# Phoenix guide

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

## Quick Start

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

## SQL Support

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

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

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

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

# Phoenix in 15 minutes or less

***What is this new*** [***Phoenix***](http://phoenix.apache.org/index.html) ***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](http://phoenix.apache.org/performance.html) 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](http://files.meetup.com/1350427/IntelPhoenixHBaseMeetup.ppt) 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](http://phoenix.apache.org/issues.html) for us and/or join our [mailing list](http://phoenix.apache.org/mailing_list.html).

***Blah, blah, blah - I just want to get started!***  
Ok, great! Just follow our [install instructions](http://phoenix.apache.org/download.html#Installation):

* [download](http://phoenix.apache.org/download.html) 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](http://phoenix.apache.org/installation.html#SQL_Client) 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!

# Performance

Phoenix follows the philosophy of **bringing the computation to the data** by using:

* **coprocessors** to perform operations on the server-side thus minimizing client/server data transfer
* **custom filters** to prune data as close to the source as possible In addition, to minimize any startup costs, Phoenix uses native HBase APIs rather than going through the map/reduce framework.

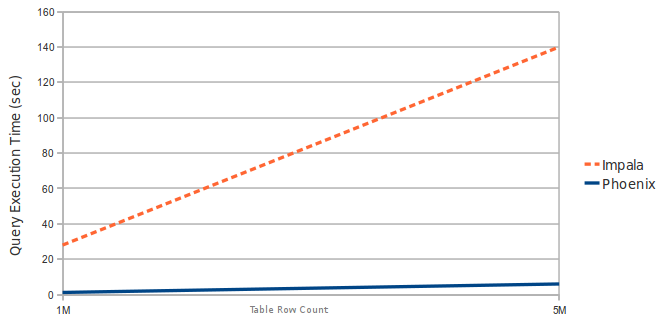
## Phoenix vs related products

Below are charts showing relative performance between Phoenix and some other related products.

### Phoenix vs Hive (running over HDFS and HBase)

Query: select count(1) from table over 10M and 100M rows. Data is 5 narrow columns. Number of Region Servers: 4 (HBase heap: 10GB, Processor: 6 cores @ 3.3GHz Xeon)

### Phoenix vs Impala (running over HBase)



Query: select count(1) from table over 1M and 5M rows. Data is 3 narrow columns. Number of Region Server: 1 (Virtual Machine, HBase heap: 2GB, Processor: 2 cores @ 3.3GHz Xeon)

## Latest Automated Performance Run

[Latest Automated Performance Run](http://phoenix-bin.github.io/client/performance/latest.htm) | [Automated Performance Runs History](http://phoenix-bin.github.io/client/performance/)

## Performance improvements in Phoenix 1.2

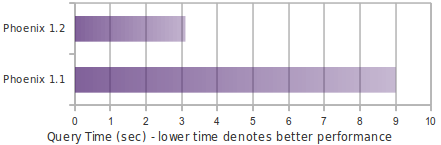
### Essential Column Family

Phoenix 1.2 query filter leverages [HBase Filter Essential Column Family]([http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/SingleColumnValueFilter.html#isFamilyEssential(byte[]](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/SingleColumnValueFilter.html#isFamilyEssential%28byte[])) feature which leads to improved performance when Phoenix query filters on data that is split in multiple column families (cf) by only loading essential cf. In second pass, all cf are are loaded as needed.

Consider the following schema in which data is split in two cf create table t (k varchar not null primary key, a.c1 integer, b.c2 varchar, b.c3 varchar, b.c4 varchar).

Running a query similar to the following shows significant performance when a subset of rows match filter select count(c2) from t where c1 = ?

Following chart shows query in-memory performance of running the above query with 10M rows on 4 region servers when 10% of the rows matches the filter. Note: cf-a is approx 8 bytes and cf-b is approx 400 bytes wide.



### Skip Scan

Skip Scan Filter leverages [SEEK\_NEXT\_USING\_HINT](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/Filter.ReturnCode.html#SEEK_NEXT_USING_HINT) of HBase Filter. It significantly improves point queries over key columns.

Consider the following schema in which data is split in two cf create table t (k varchar not null primary key, a.c1 integer, b.c2 varchar, b.c3 varchar).

Running a query similar to the following shows significant performance when a subset of rows match filter select count(c1) from t where k in (1% random k's)

Following chart shows query in-memory performance of running the above query with 10M rows on 4 region servers when 1% random keys over the entire range passed in query IN clause. Note: all varchar columns are approx 15 bytes.

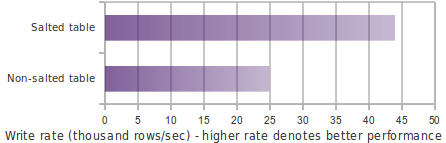
### Salting

Salting in Phoenix 1.2 leads to both improved read and write performance by adding an extra hash byte at start of key and pre-splitting data in number of regions. This eliminates hot-spotting of single or few regions servers. Read more about this feature [here](http://phoenix.apache.org/salted.html).

Consider the following schema

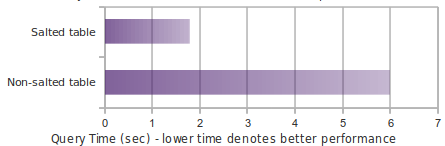
CREATE TABLE T (HOST CHAR(2) NOT NULL,DOMAIN VARCHAR NOT NULL, FEATURE VARCHAR NOT NULL,DATE DATE NOT NULL,USAGE.CORE BIGINT,USAGE.DB BIGINT,STATS.ACTIVE\_VISITOR INTEGER CONSTRAINT PK PRIMARY KEY (HOST, DOMAIN, FEATURE, DATE)) SALT\_BUCKETS = 4.

Following chart shows write performance with and without the use of Salting which splits table in 4 regions running on 4 region server cluster (Note: For optimal performance, number of salt buckets should match number of region servers).



Following chart shows in-memory query performance for 10M row table where host='NA' filter matches 3.3M rows

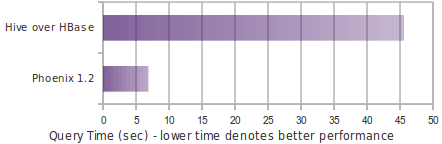
select count(1) from t where host='NA'



### Top-N

Following chart shows in-memory query time of running the Top-N query over 10M rows using Phoenix 1.2 and Hive over HBase

select core from t order by core desc limit 10



# 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

1. 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-*<version>*-client-**spark**.jar’ Note that for Phoenix versions < 4.7.0, you must use the ‘phoenix-*<version>*-client.jar’
2. Add the following 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

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

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](http://phoenix.apache.org/language/datatypes.html).

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](http://phoenix.apache.org/phoenix_mr.html) documentation.

### PageRank example

This example makes use of the Enron email data set, provided by the [Stanford Network Analysis Project](https://snap.stanford.edu/data/email-Enron.html), and executes the GraphX implementation of PageRank on it to find interesting entities. It then saves the results back to Phoenix.

1. Download and extract the file [enron.csv.gz](https://github.com/jmahonin/spark-graphx-phoenix/blob/master/enron.csv.gz?raw=true)
2. Create the necessary Phoenix schema
3. CREATE TABLE EMAIL\_ENRON(MAIL\_FROM BIGINT NOT NULL, MAIL\_TO BIGINT NOT NULL CONSTRAINT pk PRIMARY KEY(MAIL\_FROM, MAIL\_TO));
4. CREATE TABLE EMAIL\_ENRON\_PAGERANK(ID BIGINT NOT NULL, RANK DOUBLE CONSTRAINT pk PRIMARY KEY(ID));
5. Load the email data into Phoenix (assuming localhost for Zookeeper Quroum URL)
6. gunzip /tmp/enron.csv.gz
7. cd /path/to/phoenix/bin
8. ./psql.py -t EMAIL\_ENRON localhost /tmp/enron.csv
9. In spark-shell, with the phoenix-client in the Spark driver classpath, run the following:
10. import org.apache.spark.graphx.\_
11. import org.apache.phoenix.spark.\_
12. val rdd = sc.phoenixTableAsRDD("EMAIL\_ENRON", Seq("MAIL\_FROM", "MAIL\_TO"), zkUrl=Some("localhost")) // load from phoenix
13. val rawEdges = rdd.map{ e => (e("MAIL\_FROM").asInstanceOf[VertexId], e("MAIL\_TO").asInstanceOf[VertexId]) } // map to vertexids
14. val graph = Graph.fromEdgeTuples(rawEdges, 1.0) // create a graph
15. val pr = graph.pageRank(0.001) // run pagerank
16. pr.vertices.saveToPhoenix("EMAIL\_ENRON\_PAGERANK", Seq("ID", "RANK"), zkUrl = Some("localhost")) // save to phoenix
17. Query the top ranked entities in SQL
18. SELECT \* FROM EMAIL\_ENRON\_PAGERANK ORDER BY RANK DESC LIMIT 5;
19. +------------------------------------------+------------------------------------------+
20. | ID | RANK |
21. +------------------------------------------+------------------------------------------+
22. | 5038 | 497.2989872977676 |
23. | 273 | 117.18141799210386 |
24. | 140 | 108.63091596789913 |
25. | 458 | 107.2728800448782 |
26. | 588 | 106.11840798585399 |
27. +------------------------------------------+------------------------------------------+