.region`. Otherwise, if not set, then the `phoenix.stats.guidepost.width` parameter is used. No default value. |None
|phoenix.stats.updateFrequency| Server-side paramater that determines the frequency in milliseconds for which statistics will be refreshed from the statistics table and subsequently used by the client. The default value is 15 min. |900000
|phoenix.stats.minUpdateFrequency| Client-side parameter that determines the minimum amount of time in milliseconds that must pass before statistics may again be manually collected through another `UPDATE STATISTICS` call. The default value is

phoenix.stats.updateFrequency / 2. |450000
|phoenix.stats.useCurrentTime| Server-side parameter that if true causes the current time on the server-side to be used as the timestamp of rows in the statistics table when background tasks such as compactions or splits occur. If false, then the max timestamp found while traversing the table over which statistics are being collected is used as the timestamp. Unless your client is controlling the timestamps while reading and writing data, this parameter should be left alone. The default value is true. |true
|phoenix.query.spoolThresholdBytes|Threshold size in bytes after which results from parallelly executed query results are spooled to disk. Default is 20 mb.|20971520
|phoenix.query.maxSpoolToDiskBytes|Threshold size in bytes up to which results from parallelly executed query results are spooled to disk above which the query will fail. Default is 1 GB.|1024000000
|phoenix.query.maxGlobalMemoryPercentage|Percentage of total heap memory (i.e. Runtime.getRuntime().maxMemory()) that all threads may use. Only course grain memory usage is tracked, mainly accounting for memory usage in the intermediate map built during group by aggregation. When this limit is reached the clients block attempting to get more memory, essentially throttling memory usage. Defaults to 15%|15
|phoenix.query.maxGlobalMemorySize|Max size in bytes of total tracked memory usage. By default not specified, however, if present, the lower of this parameter and the phoenix.query.maxGlobalMemoryPercentage will be used |
|phoenix.query.maxGlobalMemoryWaitMs|Maximum amount of time that a client will block while waiting for more memory to become available. After this amount of time, an InsufficientMemoryException is thrown. Default is 10 sec.|10000
|phoenix.query.maxTenantMemoryPercentage|Maximum percentage of phoenix.query.maxGlobalMemoryPercentage that any one tenant is allowed to consume. After this percentage, an InsufficientMemoryException is thrown. Default is 100%|100
|phoenix.query.dateFormat|Default pattern to use for conversion of a date to/from a string, whether through the TO_CHAR(<date>) or TO_DATE(<date-string>) functions, or through resultSet.getString(<date-column>). Default is yyyy-MM-dd HH

ss|yyyy-MM-dd HH ss
|phoenix.query.dateFormatTimeZone|A timezone id that specifies the default time zone in which date, time, and timestamp literals should be interpreted when interpreting string literals or using the TO_DATE function. A time zone id can be a timezone abbreviation such as “PST”, or a full name such as “America/Los_Angeles”, or a custom offset such as “GMT-9:00”. The time zone id “LOCAL” can also be used to interpret all date, time, and timestamp literals as being in the current timezone of the client.|GMT
|phoenix.query.numberFormat|Default pattern to use for conversion of a decimal number to/from a string, whether through the TO_CHAR(<decimal-number>) or TO_NUMBER(<decimal-string>) functions, or through resultSet.getString(<decimal-column>). Default is #,##0.###|#,##0.###

|phoenix.mutate.maxSize|The maximum number of rows that may be batched on the client before a commit or rollback must be called.|500000
|phoenix.mutate.batchSize|The number of rows that are batched together and automatically committed during the execution of an UPSERT SELECT or DELETE statement. This property may be overridden at connection time by specifying the UpsertBatchSize property value. Note that the connection property value does not affect the batch size used by the coprocessor when these statements are executed completely on the server side.|1000
|phoenix.query.maxServerCacheBytes|Maximum size (in bytes) of a single sub-query result (usually the filtered result of a table) before compression and conversion to a hash map. Attempting to hash an intermediate sub-query result of a size bigger than this setting will result in a MaxServerCacheSizeExceededException. Default 100MB.|104857600
|phoenix.coprocessor.maxServerCacheTimeToLiveMs|Maximum living time (in milliseconds) of server caches. A cache entry expires after this amount of time has passed since last access. Consider adjusting this parameter when a server-side IOException(“Could not find hash cache for joinId”) happens. Getting warnings like “Earlier hash cache(s) might have expired on servers” might also be a sign that this number should be increased.|30000
|phoenix.query.useIndexes|Client-side property determining whether or not indexes are considered by the optimizer to satisfy a query. Default is true |true
|phoenix.index.failure.handling.rebuild|Server-side property determining whether or not a mutable index is rebuilt in the background in the event of a commit failure. Only applicable for indexes on mutable, non transactional tables. Default is true |true
|phoenix.index.failure.block.write|Server-side property determining whether or not a writes to the data table are disallowed in the event of a commit failure until the index can be caught up with the data table. Requires that phoenix.index.failure.handling.rebuild is true as well. Only applicable for indexes on mutable, non transactional tables. Default is false |false
|phoenix.index.failure.handling.rebuild.interval|Server-side property controlling the millisecond frequency at which the server checks whether or not a mutable index needs to be partially rebuilt to catch up with updates to the data table. Only applicable for indexes on mutable, non transactional tables. Default is 10 seconds. |10000
|phoenix.index.failure.handling.rebuild.overlap.time|Server-side property controlling how many milliseconds to go back from the timestamp at which the failure occurred to go back when a partial rebuild is performed. Only applicable for indexes on mutable, non transactional tables. Default is 1 millisecond. |1
|phoenix.index.mutableBatchSizeThreshold|Number of mutations in a batch beyond which index metadata will be sent as a separate RPC to each region server as opposed to included inline with each mutation. Defaults to 5. |5
|phoenix.schema.dropMetaData|Determines whether or not an HBase table is dropped when the Phoenix table is dropped. Default is true |true
|phoenix.groupby.spillable|Determines whether or not a GROUP BY over a large number of distinct values is allowed to spill to disk on the region server. If false, an InsufficientMemoryException will be thrown instead. Default is true |true

```
|phoenix.groupby.spillFiles|Number of memory mapped spill files to be used when
spilling GROUP BY distinct values to disk. Default is 2 |2
|phoenix.groupby.maxCacheSize|Size in bytes of pages cached during GROUP BY spilling.
Default is 100Mb |102400000
|phoenix.groupby.estimatedDistinctValues|Number of estimated distinct values when a
GROUP BY is performed. Used to perform initial sizing with growth of 1.5x each time
reallocation is required. Default is 1000 |1000
|phoenix.distinct.value.compress.threshold|Size in bytes beyond which aggregate
operations which require tracking distinct value counts (such as COUNT DISTINCT) will
use Snappy compression. Default is 1Mb |1024000
|phoenix.index.maxDataFileSizePerc|Percentage used to determine the MAX_FILESIZE for
the shared index table for views relative to the data table MAX_FILESIZE. The
percentage should be estimated based on the anticipated average size of an view index
row versus the data row. Default is 50%. |50
|phoenix.coprocessor.maxMetaDataCacheTimeToLiveMs|Time in milliseconds after which the
server-side metadata cache for a tenant will expire if not accessed. Default is 30mins
|180000
|phoenix.coprocessor.maxMetaDataCacheSize|Max size in bytes of total server-side
metadata cache after which evictions will begin to occur based on least recent access
time. Default is 20Mb |20480000
|phoenix.client.maxMetaDataCacheSize|Max size in bytes of total client-side metadata
cache after which evictions will begin to occur based on least recent access time.
Default is 10Mb |10240000
|phoenix.sequence.cacheSize|Number of sequence values to reserve from the server and
cache on the client when the next sequence value is allocated. Only used if not
defined by the sequence itself. Default is 100 |100
|phoenix.clock.skew.interval|Delay interval(in milliseconds) when opening
SYSTEM.CATALOG to compensate possible time clock skew when SYSTEM.CATALOG moves among
region servers. |2000
|phoenix.index.failure.handling.rebuild|Boolean flag which turns on/off auto-rebuild a
failed index from when some updates are failed to be updated into the index. |true
|phoenix.index.failure.handling.rebuild.interval|Time interval(in milliseconds) for
index rebuild backend Job to check if there is an index to be rebuilt |10000
|phoenix.index.failure.handling.rebuild.overlap.time|Index rebuild job builds an index
from when it failed - the time interval(in milliseconds) in order to create a time
overlap to prevent missing updates when there exists time clock skew. |300000
|phoenix.query.force.rowkeyorder|Whether or not a non aggregate query returns rows in
row key order for salted tables. For version prior to 4.4, use
phoenix.query.rowKeyOrderSaltedTable instead. Default is true.|true
|phoenix.connection.autoCommit|Whether or not a new connection has auto-commit enabled
when it is created. Default is false.|false
|phoenix.table.default.store.nulls|The default value of the STORE_NULLS flag used for
table creation which determines whether or not null values should be explicitly stored
in HBase. Default is false. This is a client side parameter. Available starting from
Phoenix 4.3.|false
|phoenix.table.istransactional.default|The default value of the TRANSACTIONAL flag
```

used for table creation which determines whether or not a table is transactional . Default is false. This is a client side parameter. Available starting from Phoenix 4.7.|false
|phoenix.transactions.enabled| Determines whether or not transactions are enabled in Phoenix. A table may not be declared as transactional if transactions are disabled. Default is false. This is a client side parameter. Available starting from Phoenix 4.7.|false
|phoenix.mapreduce.split.by.stats|Determines whether to use the splits determined by stastics for MapReduce input splits. Default is true. This is a server side parameter. Available starting from Phoenix 4.10. Set to false to enable behavior from previous versions.|true

> 原文: *http://phoenix.apache.org/tuning.html*

Backward Compatibility

Backward Compatibility

Phoenix maintains backward compatibility across at least two minor releases to allow for **no downtime** through server-side rolling restarts upon upgrading. See below for details.

Versioning Convention

Phoenix uses a standard three number versioning schema of the form:

```
<major version> . <minor version> . <patch version>
```

For example, **4.2.1** has a major version of **4**, a minor version of **2**, and a patch version of **1**.

Patch Release

Upgrading to a new patch release (i.e. only the patch version has changed) is the simplest case. The jar upgrade may occur in any order: client first or server first, and a mix of clients with different patch release versions is fine.

Minor Release

When upgrading to a new minor release (i.e. the major version is the same, but the minor version has changed), sometimes modifications to the system tables are necessary to either fix a bug or provide a new feature. This upgrade will occur automatically the first time a newly upgraded client connects to the newly upgraded server. It is **required** that the server-side jar be upgraded first across your entire cluster, before any clients are upgraded. An older client (two minor versions back) will work with a newer server jar when the minor version is different, but not visa versa. In other words, clients do not need to be upgraded in lock step with the server. However, as the server version moves forward, the client version should move forward as well. This allows Phoenix to evolve its client/server protocol while still providing clients sufficient time to upgrade their clients.

As of the 4.3 release, a mix of clients on different minor release versions is supported as well (note that prior releases required all clients to be upgraded at the same time). Another improvement as of the 4.3 release is that an upgrade may be done directly from one minor version to another higher minor version (prior releases required an upgrade to each minor version in between).

Major Release

Upgrading to a new major release may require downtime as well as potentially the running of a migration script. Additionally, all clients and servers may need to be upgraded at the same time. This will be determined on a release by release basis.

Release Notes

Specific details on issues and their fixes that may impact you may be found here.

> 原文: *http://phoenix.apache.org/upgrading.html*

Release Notes

Release Notes

Release notes provide details on issues and their fixes which may have an impact on prior Phoenix behavior. For some issues an upgrade may be required to be performed for a fix to take affect. See below for directions specific to a particular release.

Phoenix 5.0.0-alpha Release Notes

Phoenix 5.0.0-alpha is a “preview” release. This release is the first version of Phoenix which is compatible with Apache Hadoop 3.0.x and Apache HBase 2.0.x. This release also is designated an “alpha” release because there are several known deficiencies which impact the production readiness. This release should be used carefully by users who have taken the time to understand what is known to be working and what is not.

Known issues:

- The Apache Hive integration is known to be non-functional (PHOENIX-4423)
- Split/Merge logic with Phoenix local indexes are broken (PHOENIX-4440)
- Apache Tepha integration/transactional tables are non-functional (PHOENIX-4580)
- Point-in-time queries and tools that look at “old” cells are broken, e.g. IndexScrutiny (PHOENIX-4378)
  The developers would like to encourage users to test this release out and report any observed issues so that the official 5.0.0 release quality may be significantly improved.

Phoenix-4.8.0 Release Notes

PHOENIX-3164 is a relatively serious bug that affects the Phoenix Query Server deployed with “security enabled” (Kerberos or Active Directory). Due to another late-game change in the 4.8.0 release as well as an issue with the use of Hadoop’s UserGroupInformation class, every “client session” to the Phoenix Query Server with security enabled will result in a new instance of the Phoenix JDBC driver `PhoenixConnection` (and other related classes). This ultimately results in a new connection to ZooKeeper for each “client session”.

Within a short amount of time of active use with the Phoenix Query Server creating a new ZooKeeper connection for each “client session”, the number of ZooKeeper connections will have grown rapidly likely triggering ZooKeeper’s built-in denial of service protection (maxClientCnxns). This will cause all future connections to ZooKeeper by the host running the Phoenix Query Server to be dropped. This would prevent all HBase client API calls which need to access ZooKeeper from completing.

As part of PHOENIX-1734 we have changed the local index implementation to store index data in the separate column families in the same data table. So while upgrading the phoenix at server we need to remove below local index related configurations from `hbase-site.xml` and run upgrade steps mentioned here

```
<property>
  <name>hbase.master.loadbalancer.class</name>
  <value>org.apache.phoenix.hbase.index.balancer.IndexLoadBalancer</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.phoenix.hbase.index.master.IndexMasterObserver</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.regionserver.LocalIndexMerger</value>
</property>
```

Phoenix-4.5.0 Release Notes

Both PHOENIX-2067 and PHOENIX-2120 cause rows to not be ordered correctly for the following types of columns:

- VARCHAR DESC columns
- DECIMAL DESC columns
- ARRAY DESC columns
- Nullable DESC columns which are indexed (impacts the index, but not the data table)
- BINARY columns included in the primary key constraint
  To get an idea if any of your tables are impacted, you may run the following command:

```
./psql.py -u my_host_name
```

This will look through all tables you’ve defined and indicate if any upgrades are necessary. Ensure your client-side `phoenix.query.timeoutMs` property and server-side `hbase.regionserver.lease.period` are set high enough for the command to complete.

To upgrade the tables, run the same command, but list the tables you’d like upgraded like this:

```
./psql.py -u my_host_name table1 table2 table3
```

This will first make a snapshot of your table and then upgrade it. If any problems occur during the upgrade process, the snapshot of your original table will be restored. Again, make sure your timeouts are set high enough, as the tables being upgraded need to be rewritten in order to fix them.

For the case of BINARY columns, no update is required if you’ve always provided all of the bytes making up that column value (i.e. you have not relied on Phoenix to auto-pad the column up to the fixed length). In this case, you should bypass the upgrade by running the following command:

```
./psql.py -u -b my_host_name table1
```

This is important, because the PHOENIX-2120 was caused by BINARY columns being incorrectly padded with a space characters instead of a zero byte characters. The upgrade will replace trailing space characters with zero byte characters which may be invalid if the space characters are legitimate/intentional characters. Unfortunately, Phoenix has no way to know if this is the case.

Upgrading your tables is important, as without this, Phoenix will need to reorder rows it retrieves back from the server when otherwise not necessary. This will have a large negative impact on performance until the upgrade is performed.

Future releases of Phoenix may require that affected tables be upgraded prior to moving to the new release.

> 原文: *http://phoenix.apache.org/release_notes.html*

Performance Testing

pherf

Phoenix and HBase SQL Performance Test Framework

Overview

Pherf is a standalone tool that can perform performance and functional testing through Phoenix. Pherf can be used both generate highly customized data sets and to measure performance of SQL against that data.

Build all of Phoenix. This includes Pherf’s default profile

mvn clean package -DskipTests

Running

- Edit the config/env.sh to include the required property values.
- bin/pherf-standalone.py -h
- To use libraries included with HBase deployment on a cluster: bin/pherf-cluster.py -h
- Example: bin/pherf-cluster.py -drop all -l -q -z [zookeeper] -schemaFile *.user_defined_schema.sql -scenarioFile* .user_defined_scenario.xml HBASE_CONF_DIR, HBASE_DIR environment variable needs to be set to use against a cluster deployment

Example run commands.

List all scenario files available to run.

$./pherf-standalone.py -listFiles

Drop all existing tables, load and query data specified in all scenario files.

$./pherf-standalone.py -drop all -l -q -z localhost

Pherf arguments:

- -h *Help*
- -l *Apply schema and load data*
- -q *Executes Multi-threaded query sets and write results*
- -z [quorum] *Zookeeper quorum*
- -m *Enable monitor for statistics*
- -monitorFrequency [frequency in Ms] _Frequency at which the monitor will snopshot stats to log file.
- -drop [pattern] *Regex drop all tables with schema name as PHERF. Example drop Event tables: -drop .(EVENT). Drop all: -drop .* or -drop all*
- -scenarioFile *Regex or file name of a specific scenario file to run.*
- -schemaFile *Regex or file name of a specific schema file to run.*
- -export Exports query results to CSV files in CSV_EXPORT directory
- -diff Compares results with previously exported results
- -hint *Executes all queries with specified hint. Example SMALL*
- -rowCountOverride
- -rowCountOverride [number of rows] *Specify number of rows to be upserted rather than using row count specified in schema*

Adding Rules for Data Creation

Review test_scenario.xml for syntax examples.

- Rules are defined as and are applied in the order they appear in file.
- Rules of the same type override the values of a prior rule of the same type. If true is set, rule will only apply override when type and name match the column name in Phoenix.
- tag is set at the column level. It can be used to define a constant string appended to the beginning of CHAR and VARCHAR data type values.
- **Required field** Supported Phoenix types: VARCHAR, CHAR, DATE, DECIMAL, INTEGER
  - denoted by the tag
- User defined true changes rule matching to use both name and type fields to determine equivalence.
  - Default is false if not specified and equivalence will be determined by type only. **An important note here is that you can still override rules without the user defined flag, but they will change the rule globally and not just for a specified column.**
- **Required field** Supported Data Sequences
  - RANDOM: Random value which can be bound by other fields such as length.
  - SEQUENTIAL: Monotonically increasing long prepended to random strings.
    - Only supported on VARCHAR and CHAR types
  - LIST: Means pick values from predefined list of values
- **Required field** Length defines boundary for random values for CHAR and VARCHAR types.

    - denoted by the tag
- Column level Min/Max value defines boundaries for numerical values. For DATES, these values supply a range between which values are generated. At the column level the granularity is a year. At a specific data value level, the granularity is down to the Ms.
    - denoted by the tag
    - denoted by the tag
- Null chance denotes the probability of generating a null value. From [0-100]. The higher the number, the more likely the value will be null.
    - denoted by
- Name can either be any text or the actual column name in the Phoenix table.
    - denoted by the
- Value List is used in conjunction with LIST data sequences. Each entry is a DataValue with a specified value to be used when generating data.
    - Denoted by the tags
    - If the distribution attribute on the datavalue is set, values will be created according to that probability.
    - When distribution is used, values must add up to 100%.
    - If distribution is not used, values will be randomly picked from the list with equal distribution.

Defining Scenario

Scenario can have multiple querySets. Consider following example, concurrency of 1-4 means that each query will be executed starting with concurrency level of 1 and reach up to maximum concurrency of 4. Per thread, query would be executed to a minimum of 10 times or 10 seconds (whichever comes first). QuerySet by defult is executed serially but you can change executionType to PARALLEL so queries are executed concurrently. Scenarios are defined in XMLs stored in the resource directory.

```
<scenarios>
    <!--Minimum of executionDurationInMs or numberOfExecutions. Which ever is reached first -->
    <querySet concurrency="1-4" executionType="PARALLEL" executionDurationInMs="10000"
numberOfExecutions="10">
        <query id="q1" verifyRowCount="false" statement="select count(*) from PHERF.TEST_TABLE"/>
        <query id="q2" tenantId="1234567890" ddl="create view if not exists
        myview(mypk varchar not null primary key, mycol varchar)" statement="upsert select ..."/>
    </querySet>
    <querySet concurrency="3" executionType="SERIAL" executionDurationInMs="20000" numberOfExecutions="100">
        <query id="q3" verifyRowCount="false" statement="select count(*) from PHERF.TEST_TABLE"/>
        <query id="q4" statement="select count(*) from PHERF.TEST_TABLE WHERE TENANT_ID='00D000000000062'"/>
    </querySet>
</scenario>

```

Results

Results are written real time in *results* directory. Open the result that is saved in .jpg format for real time visualization.

Testing

Default quorum is localhost. If you want to override set the system variable.

Run unit tests: `mvn test -DZK_QUORUM=localhost` Run a specific method: `mvn -Dtest=ClassName#methodName test`

More to come…

> 原文: *http://phoenix.apache.org/pherf.html*

Apache Spark Integration

Apache Spark Plugin

The phoenix-spark plugin extends Phoenix's MapReduce support to allow Spark to load Phoenix tables as RDDs or DataFrames, and enables persisting them back to Phoenix.

Prerequisites

- Phoenix 4.4.0+
- Spark 1.3.1+ (prebuilt with Hadoop 2.4 recommended)

Why not JDBC?

Although Spark supports connecting directly to JDBC databases, it's only able to parallelize queries by partioning on a numeric column. It also requires a known lower bound, upper bound and partition count in order to create split queries.

In contrast, the phoenix-spark integration is able to leverage the underlying splits provided by Phoenix in order to retrieve and save data across multiple workers. All that's required is a database URL and a table name. Optional SELECT columns can be given, as well as pushdown predicates for efficient filtering.

The choice of which method to use to access Phoenix comes down to each specific use case.

Spark setup

- To ensure that all requisite Phoenix / HBase platform dependencies are available on the classpath for the Spark executors and drivers, set both '*spark.executor.extraClassPath*' and '*spark.driver.extraClassPath*' in spark-defaults.conf to include the 'phoenix--client.jar'

- Note that for Phoenix versions 4.7 and 4.8 you must use the 'phoenix--client-spark.jar'. As of Phoenix 4.10, the 'phoenix--client.jar' is compiled against Spark 2.x. If compability with Spark 1.x if needed, you must compile Phoenix with the spark16 maven profile.

- To help 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>
```

Reading Phoenix Tables

Given a Phoenix table with the following DDL

```
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');
```

Load as 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
```

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)

// Load the columns 'ID' and 'COL1' from TABLE1 as a DataFrame
val df = sqlContext.phoenixTableAsDataFrame(
  "TABLE1", Array("ID", "COL1"), conf = configuration
)

df.show
```

Load as an RDD, using a Zookeeper URL

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

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

// Load 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

Given a Phoenix table with the following DDL

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

Saving RDDs

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.

```scala
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")
  )
```

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.

Given two Phoenix tables with the following DDL:

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

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

// Load INPUT_TABLE
val sc = new SparkContext("local", "phoenix-test")
val sqlContext = new SQLContext(sc)
val df = sqlContext.load("org.apache.phoenix.spark", Map("table" -> "INPUT_TABLE",
  "zkUrl" -> hbaseConnectionString))

// Save to OUTPUT_TABLE
df.save("org.apache.phoenix.spark", SaveMode.Overwrite, Map("table" -> "OUTPUT_TABLE",
  "zkUrl" -> hbaseConnectionString))
```

PySpark

With Spark’s DataFrame support, you can also use `pyspark` to read and write from Phoenix tables.

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()
```

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()
```

Notes

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

- Basic support for column and predicate pushdown using the Data Source API
- The Data Source API does not support passing custom Phoenix settings in configuration, you must create the DataFrame or RDD directly if you need fine-grained configuration.
- No support for aggregate or distinct queries as explained in our Map Reduce Integration documentation.

PageRank example

This example makes use of the Enron email data set, provided by the Stanford Network Analysis Project, and executes the GraphX implementation of PageRank on it to find interesting entities. It then saves the results back to Phoenix.

- Download and extract the file enron.csv.gz

- Create the necessary Phoenix schema

CREATE TABLE EMAIL_ENRON(MAIL_FROM BIGINT NOT NULL, MAIL_TO BIGINT NOT NULL CONSTRAINT pk PRIMARY KEY(MAIL_FROM, MAIL_TO));
CREATE TABLE EMAIL_ENRON_PAGERANK(ID BIGINT NOT NULL, RANK DOUBLE CONSTRAINT pk PRIMARY KEY(ID));

- Load the email data into Phoenix (assuming localhost for Zookeeper Quroum URL)

gunzip /tmp/enron.csv.gz
cd /path/to/phoenix/bin
./psql.py -t EMAIL_ENRON localhost /tmp/enron.csv

- In spark-shell, with the phoenix-client in the Spark driver classpath, run the following:

import org.apache.spark.graphx.
import org.apache.phoenix.spark.
val rdd = sc.phoenixTableAsRDD("EMAIL_ENRON", Seq("MAIL_FROM", "MAIL_TO"), zkUrl=Some("localhost")) // load from phoenix
val rawEdges = rdd.map{ e => (e("MAIL_FROM").asInstanceOf[VertexId], e("MAIL_TO").asInstanceOf[VertexId]) } // map to vertexids
val graph = Graph.fromEdgeTuples(rawEdges, 1.0) // create a graph
val pr = graph.pageRank(0.001) // run pagerank
pr.vertices.saveToPhoenix("EMAIL_ENRON_PAGERANK", Seq("ID", "RANK"), zkUrl = Some("localhost")) // save to phoenix

- Query the top ranked entities in SQL

```
SELECT * FROM EMAIL_ENRON_PAGERANK ORDER BY RANK DESC LIMIT 5;
+——————————————+——————————————+
| ID | RANK |
+——————————————+——————————————+
| 5038 | 497.2989872977676 |
| 273 | 117.18141799210386 |
| 140 | 108.63091596789913 |
| 458 | 107.2728800448782 |
| 588 | 106.11840798585399 |
+——————————————+——————————————+
```

> 原文: *http://phoenix.apache.org/phoenix_spark.html*

Phoenix Storage Handler for Apache Hive

Phoenix Storage Handler for Apache Hive

The Apache Phoenix Storage Handler is a plugin that enables Apache Hive access to Phoenix tables from the Apache Hive command line using HiveQL.

Prerequisites

- Phoenix 4.8.0+
- Hive 1.2.1+

Hive Setup

Make phoenix-version-hive.jar available for Hive:

Step 1: Add to hive-env.sh:

```
HIVE_AUX_JARS_PATH=<path to jar>
```

Step 2: Add a property to hive-site.xml so that Hive MapReduce jobs can use the .jar:

```
<property>
  <name>hive.aux.jars.path</name>
  <value>file://<path></value>
</property>
```

Table Creation and Deletion

The Phoenix Storage Handler supports both INTERNAL and EXTERNAL Hive tables.

Create INTERNAL Table

For INTERNAL tables, Hive manages the lifecycle of the table and data. When a Hive table is created, a corresponding Phoenix table is also created. Once the Hive table is dropped, the Phoenix table is also deleted.

```
    create table phoenix_table (
      s1 string,
      i1 int,
      f1 float,
      d1 double
    )
    STORED BY 'org.apache.phoenix.hive.PhoenixStorageHandler'
    TBLPROPERTIES (
      "phoenix.table.name" = "phoenix_table",
      "phoenix.zookeeper.quorum" = "localhost",
      "phoenix.zookeeper.znode.parent" = "/hbase",
      "phoenix.zookeeper.client.port" = "2181",
      "phoenix.rowkeys" = "s1, i1",
      "phoenix.column.mapping" = "s1:s1, i1:i1, f1:f1, d1:d1",
      "phoenix.table.options" = "SALT_BUCKETS=10, DATA_BLOCK_ENCODING='DIFF'"
    );
```

Create EXTERNAL Table

For EXTERNAL tables, Hive works with an existing Phoenix table and manages only Hive metadata. Dropping an EXTERNAL table from Hive deletes only Hive metadata but does not delete the Phoenix 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"
);
```

Properties

- phoenix.table.name
  - Specifies the Phoenix table name
  - Default: the same as the Hive table
- phoenix.zookeeper.quorum
  - Specifies the ZooKeeper quorum for HBase
  - Default: localhost
- phoenix.zookeeper.znode.parent
  - Specifies the ZooKeeper parent node for HBase
  - Default: /hbase
- phoenix.zookeeper.client.port
  - Specifies the ZooKeeper port
  - Default: 2181

- phoenix.rowkeys
  - The list of columns to be the primary key in a Phoenix table
  - Required
- phoenix.column.mapping
  - Mappings between column names for Hive and Phoenix. See Limitations for details.

Data Ingestion, Deletions, and Updates

Data ingestion can be done by all ways that Hive and Phoenix support:

Hive:

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

Phoenix:

```
upsert into table T values (.....);
    Phoenix CSV BulkLoad tools
```

All delete and update operations should be performed on the Phoenix side. See Limitations for more details.

Additional Configuration Options

Those options can be set in a Hive command-line interface (CLI) environment.

Performance Tuning

Parameter	Default Value	Description
phoenix.upsert.batch.size	1000	Batch size for upsert.
[phoenix-table-name].disable.wal	false	Temporarily sets the table attribute DISABLE_WAL to true. Sometimes used to improve performance
[phoenix-table-name].auto.flush	false	When WAL is disabled and if this value is true, then MemStore is flushed to an HFile.

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

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

- Hive update and delete operations require transaction manager support on both Hive and Phoenix sides. Related Hive and Phoenix JIRAs are listed in the Resources section.
- Column mapping does not work correctly with mapping row key columns.
- MapReduce and Tez jobs always have a single reducer.

Resources

- PHOENIX-2743 : Implementation, accepted by Apache Phoenix community. Original pull request contains modification for Hive classes.
- PHOENIX-331 : An outdated implementation with support of Hive 0.98.

> 原文: *http://phoenix.apache.org/hive_storage_handler.html*

Apache Pig Integration

Apache Pig Integration

Pig integration may be divided into two parts: a **StoreFunc** as a means to generate Phoenix-encoded data through Pig, and a **Loader** which enables Phoenix-encoded data to be read by Pig.

Pig StoreFunc

The StoreFunc allows users to write data in Phoenix-encoded format to HBase tables using Pig scripts. This is a nice way to bulk upload data from a MapReduce job in parallel to a Phoenix table in HBase. All you need to specify is the endpoint address, HBase table name and a batch size. For example:

```
A = load 'testdata' as (a:chararray, b:chararray, c:chararray, d:chararray, e: datetime);
STORE A into 'hbase://CORE.ENTITY_HISTORY' using
    org.apache.phoenix.pig.PhoenixHBaseStorage('localhost','-batchSize 5000');
```

The above reads a file ‘testdata’ and writes the elements to a table “CORE.ENTITY_HISTORY” in HBase that is running on localhost. First argument to this StoreFunc is the server, the 2nd argument is the batch size for upserts via Phoenix. The batch size is related to how many rows you are able to hold in memory. A good default is 1000 rows, but if your row is wide, you may want to decrease this.

Note that Pig types must be in sync with the target Phoenix data types. This StoreFunc tries best to cast based on input Pig types and target Phoenix data types, but it is recommended to provide an appropriate schema.

Gotchas

It is advised that the upsert operation be idempotent. That is, trying to re-upsert data should not cause any inconsistencies. This is important in the case when a Pig job fails in process of writing to a Phoenix table. There is no notion of rollback (due to lack of transactions in HBase), and re-trying the upsert with PhoenixHBaseStorage must result in the same data in HBase table.

For example, let’s assume we are writing records n1…n10 to HBase. If the job fails in the middle of this process, we are left in an inconsistent state where n1…n7 made it to the phoenix tables but n8…n10 were missed. If we retry the same operation, n1…n7 would be re-upserted and n8…n10 would be upserted this time.

Pig Loader

A Pig data loader allows users to read data from Phoenix backed HBase tables within a Pig script.

The Load func provides two alternative ways to load data.

- Given a table name, the following will load the data for all the columns in the HIRES table:

A = load 'hbase://table/HIRES' using org.apache.phoenix.pig.PhoenixHBaseLoader('localhost');

To restrict the list of columns, you may specify the column names as part of LOAD as shown below:

A = load 'hbase://table/HIRES/ID,NAME' using org.apache.phoenix.pig.PhoenixHBaseLoader('localhost');

Here, only data for ID and NAME columns are returned.

- Given a query, the following loads data for all those rows whose AGE column has a value of greater than 50:

A = load 'hbase://query/SELECT ID,NAME FROM HIRES WHERE AGE > 50' using org.apache.phoenix.pig.PhoenixHBaseLoader('localhost');

The LOAD func merely executes the given SQL query and returns the results. Though there is a provision to provide a query as part of LOAD, it is restricted to the following:

- Only a SELECT query is allowed. No DML statements such as UPSERT or DELETE.
- The query may not contain any GROUP BY, ORDER BY, LIMIT, or DISTINCT clauses.
- The query may not contain any AGGREGATE functions.
  In both the cases, the zookeeper quorum should be passed to the PhoenixHBaseLoader as an argument to the constructor.

The Loadfunc makes best effort to map Phoenix Data Types to Pig datatype. You can have a look at org.apache.phoenix.pig.util.TypeUtil to see how each of Phoenix data type is mapped to Pig data type.

Example

Determine the number of users by a CLIENT ID

Ddl

```
CREATE TABLE HIRES( CLIENTID INTEGER NOT NULL, EMPID INTEGER NOT NULL, NAME VARCHAR CONSTRAINT pk PRIMARY KEY(CLIENTID,EMPID));
```

Pig Script

```
raw = LOAD 'hbase://table/HIRES USING org.apache.phoenix.pig.PhoenixHBaseLoader('localhost')';
grpd = GROUP raw BY CLIENTID;
cnt = FOREACH grpd GENERATE group AS CLIENT,COUNT(raw);
DUMP cnt;
```

Future Work

- Support for ARRAY data type.
- Usage of expressions within the SELECT clause when providing a full query.

> 原文: *http://phoenix.apache.org/pig_integration.html*

Map Reduce Integration

Phoenix Map Reduce

Phoenix provides support for retrieving and writing to Phoenix tables from within MapReduce jobs. The framework now provides custom InputFormat and OutputFormat classes PhoenixInputFormat , PhoenixOutputFormat.

PhoenixMapReduceUtil provides several utility methods to set the input and output configuration parameters to the job.

When a Phoenix table is the source for the Map Reduce job, we can provide a SELECT query or pass a table name and specific columns to import data . To retrieve data from the table within the mapper class, we need to have a class that implements DBWritable and pass it as an argument to PhoenixMapReduceUtil.**setInput** method. The custom DBWritable class provides implementation for readFields(ResultSet rs) that allows us to retrieve columns for each row. This custom DBWritable class will form the input value to the mapper class.

> *Note: The SELECT query must not perform any aggregation or use DISTINCT as these are not supported by our map-reduce integration.*

Similarly, when writing to a Phoenix table, we use the PhoenixMapReduceUtil.**setOutput** method to set the output table and the columns.

> *Note: Phoenix internally builds the UPSERT query for you .*

The output key and value class for the job should always be NullWritable and the custom DBWritable class that implements the write method .

Let’s dive into an example where we have a table, **STOCK** , that holds the master data of quarterly recordings in a double array for each year and we would like to find out the max price of each stock across all years. Let’s store the output to a **STOCK_STATS** table which is another Phoenix table.

> *Note , you can definitely have a job configured to read from hdfs and load into a phoenix table.*

a) *stock*

```
 CREATE TABLE IF NOT EXISTS STOCK (STOCK_NAME VARCHAR NOT NULL ,RECORDING_YEAR INTEGER NOT  NULL,
RECORDINGS_QUARTER DOUBLE array[] CONSTRAINT pk PRIMARY KEY (STOCK_NAME , RECORDING_YEAR));
```

b) *stock_stats*

```
CREATE TABLE IF NOT EXISTS STOCK_STATS (STOCK_NAME VARCHAR NOT NULL , MAX_RECORDING DOUBLE CONSTRAINT pk
PRIMARY KEY (STOCK_NAME));
```

Sample Data

```
    UPSERT into STOCK values ('AAPL',2009,ARRAY[85.88,91.04,88.5,90.3]);
    UPSERT into STOCK values ('AAPL',2008,ARRAY[199.27,200.26,192.55,194.84]);
    UPSERT into STOCK values ('AAPL',2007,ARRAY[86.29,86.58,81.90,83.80]);
    UPSERT into STOCK values ('CSCO',2009,ARRAY[16.41,17.00,16.25,16.96]);
    UPSERT into STOCK values ('CSCO',2008,ARRAY[27.00,27.30,26.21,26.54]);
    UPSERT into STOCK values ('CSCO',2007,ARRAY[27.46,27.98,27.33,27.73]);
    UPSERT into STOCK values ('CSCO',2006,ARRAY[17.21,17.49,17.18,17.45]);
    UPSERT into STOCK values ('GOOG',2009,ARRAY[308.60,321.82,305.50,321.32]);
    UPSERT into STOCK values ('GOOG',2008,ARRAY[692.87,697.37,677.73,685.19]);
    UPSERT into STOCK values ('GOOG',2007,ARRAY[466.00,476.66,461.11,467.59]);
    UPSERT into STOCK values ('GOOG',2006,ARRAY[422.52,435.67,418.22,435.23]);
    UPSERT into STOCK values ('MSFT',2009,ARRAY[19.53,20.40,19.37,20.33]);
    UPSERT into STOCK values ('MSFT',2008,ARRAY[35.79,35.96,35.00,35.22]);
    UPSERT into STOCK values ('MSFT',2007,ARRAY[29.91,30.25,29.40,29.86]);
    UPSERT into STOCK values ('MSFT',2006,ARRAY[26.25,27.00,26.10,26.84]);
    UPSERT into STOCK values ('YHOO',2009,ARRAY[12.17,12.85,12.12,12.85]);
    UPSERT into STOCK values ('YHOO',2008,ARRAY[23.80,24.15,23.60,23.72]);
    UPSERT into STOCK values ('YHOO',2007,ARRAY[25.85,26.26,25.26,25.61]);
    UPSERT into STOCK values ('YHOO',2006,ARRAY[39.69,41.22,38.79,40.91]);
```

Below is a simple job configuration

Job Configuration

```
final Configuration configuration = HBaseConfiguration.create();
final Job job = Job.getInstance(configuration, "phoenix-mr-job");

// We can either specify a selectQuery or ignore it when we would like to retrieve all the columns
final String selectQuery = "SELECT STOCK_NAME,RECORDING_YEAR,RECORDINGS_QUARTER FROM STOCK ";

// StockWritable is the DBWritable class that enables us to process the Result of the above query
PhoenixMapReduceUtil.setInput(job, StockWritable.class, "STOCK",  selectQuery);

// Set the target Phoenix table and the columns
PhoenixMapReduceUtil.setOutput(job, "STOCK_STATS", "STOCK_NAME,MAX_RECORDING");

job.setMapperClass(StockMapper.class);
job.setReducerClass(StockReducer.class);
job.setOutputFormatClass(PhoenixOutputFormat.class);

job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(DoubleWritable.class);
job.setOutputKeyClass(NullWritable.class);
job.setOutputValueClass(StockWritable.class);
TableMapReduceUtil.addDependencyJars(job);
job.waitForCompletion(true);
```

StockWritable

```
public class StockWritable implements DBWritable,Writable {

    private String stockName;

    private int year;

    private double[] recordings;

    private double maxPrice;

    @Override
    public void readFields(DataInput input) throws IOException {

    }

    @Override
    public void write(DataOutput output) throws IOException {

    }

    @Override
    public void readFields(ResultSet rs) throws SQLException {
       stockName = rs.getString("STOCK_NAME");
       year = rs.getInt("RECORDING_YEAR");
       final Array recordingsArray = rs.getArray("RECORDINGS_QUARTER");
       recordings = (double[])recordingsArray.getArray();
    }

    @Override
    public void write(PreparedStatement pstmt) throws SQLException {
       pstmt.setString(1, stockName);
       pstmt.setDouble(2, maxPrice);
    }

    // getters / setters for the fields
     ...
     ...
```

Stock Mapper

```
 public static class StockMapper extends Mapper<NullWritable, StockWritable, Text , DoubleWritable> {

    private Text stock = new Text();
    private DoubleWritable price = new DoubleWritable ();

    @Override
    protected void map(NullWritable key, StockWritable stockWritable, Context context) throws IOException,
InterruptedException {
       double[] recordings = stockWritable.getRecordings();
       final String stockName = stockWritable.getStockName();
       double maxPrice = Double.MIN_VALUE;
       for(double recording : recordings) {
         if(maxPrice < recording) {
          maxPrice = recording;
```

```
                }
            }
            stock.set(stockName);
            price.set(maxPrice);
            context.write(stock,price);
        }

}
```

Stock Reducer

```
 public static class StockReducer extends Reducer<Text, DoubleWritable, NullWritable , StockWritable> {

    @Override
    protected void reduce(Text key, Iterable<DoubleWritable> recordings, Context context) throws IOException,
InterruptedException {
        double maxPrice = Double.MIN_VALUE;
        for(DoubleWritable recording : recordings) {
          if(maxPrice < recording.get()) {
            maxPrice = recording.get();
          }
        }
         final StockWritable stock = new StockWritable();
         stock.setStockName(key.toString());
        stock.setMaxPrice(maxPrice);
         context.write(NullWritable.get(),stock);
    }

}
```

Packaging & Running

- Ensure phoenix-[version]-client.jar is in the classpath of your Map Reduce job jar.
- To run the job, use the **hadoop jar** command with the necessary arguments.

> 原文: *http://phoenix.apache.org/phoenix_mr.html*

Apache Flume Plugin

Apache Flume Plugin

The plugin enables us to reliably and efficiently stream large amounts of data/logs onto HBase using the Phoenix API. The necessary configuration of the custom Phoenix sink and the Event Serializer has to be configured in the Flume configuration file for the Agent. Currently, the only supported Event serializer is a RegexEventSerializer which primarily breaks the Flume Event body based on the regex specified in the configuration file.

Prerequisites:

- Phoenix v 3.0.0 SNAPSHOT +
- Flume 1.4.0 +

Installation & Setup:

- Download and build Phoenix v 0.3.0 SNAPSHOT
- Follow the instructions as specified here to build the project as the Flume plugin is still under beta
- Create a directory plugins.d within $FLUME_HOME directory. Within that, create a sub-directories phoenix-sink/lib
- Copy the generated phoenix-3.0.0-SNAPSHOT-client.jar onto $FLUME_HOME/plugins.d/phoenix-sink/lib

Configuration:

Property Name	Default	Description
type		org.apache.phoenix.flume.sink.PhoenixSink
batchSize	100	Default number of events per transaction
zookeeperQuorum		Zookeeper quorum of the HBase cluster
table		The name of the table in HBase to write to.
ddl		The CREATE TABLE query for the HBase table where the events will be upserted to. If specified, the query will be executed. Recommended to include the IF NOT EXISTS clause in the ddl.
serializer	regex	Event serializers for processing the Flume Event . Currently , only regex is supported.
serializer.regex	(.*)	The regular expression for parsing the event.
serializer.columns		The columns that will be extracted from the Flume event for inserting into HBase.
serializer.headers		Headers of the Flume Events that go as part of the UPSERT query. The data type for these

		columns are VARCHAR by default.
serializer.rowkeyType		A custom row key generator . Can be one of timestamp,date,uuid,random and nanotimestamp. This should be configured in cases where we need a custom row key value to be auto generated and set for the primary key column.

For an example configuration for ingesting Apache access logs onto Phoenix, see this property file. Here we are using UUID as a row key generator for the primary key.

Starting the agent:

```
$ bin/flume-ng agent -f conf/flume-conf.properties -c ./conf -n agent
```

Monitoring:

For monitoring the agent and the sink process , enable JMX via flume-env.sh($FLUME_HOME/conf/flume-env.sh) script. Ensure you have the following line uncommented.

```
JAVA_OPTS="-Xms1g -Xmx1g -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=3141 -
Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
```

> 原文: *http://phoenix.apache.org/flume.html*

Apache Kafka Plugin

Apache Kafka Plugin

The plugin enables us to reliably and efficiently stream large amounts of data/logs onto HBase using the Phoenix API.

Apache Kafka™ is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

So, at a high level, producers send messages over the network to the Kafka cluster which in turn serves them up to consumers like this:

We are providing **PhoenixConsumer** to recieves the messages from **Kafka Producer**.

Prerequisites:

- Phoenix 4.10.0+
- Kafka 0.9.0.0+

Installation & Setup:

Use our binary artifacts for Phoenix 4.10.0+ directly or download and build Phoenix yourself (see instructions here)

Phoenix Consumer for RegexEventSerializer Example:

Create a `kafka-consumer-regex.properties` file with below properties

```
serializer=regex
serializer.rowkeyType=uuid
serializer.regex=([^\,]*),([^\,]*),([^\,]*)
serializer.columns=c1,c2,c3

jdbcUrl=jdbc:phoenix:localhost
table=SAMPLE1
ddl=CREATE TABLE IF NOT EXISTS SAMPLE1(uid VARCHAR NOT NULL,c1 VARCHAR,c2 VARCHAR,c3 VARCHAR CONSTRAINT pk PRIMARY KEY(uid))

bootstrap.servers=localhost:9092
topics=topic1,topic2
poll.timeout.ms=100
```

Phoenix Consumer for JsonEventSerializer Example:

Create a `kafka-consumer-json.properties` file with below properties

```
serializer=json
serializer.rowkeyType=uuid
serializer.columns=c1,c2,c3

jdbcUrl=jdbc:phoenix:localhost
table=SAMPLE2
ddl=CREATE TABLE IF NOT EXISTS SAMPLE2(uid VARCHAR NOT NULL,c1 VARCHAR,c2 VARCHAR,c3 VARCHAR CONSTRAINT pk PRIMARY KEY(uid))

bootstrap.servers=localhost:9092
topics=topic1,topic2
poll.timeout.ms=100
```

Phoenix Consumer Execution Procedure:

Start the Kakfa Producer then send some messages

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topic1
```

Learn more about Apache Kafka here

Start the **PhoenixConsumer** using below command

```
HADOOP_CLASSPATH=$(hbase classpath):/path/to/hbase/conf hadoop jar phoenix-kafka-<version>-minimal.jar org.apache.phoenix.kafka.consumer.PhoenixConsumerTool --file /data/kafka-consumer.properties
```

The input file must be present on HDFS (not the local filesystem where the command is being run).

Configuration:

Property Name	Default	Description
bootstrap.servers		List of Kafka servers used to bootstrap connections to Kafka. This list should be in the form host1:port1,host2:port2,…
topics		List of topics to use as input for this connector. This list should be in the form topic1,topic2,…
poll.timeout.ms	100	Default poll timeout in millisec
batchSize	100	Default number of events per transaction
zookeeperQuorum		Zookeeper quorum of the HBase cluster
table		The name of the table in HBase to write to.
ddl		The CREATE TABLE query for the HBase table where the events will be upserted to. If specified, the query will be executed. Recommended to include the IF NOT EXISTS clause in the ddl.

serializer		Event serializers for processing the Kafka Message.This Plugin supports all Phoenix Flume Event Serializers. Like regex, json
serializer.regex	(.*)	The regular expression for parsing the message.
serializer.columns		The columns that will be extracted from the Flume event for inserting into HBase.
serializer.headers		Headers of the Flume Events that go as part of the UPSERT query. The data type for these columns are VARCHAR by default.
serializer.rowkeyType		A custom row key generator . Can be one of timestamp,date,uuid,random and nanotimestamp. This should be configured in cases where we need a custom row key value to be auto generated and set for the primary key column.

Note: This Plugin supports all Phoenix Flume Event Serializers.

RegexEventSerializer which primarily breaks the Kafka Message based on the regex specified in the configuration file.

JsonEventSerializer which primarily breaks the Kafka Message based on the schema specified in the configuration file.

> 原文: *http://phoenix.apache.org/kafka.html*

Python Driver

Python Driver for Phoenix

The Python Driver for Apache Phoenix implements the Python DB 2.0 API to access Phoenix via the Phoenix Query Server. The driver is tested with Python 2.7, 3.5, and 3.6. This code was originally called Python Phoenixdb and was graciously donated by its authors to the Apache Phoenix project.

All future development of the project is being done in Apache Phoenix.

Installation

Phoenix does not presently deploy the driver, so users must first build the driver and install it themselves:

```
$ cd python
$ pip install -r requirements.txt
$ python setup.py install
```

Deploying a binary release to central Python package-hosting services (e.g. installable via `pip`) will be done eventually. Presently, releases are in source-form only.

Examples

```python
import phoenixdb
import phoenixdb.cursor

database_url = 'http://localhost:8765/'
conn = phoenixdb.connect(database_url, autocommit=True)

cursor = conn.cursor()
cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, username VARCHAR)")
cursor.execute("UPSERT INTO users VALUES (?, ?)", (1, 'admin'))
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())

cursor = conn.cursor(cursor_factory=phoenixdb.cursor.DictCursor)
cursor.execute("SELECT * FROM users WHERE id=1")
print(cursor.fetchone()['USERNAME'])
```

Limitations

- The driver presently does not support Kerberos authentication PHOENIX-4688

Resources

- PHOENIX-4636 : Initial landing of the driver into Apache Phoenix.

> 原文: *http://phoenix.apache.org/python.html*