APACHE Phoenix
"We put the SQL back in NoSQL"

Apache Phoenix

Transforming HBase into a Relational Database

@ApachePhoenix
http://phoenix.apache.org/

V5

Agenda

- What is Apache Phoenix (and Apache HBase)?
- A Deeper Look
  - Views and Multi-Tenancy
  - Secondary Indexes
  - Joins
  - Query Optimizations
- Roadmap
- Q&A

What is Apache Phoenix (and Apache HBase)?

APACHE Phoenix

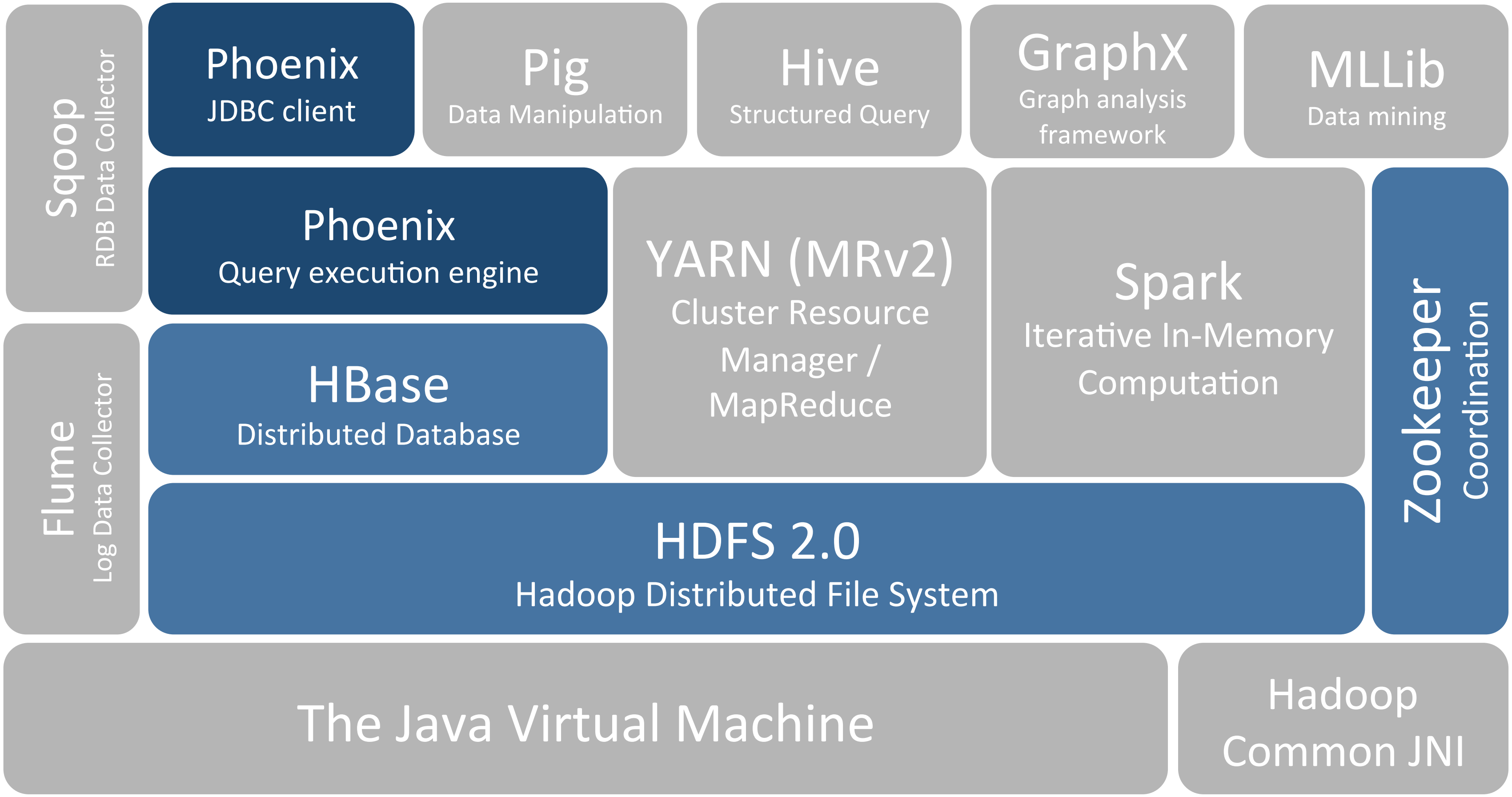
"We put the SQL back in NoSQL"

What is Apache Phoenix?

- A relational database layer for Apache HBase
  - Query engine
    - Transforms SQL queries into native HBase API calls
    - Pushes as much work as possible onto the cluster for parallel execution
  - Metadata repository
    - Typed access to data stored in HBase tables
  - A JDBC driver
- A new Apache Software Foundation project
  - Originally developed at Salesforce
  - Now a top-level project at the ASF
  - A growing community with momentum

APACHE Phoenix
"We put the SQL back in NoSQL"

Where Does Phoenix Fit In?



Sqoop
RDB Data Collector

Phoenix
JDBC client

Pig
Data Manipulation

Hive
Structured Query

GraphX
Graph analysis framework

MLLib
Data mining

Phoenix
Query execution engine

YARN (MRv2)
Cluster Resource Manager / MapReduce

Spark
Iterative In-Memory Computation

Zookeeper
Coordination

Flume
Log Data Collector

HBase
Distributed Database

HDFS 2.0
Hadoop Distributed File System

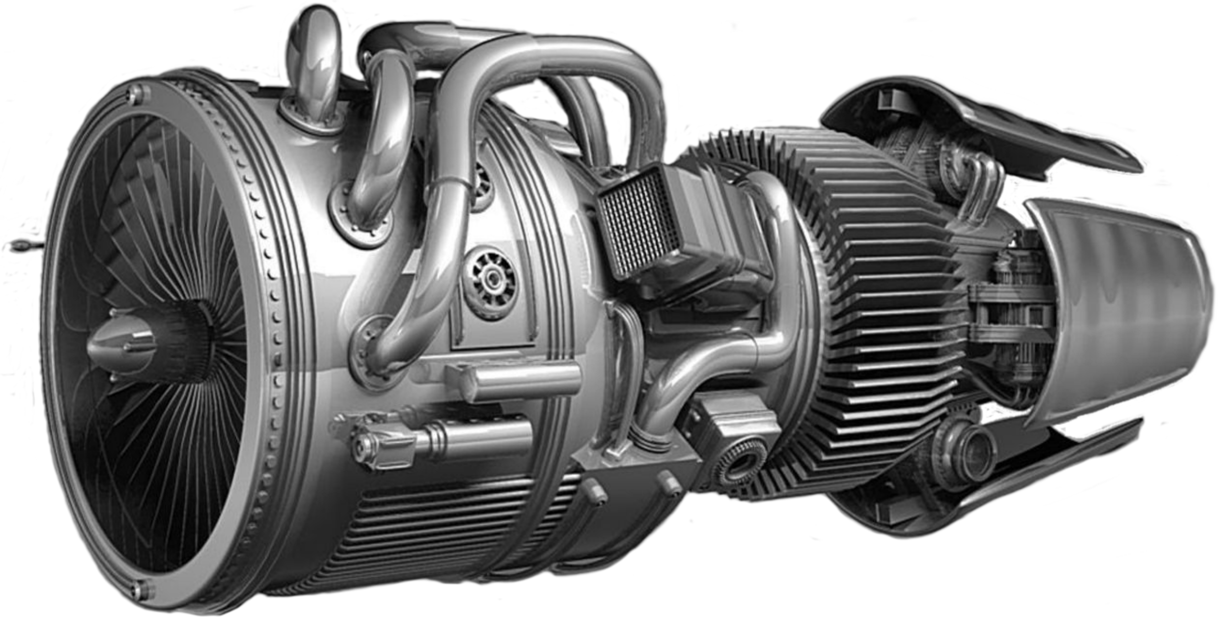
The Java Virtual Machine

Hadoop
Common JNI

APACHE Phoenix
"We put the SQL back in NoSQL"

What is Apache HBase?

- A high performance horizontally scalable datastore engine for Big Data, suitable as the store of record for mission critical data



APACHE HBASE

APACHE Phoenix
"We put the SQL back in NoSQL"

What is Apache HBase?

- An emerging platform for scale out relational datastores

APACHE Phoenix
"We put the SQL back in NoSQL"

splice MACHINE

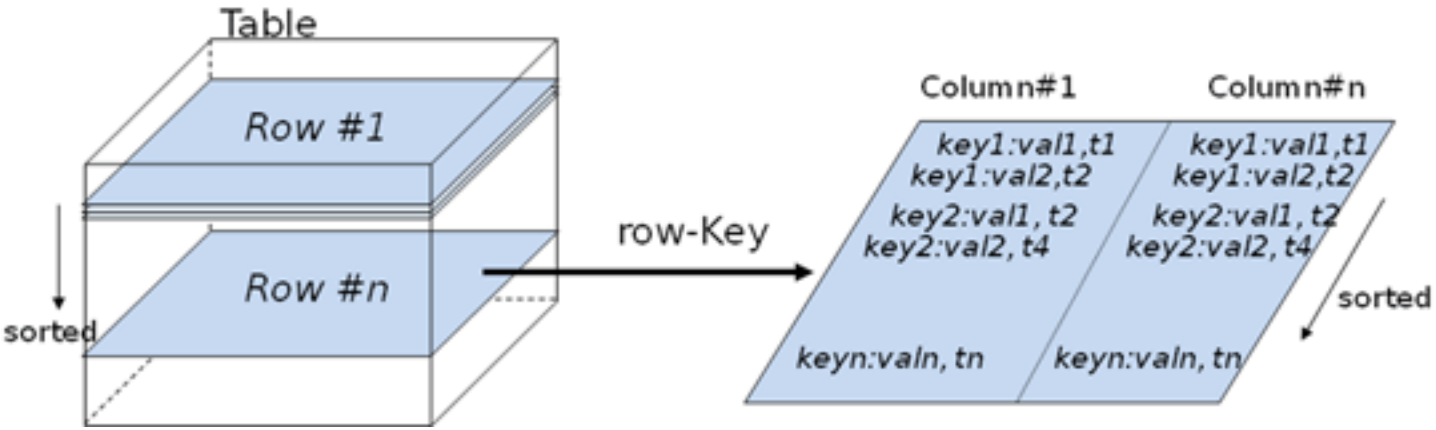
Trafodion

APACHE HBASE

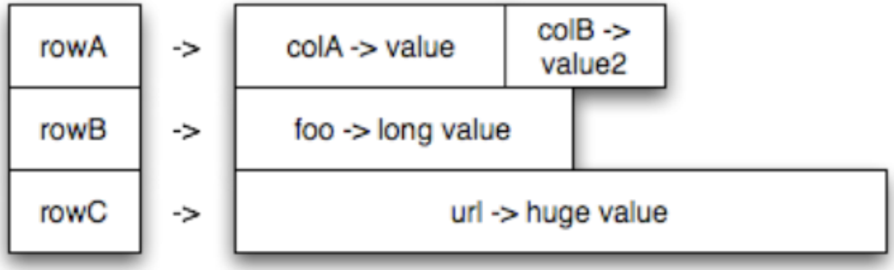
APACHE Phoenix
"We put the SQL back in NoSQL"

The HBase Data Model

- Tablespaces



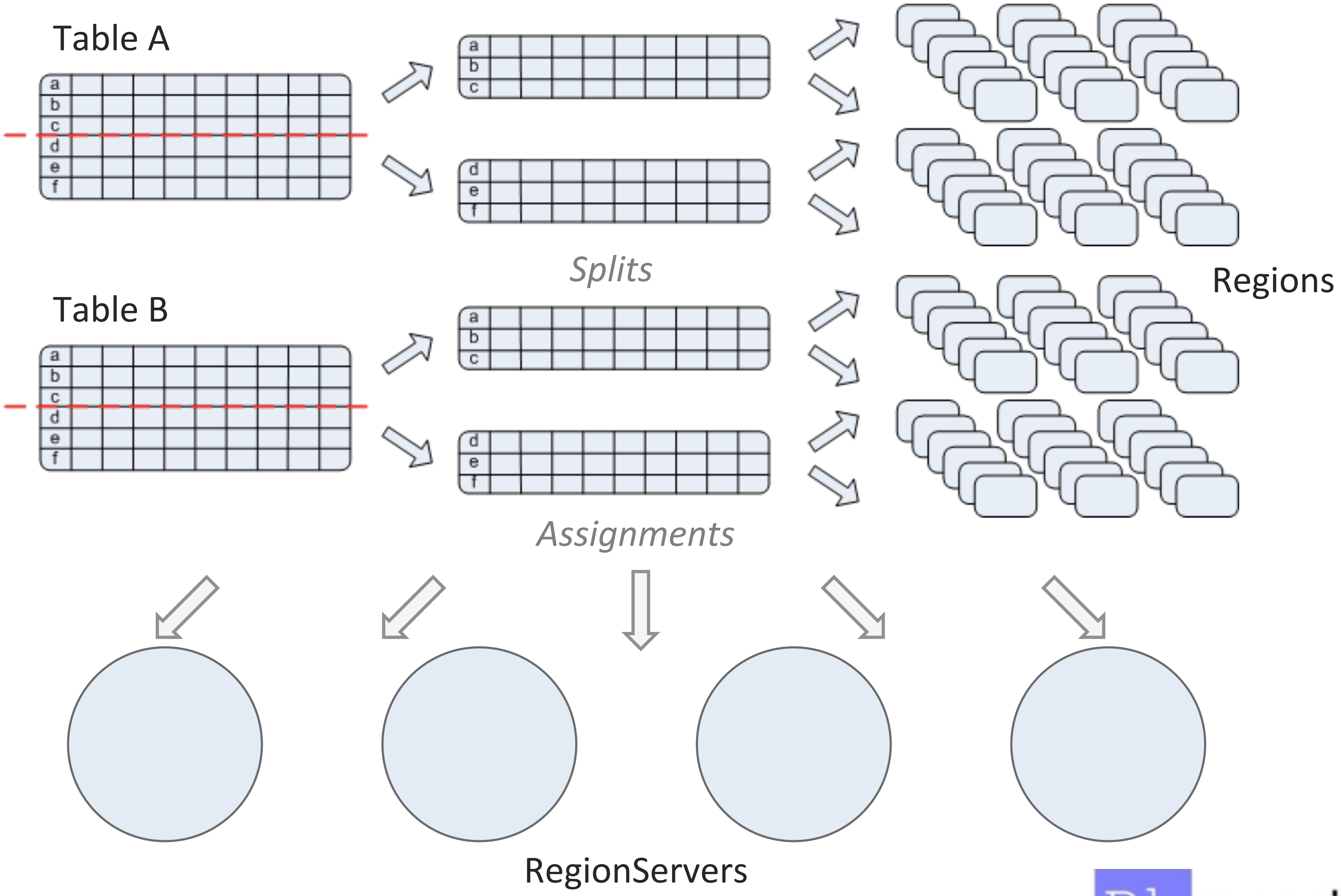
- Not like a spreadsheet, a “sparse, consistent, distributed, multi-dimensional, sorted map”


rowA -> colA -> value | colB -> value2

rowB -> foo -> long value

rowC -> url -> huge value

How HBase Achieves Scalability



APACHE Phoenix "We put the SQL back in NoSQL"

How is HBase Different from a RDBMS?

	RDBMS	HBase
Data layout	Row oriented	Column oriented
Transactions	Multi-row ACID	Single row or adjacent row groups only
Query language	SQL	None (API access)
Joins	Yes	No
Indexes	On arbitrary columns	Single row index only
Max data size	Terabytes	Petabytes*
R/W throughput limits	1000s of operations per second	Millions of operations per second*

* - No architectural upper bound on data size or aggregate throughput

APACHE Phoenix
"We put the SQL back in NoSQL"

SQL: In and Out Of Fashion

- 1969: CODASYL (network database)
- 1979: First commercial SQL RDBMs
- 1990: Transaction processing on SQL now popular
- 1993: Multidimensional databases
- 1996: Enterprise Data Warehouses
- 2006: Hadoop and other “big data” technologies
- 2008: NoSQL
- 2011: SQL on Hadoop
- 2014: Interactive analytics on Hadoop and NoSQL with SQL

- Why?

From “SQL On Everything, In Memory” by Julian Hyde, Strata NYC 2014

APACHE Phoenix
"We put the SQL back in NoSQL"

SQL: In and Out Of Fashion

- Implementing structured queries well is hard
  - Systems cannot just “run the query” as written
  - Relational systems require the algebraic operators, a query planner, an optimizer, metadata, statistics, etc.
- … but the result is very useful to non-technical users
  - Dumb queries (e.g. tool generated) can still get high performance
  - Adding new algorithms (e.g. a better join) or reorganizations of physical data layouts or migrations from one data store to another are transparent
- The challenge today is blending the scale and performance of NoSQL with the ease of use of SQL

Inspired by “SQL On Everything, In Memory” by Julian Hyde, Strata NYC 2014

Phoenix Puts the SQL Back in NoSQL

- A complete relational system
- Reintroduces the familiar declarative SQL interface to data (DDL, DML, etc.) with additional benefits
  - Read only views on existing HBase data
  - Dynamic columns extend schema at runtime
  - Schema is versioned – for free by HBase – allowing flashback queries using prior versions of metadata
- Reintroduces typed data and query optimizations possible with it
- Secondary indexes, query optimization, statistics, …
- Integrates the scalable HBase data storage platform as just another JDBC data source

APACHE Phoenix

"We put the SQL back in NoSQL"

Phoenix Puts the SQL Back in NoSQL

	Supported?
CREATE / DROP / ALTER TABLE	Yes
UPSERT / DELETE	Yes
SELECT	Yes
WHERE / HAVING	Yes
GROUP BY / ORDER BY	Yes
LIMIT	Yes
JOIN	Yes, with limitations
Views	Yes
Secondary Indexes	Yes
Transactions	Not yet*

* - PHOENIX-400 Transaction support
https://issues.apache.org/jira/browse/PHOENIX-400

APACHE Phoenix
"We put the SQL back in NoSQL"

Phoenix Puts the SQL Back in NoSQL

- Accessing HBase data with Phoenix can be substantially easier than direct HBase API use

```
SELECT * FROM foo WHERE bar > 30
```

Versus

```
HTable t = new HTable("foo");
RegionScanner s = t.getScanner(new Scan(…,
      new ValueFilter(CompareOp.GT,
            new CustomTypedComparator(30)), …));
while ((Result r = s.next()) != null) {
    // blah blah blah Java Java Java
}
s.close();
t.close();
```

Your BI tool probably can't do this

(And we didn't include error handling…)

APACHE Phoenix
"We put the SQL back in NoSQL"

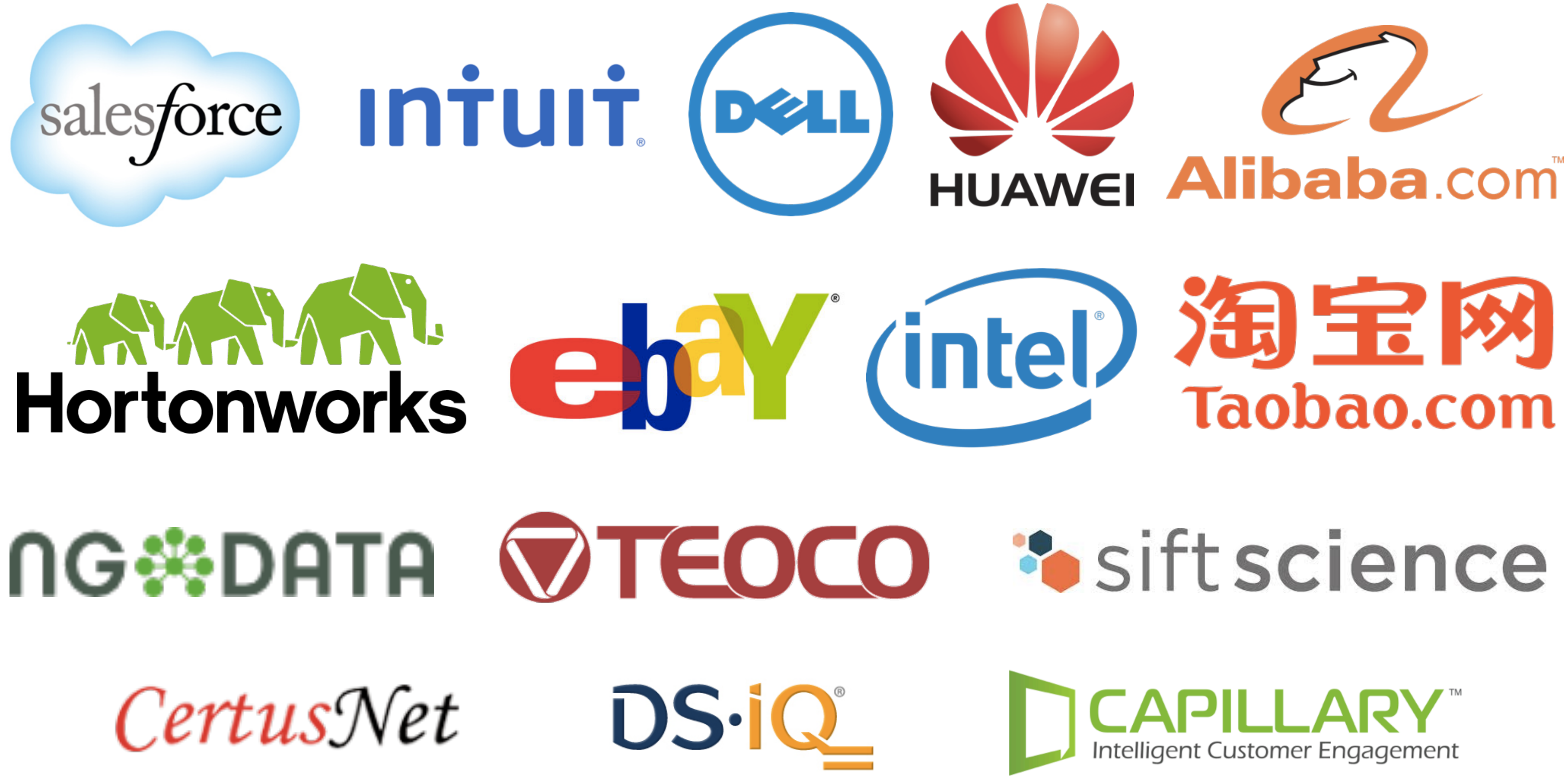
Phoenix Puts the SQL Back in NoSQL

- Accessing HBase data with Phoenix can be substantially faster than direct HBase API use
  - Phoenix parallelizes queries based on stats; HBase does not know how to chunk queries beyond scanning an entire region
  - Phoenix pushes processing to the server - most "by hand" API accesses do not use coprocessors
    - A huge difference for aggregation queries
  - Phoenix supports and uses secondary indexes
  - Phoenix uses "every trick in the book" based on various factors: the HBase version, metadata and query, reverse scans, small scans, skip scans, etc.

APACHE Phoenix

"We put the SQL back in NoSQL"

Who Uses Apache Phoenix?



And more ….

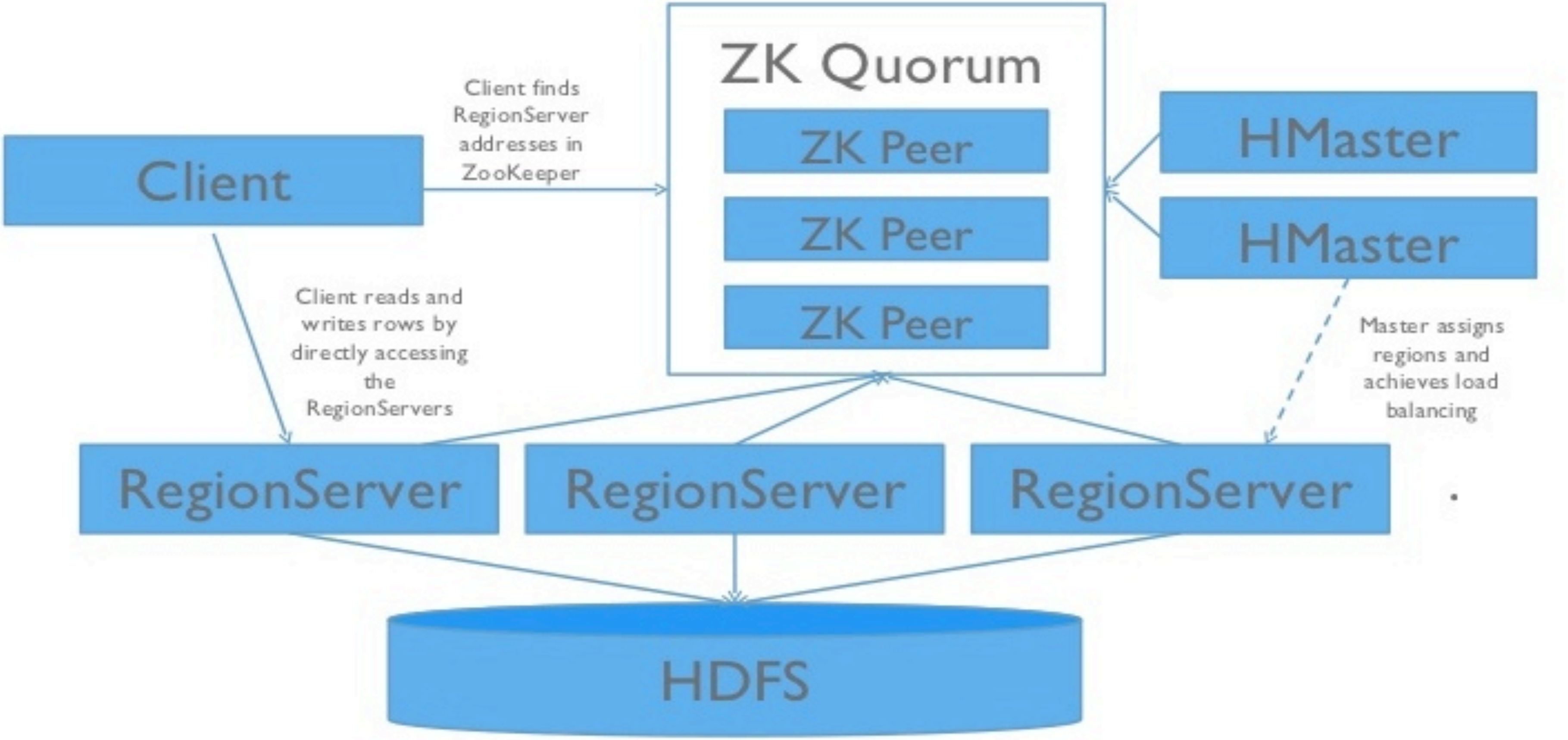
Salesforce Phoenix Use Cases

- Entity history
- System of Record
  - High scale, secure, non-transactional store for new use cases
- "Custom Objects" SQL-like public facing APIs
  - Custom objects are the user's business objects
  - We are adding a new complimentary implementation with a Phoenix+HBase back end for big data use cases

APACHE Phoenix
"We put the SQL back in NoSQL"

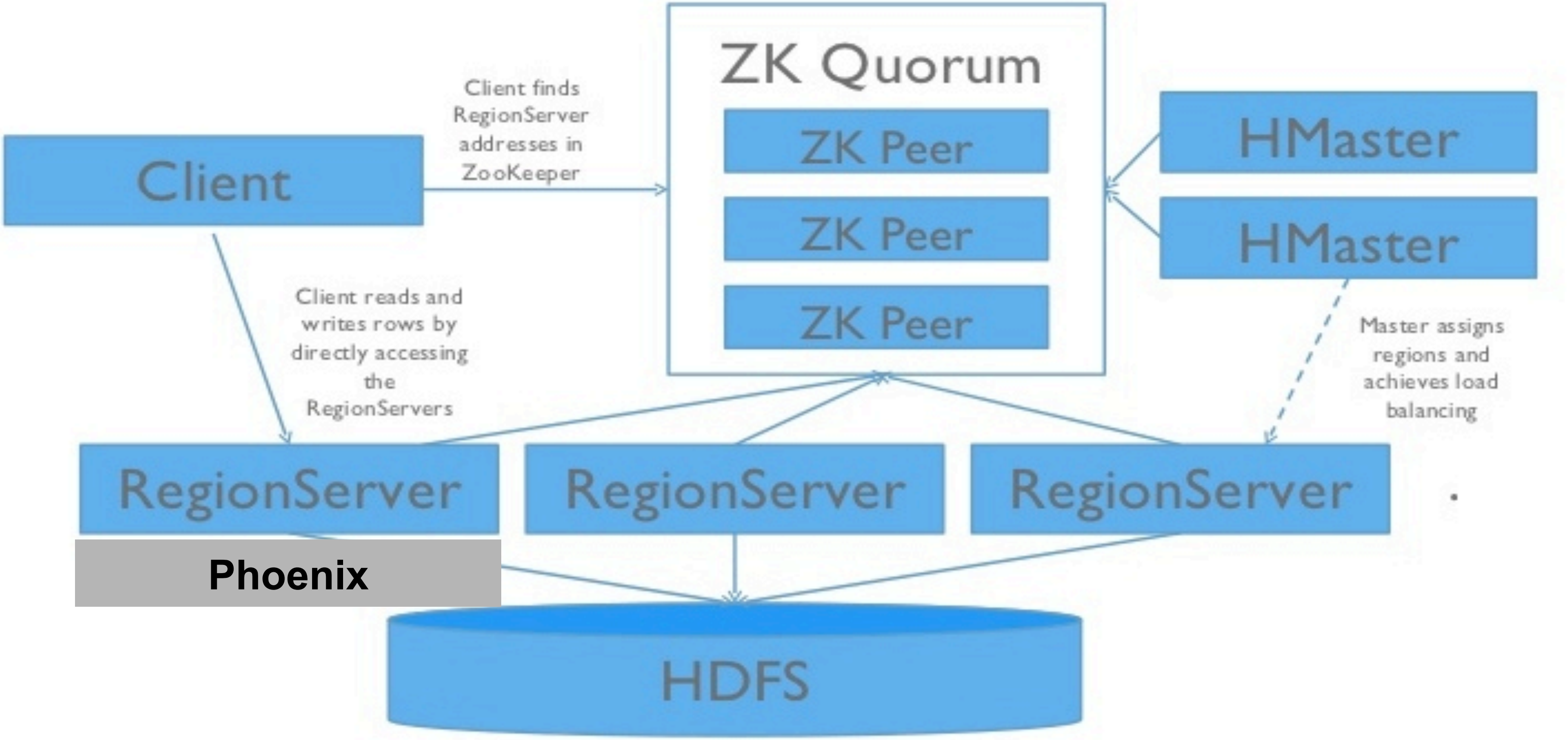
A Deeper Look

Phoenix + HBase Architecture


ZK Quorum
ZK Peer
ZK Peer
ZK Peer
Client
Client finds RegionServer addresses in ZooKeeper
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
HDFS

APACHE Phoenix
"We put the SQL back in NoSQL"

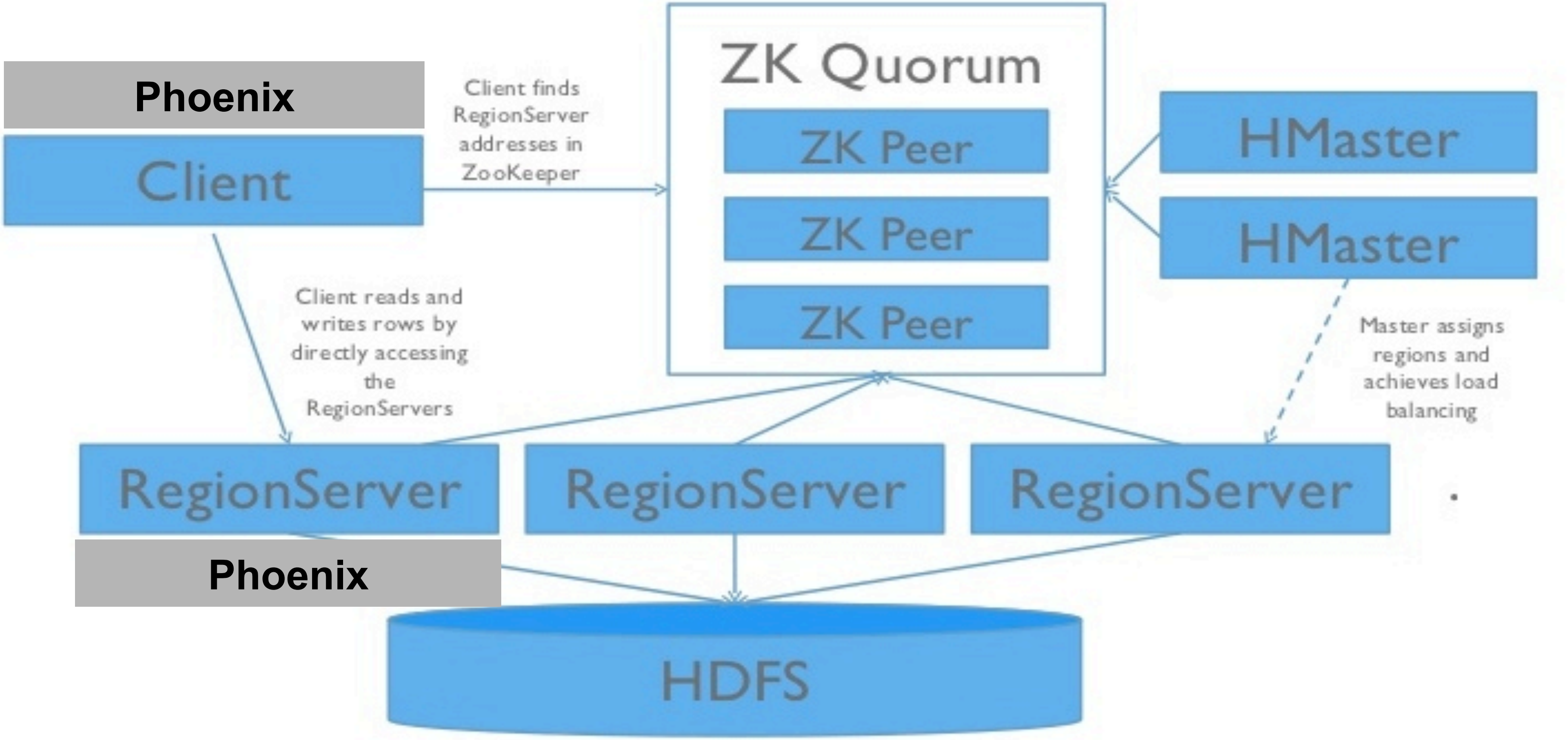
Phoenix + HBase Architecture


ZK Quorum
ZK Peer
ZK Peer
ZK Peer
Client
Client finds RegionServer addresses in ZooKeeper
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
Phoenix
HDFS

- The Phoenix jar is installed on the HBase RegionServer classpath

APACHE Phoenix
"We put the SQL back in NoSQL"

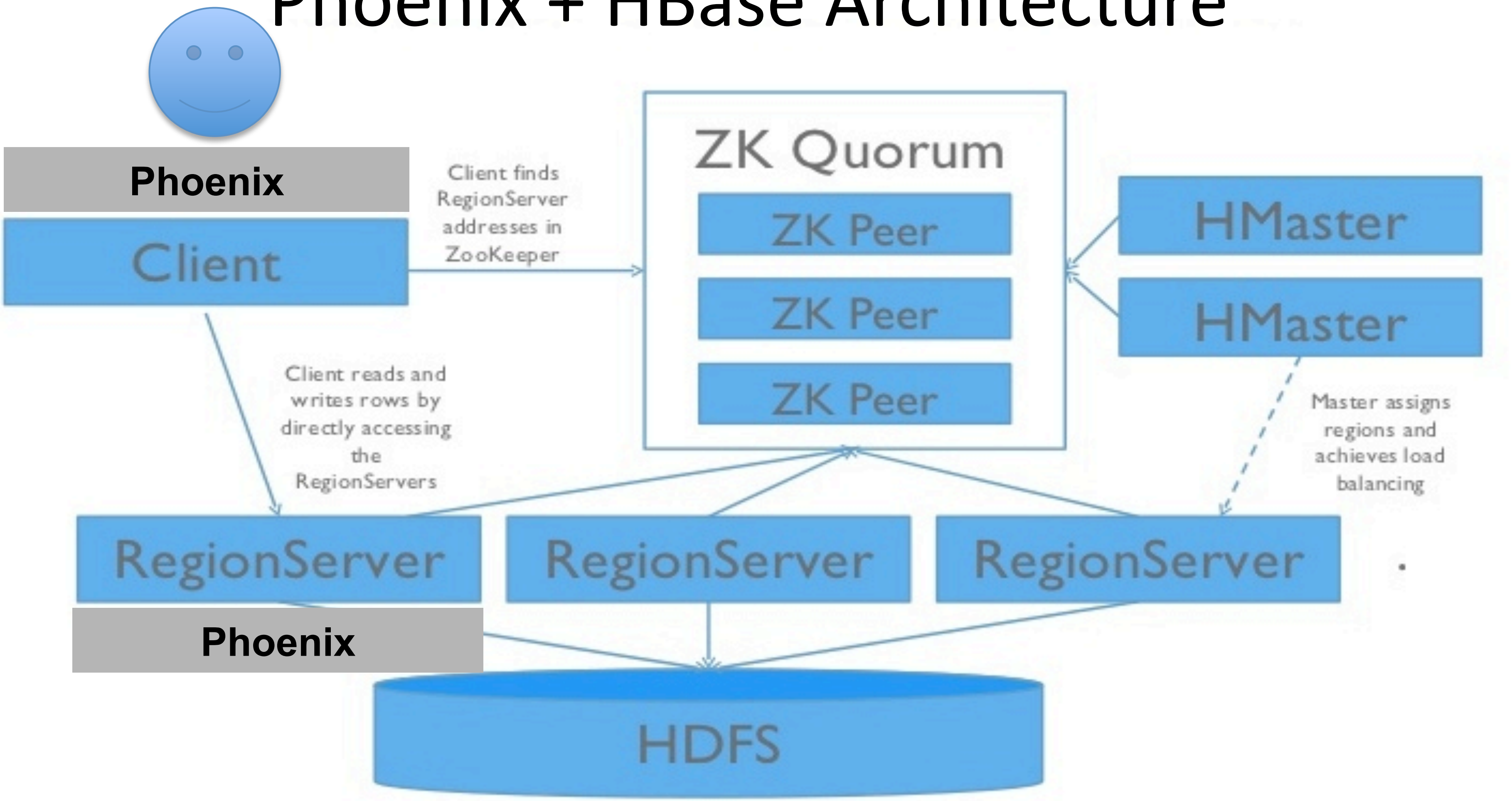
Phoenix + HBase Architecture


Phoenix
Client
Client finds RegionServer addresses in ZooKeeper
ZK Quorum
ZK Peer
ZK Peer
ZK Peer
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
Phoenix
HDFS

- The Phoenix jar is installed on the HBase RegionServer classpath
- The Phoenix JDBC driver is installed on the client

APACHE Phoenix
"We put the SQL back in NoSQL"

Phoenix + HBase Architecture

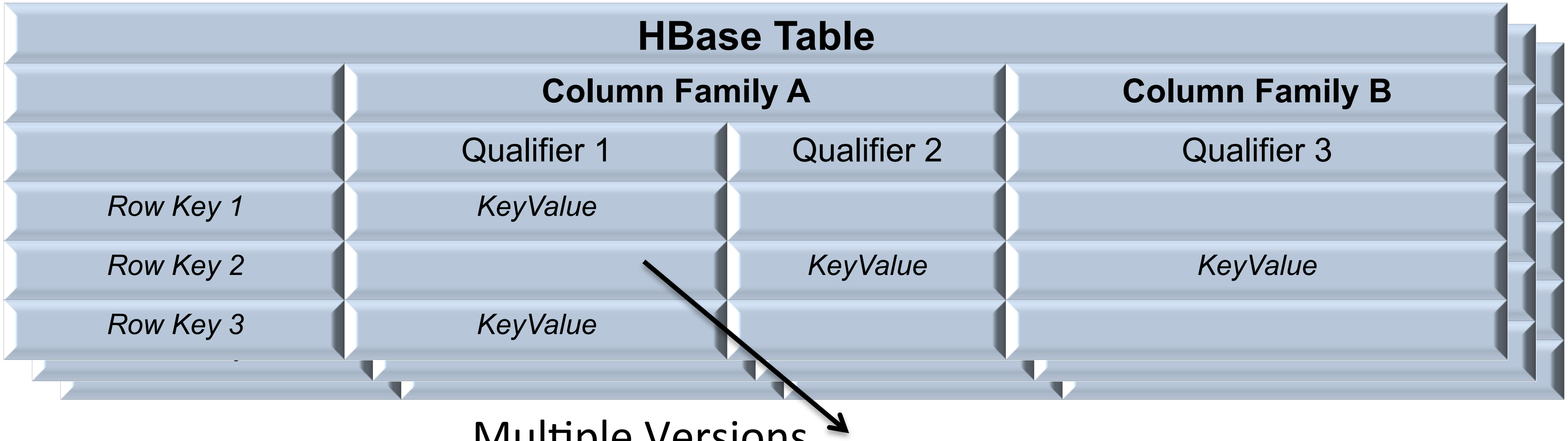

Phoenix
Client
Client finds RegionServer addresses in ZooKeeper
ZK Quorum
ZK Peer
ZK Peer
ZK Peer
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
Phoenix
HDFS

- The Phoenix jar is installed on the HBase RegionServer classpath
- The Phoenix JDBC driver is installed on the client
- The application speaks SQL to HBase

APACHE Phoenix
"We put the SQL back in NoSQL"

Phoenix Data Model

- Phoenix maps the HBase data model to the relational world

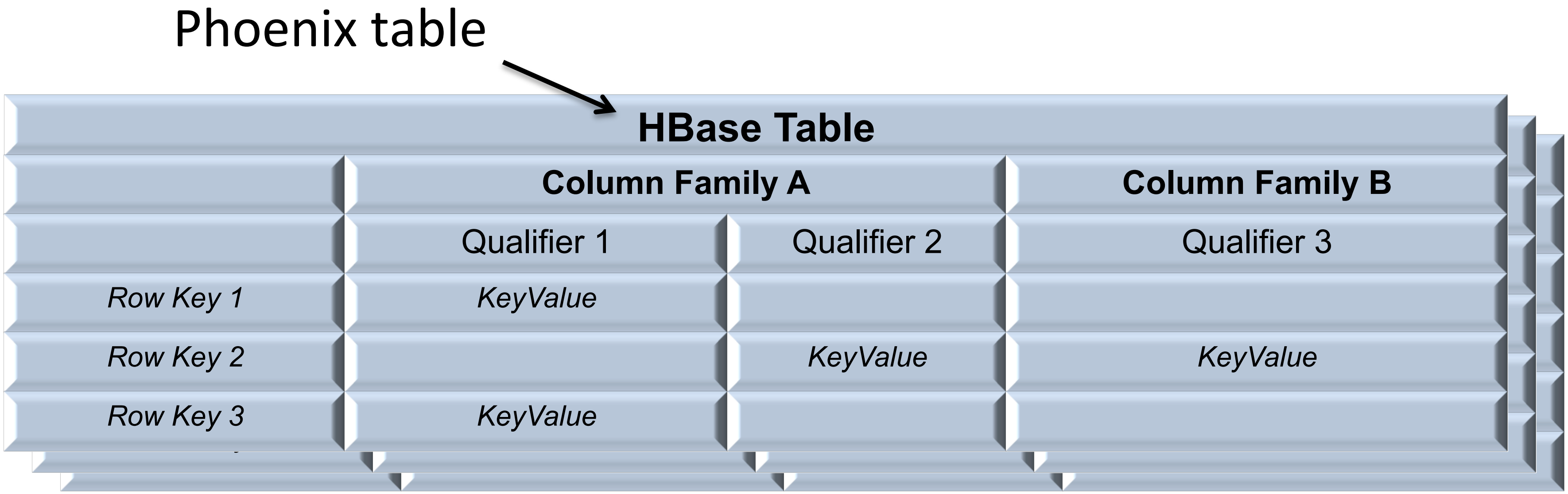


HBase Table			
	Column Family A		**Column Family B**
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	*KeyValue*		
Row Key 2		*KeyValue*	*KeyValue*
Row Key 3	*KeyValue*		

Multiple Versions

Remember: “*sparse, consistent, distributed, multi-dimensional, sorted map*”

Phoenix Data Model

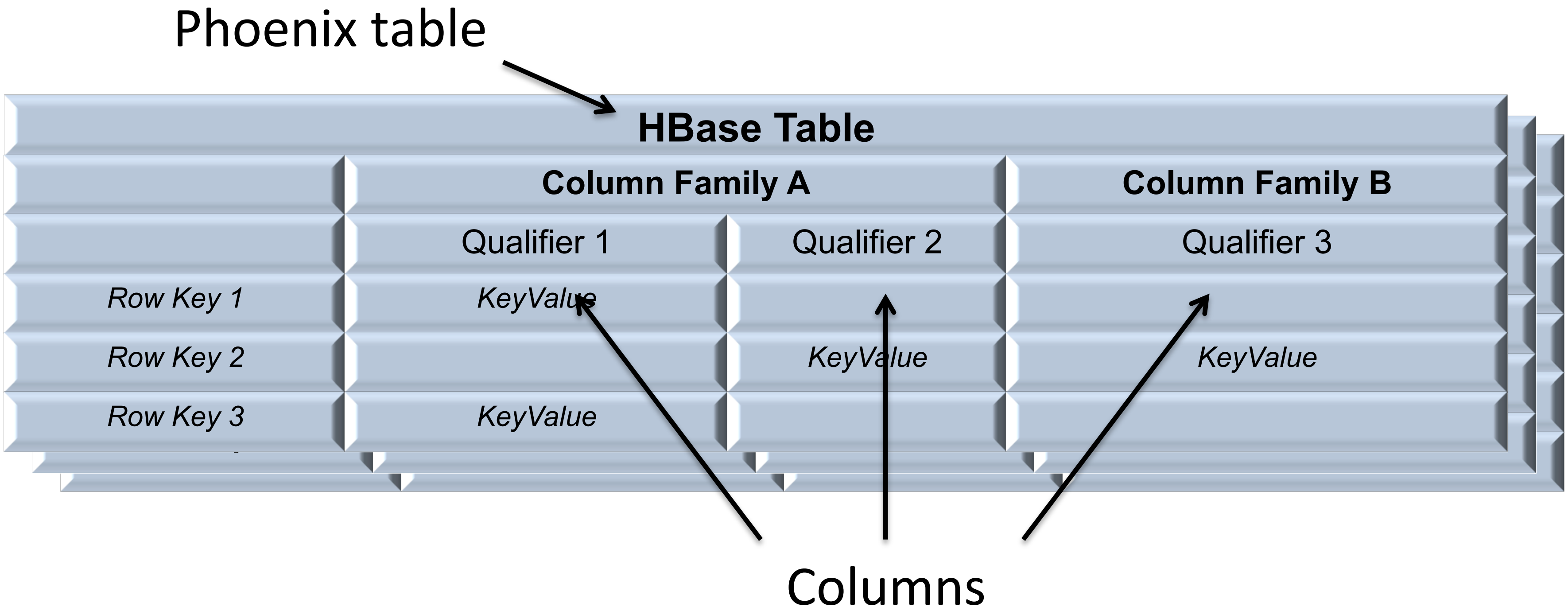


Phoenix table

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	*KeyValue*		
Row Key 2		*KeyValue*	*KeyValue*
Row Key 3	*KeyValue*		

APACHE Phoenix
"We put the SQL back in NoSQL"

Phoenix Data Model



Phoenix table

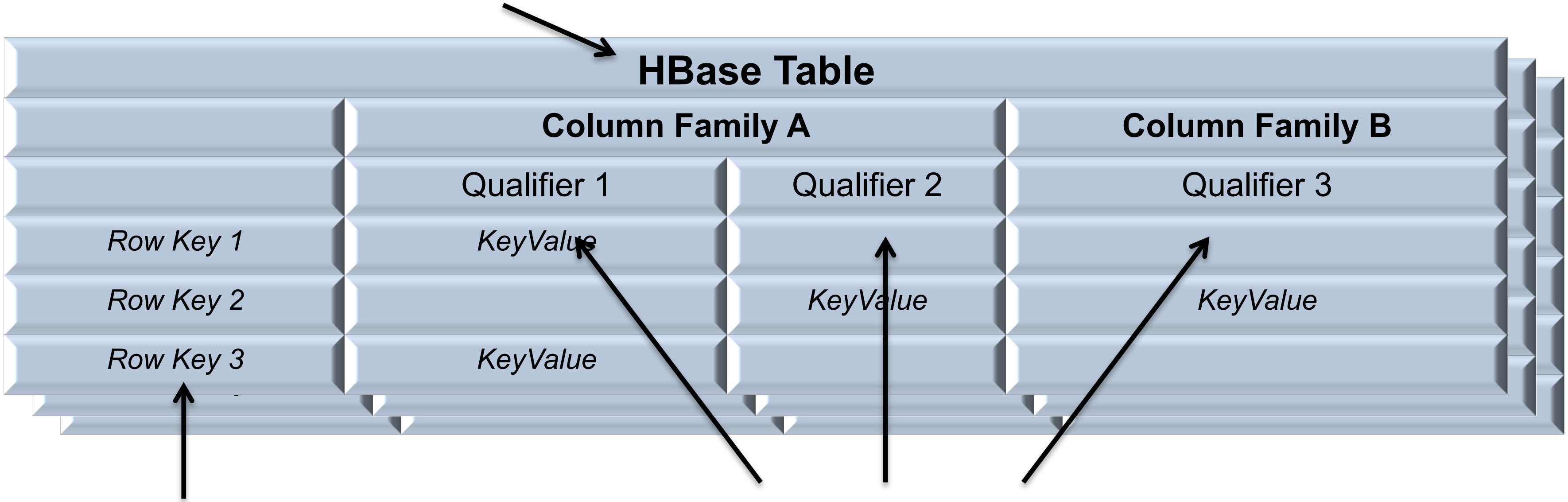
HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	*KeyValue*		
Row Key 2		*KeyValue*	*KeyValue*
Row Key 3	*KeyValue*		

Columns

APACHE Phoenix
"We put the SQL back in NoSQL"

Phoenix Data Model

Phoenix table



HBase Table			
	Column Family A		**Column Family B**
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	*KeyValue*		
Row Key 2		*KeyValue*	*KeyValue*
Row Key 3	*KeyValue*		

Primary Key Constraint

Columns

APACHE Phoenix
"We put the SQL back in NoSQL"

Data Model - Example

- Consider a table containing metrics data for servers with a schema like this:

SERVER METRICS		
HOST	VARCHAR	**Row Key**
DATE	DATE	
RESPONSE_TIME	INTEGER	**Columns**
GC_TIME	INTEGER	
CPU_TIME	INTEGER	
IO_TIME	INTEGER	

Data Model - Example

- The DDL command would look like:

```
CREATE TABLE SERVER_METRICS (
    HOST                VARCHAR         NOT NULL,
    DATE                DATE            NOT NULL,
    RESPONSE_TIME       INTEGER,
    GC_TIME             INTEGER,
    CPU_TIME            INTEGER,
    IO_TIME             INTEGER,
    CONSTRAINT pk PRIMARY KEY (HOST, DATE))
```

Data Model - Example

- And in the HBase table the data would be laid out like:

SERVER METRICS			
HOST + DATE		RESPONSE_TIME	GC_TIME
SF1	1396743589	1234	
SF1	1396743589		8012
…			
SF3	1396002345	2345	
SF3	1396002345		2340
SF7	1396552341	5002	1234
…			

Row Key

Columns

Views

- Updatable views
  - Views created using only simple equality expressions in the WHERE clause are updatable
- Read-only Views
  - Using more complex WHERE clauses in the view definition will result in read only views
  - Native HBase tables can be mapped with read-only views
- Multi-Tenant Views
  - Tenant-specific views may only be created using tenant-specific connections (more on this later)

Views

- Single table only, views over multiple joined tables are not supported (yet)
- Once a view is created the underlying table cannot be dropped until all views are dropped
- Creating indexes over views is supported
- Any index data for a view will be deleted if it is dropped

Mapping Existing HBase Tables

- Phoenix supports *read only* access to existing HBase tables
  - Create a Phoenix table using CREATE TABLE
  - Or create a view using CREATE VIEW
  - Use appropriate quoting for mixed case HBase table and native column names
- NOTE: An empty cell will be inserted for each row in the native table to enforce primary key constraints
- NOTE: Serialized bytes in the table must match the expected Phoenix type serializations
  - See http://phoenix.apache.org/language/datatypes.html

Dynamic Columns

- Extend schema during query
- A subset of columns may be specified in the CREATE TABLE DDL while the remainder can be optionally surfaced at query time
- Especially useful for views mapped over native HBase tables

```
CREATE TABLE “t” (
        K                       VARCHAR         PRIMARY KEY,
        “f1”.”col1”             VARCHAR);

SELECT * FROM “t” (“f1”.”col2” VARCHAR);
```

Multi-Tenancy

- Phoenix can provide multi-tenant isolation via a combination of multi-tenant tables and tenant-specific connections
  - Tenant-specific connections only access data that belongs to the tenant
  - Tenants can create tenant-specific views and add their own columns
- Multi-Tenant Tables
  - Declare these using the `MULTI_TENANT=true` DDL property
- Tenant-Specific Connections
  - Tenants are identified by the presence or absence of the tenantId property in the JDBC connection string
  - A tenant-specific connection may only query:
    - Their own data in multi-tenant tables
    - All data in non-multi-tenant (global) tables
    - Their own schema (tenant-specific views)

Multi-Tenancy

```
CREATE TABLE event (
    tenant_id VARCHAR,
    type CHAR(1),
    event_id BIGINT,
    created_date DATE,
    created_by VARCHAR,
    CONSTRAINT pk PRIMARY KEY (tenant_id, type, event_id))
    MULTI_TENANT=true;
```

First PK column identifies tenant ID

Tenant-specific connection

```
DriverManager.connect(“jdbc:phoenix:localhost;tenantId=me”);
```

APACHE Phoenix

"We put the SQL back in NoSQL"

Secondary Indexes

- DDL provides several index types
- Can define *covered* columns
- Guarantees
  - On successful return to the client all data has been persisted to all interested indexes and the primary table
  - Updates are first made to the index tables
    - Index tables are only ever a single edit ahead of the primary table
- Failure handling
  - Index updates are added to the WAL of the primary table and processed as part of log recovery
  - Indexes are automatically offlined when the primary table becomes unavailable, until it comes back

Secondary Indexes

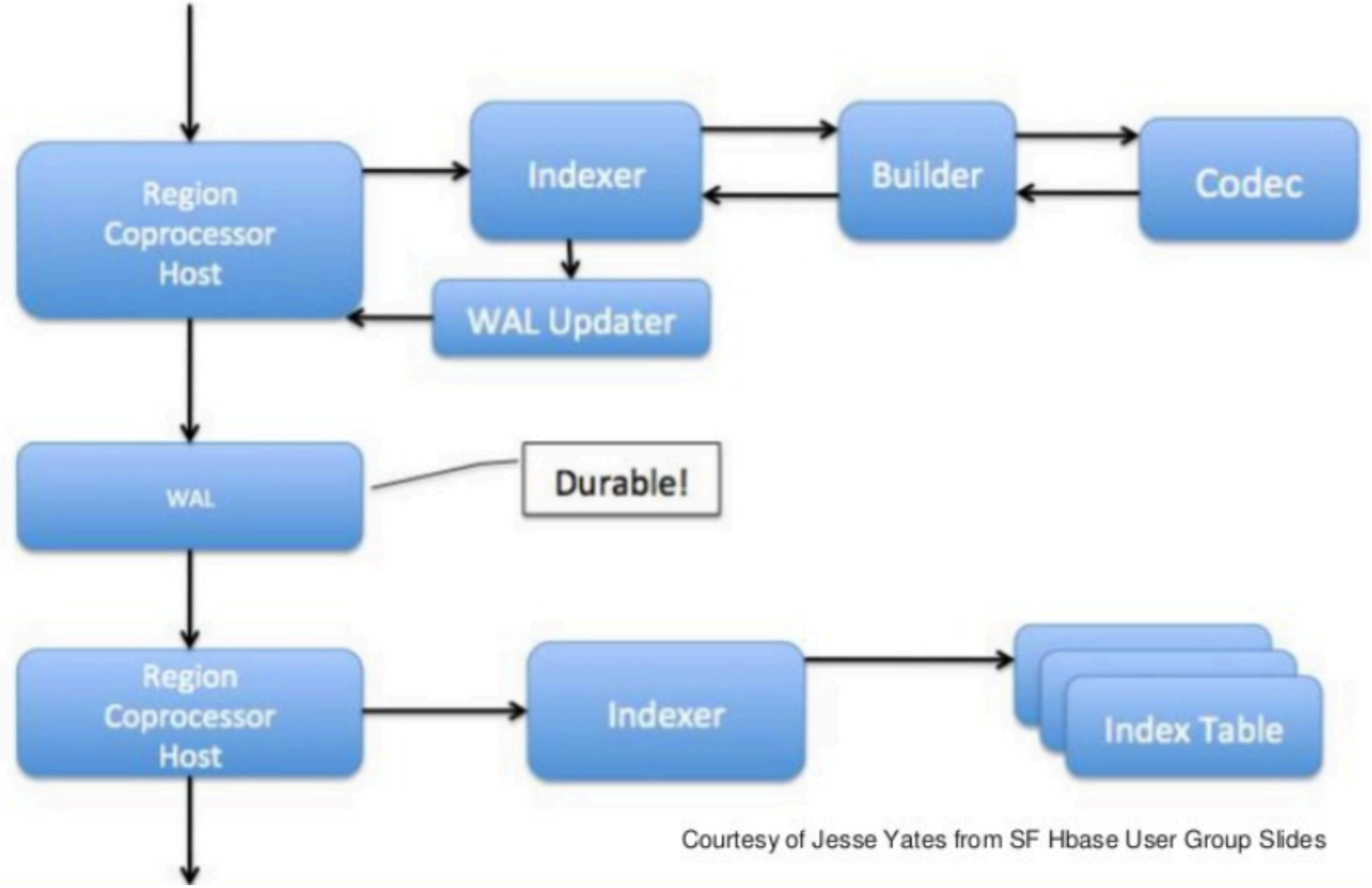
- Mutable indexes
  - Global mutable indexes
    - Server side intercepts primary table updates, builds the index updates and sends them to index tables, possibly remote
    - For *read heavy, low write* uses cases
  - Local mutable indexes
    - Index data and primary data are placed together on same servers
    - Higher read time cost than with global indexes (the exact region location of index data cannot be predetermined)
    - For *write heavy, space constrained* use cases
- Immutable indexes
  - Managed entirely by the client (writes scale more)
  - Contract is: Once written primary rows are never updated
  - For use cases that are *write once, append*

APACHE Phoenix

"We put the SQL back in NoSQL"

Secondary Indexes

- (Mutable) index update flow



Courtesy of Jesse Yates from SF Hbase User Group Slides

APACHE Phoenix
"We put the SQL back in NoSQL"

Secondary Indexes

- Creating a global index with *covered* columns

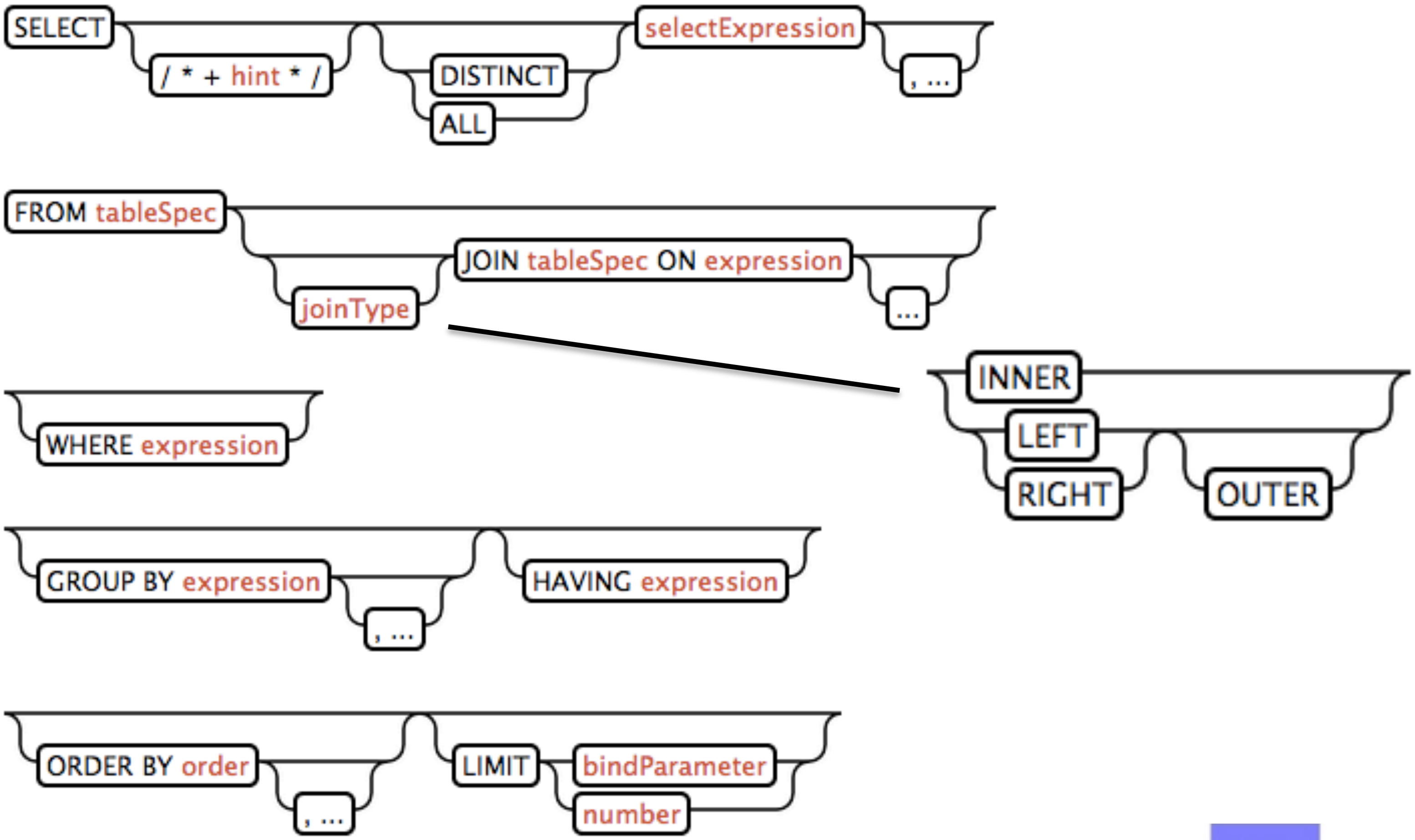
```
CREATE TABLE t (k VARCHAR PRIMARY KEY,
      v1 VARCHAR, v2 INTEGER);

CREATE INDEX i ON t (v1) INCLUDE (v2);
```

Covered column

  - Data in covered columns will be copied into the index
  - This allows a global index to be used more frequently, as a global index will only be used if all columns referenced in the query are contained by it

Joins

- Standard JOIN syntax supported, with limitations



SELECT

/ * + hint * /

DISTINCT

ALL

selectExpression

, ...

FROM tableSpec

joinType

JOIN tableSpec ON expression

...

INNER

LEFT

RIGHT

OUTER

WHERE expression

GROUP BY expression

, ...

HAVING expression

ORDER BY order

, ...

LIMIT

bindParameter

number

APACHE Phoenix

"We put the SQL back in NoSQL"

Joins

- JOIN limitations
  - FULL OUTER JOIN and CROSS JOIN are not supported
  - Only equality (=) is supported in joining conditions
    - No restriction on other predicates in the ON clause
- Enhancements in latest releases
  - Derived tables are supported as of 4.1
  - Sub-joins are supported as of 4.1, currently the join table can only be a named table
  - Semi- and anti-joins (IN and EXISTS subqueries) and correlated subqueries are supported as of 4.2, only in the WHERE clause and only with equality (=) constraints

Joins

- Only hash join physical plans are available
  - One side of the join must be small enough to be broadcast to all servers and held in memory during query execution
- Secondary indexes will be automatically utilized when running join queries if available
- Server-side caches are used to hold the hashed sub-query results
  - Configuration or query changes will be necessary if you encounter InsufficientMemoryExceptions
- Should be considered a work in progress

Salted Tables

- HBase tables can develop “hot spots” when writing data with monotonically increasing row keys
  - HBase RegionServers serve regions of data, which are ranges of lexiographically sorted rows
  - Region hosting is exclusive, load can fall all onto one server
- Phoenix can “salt” keys into N buckets such that writes fan out N-ways even with monotonic primary keys

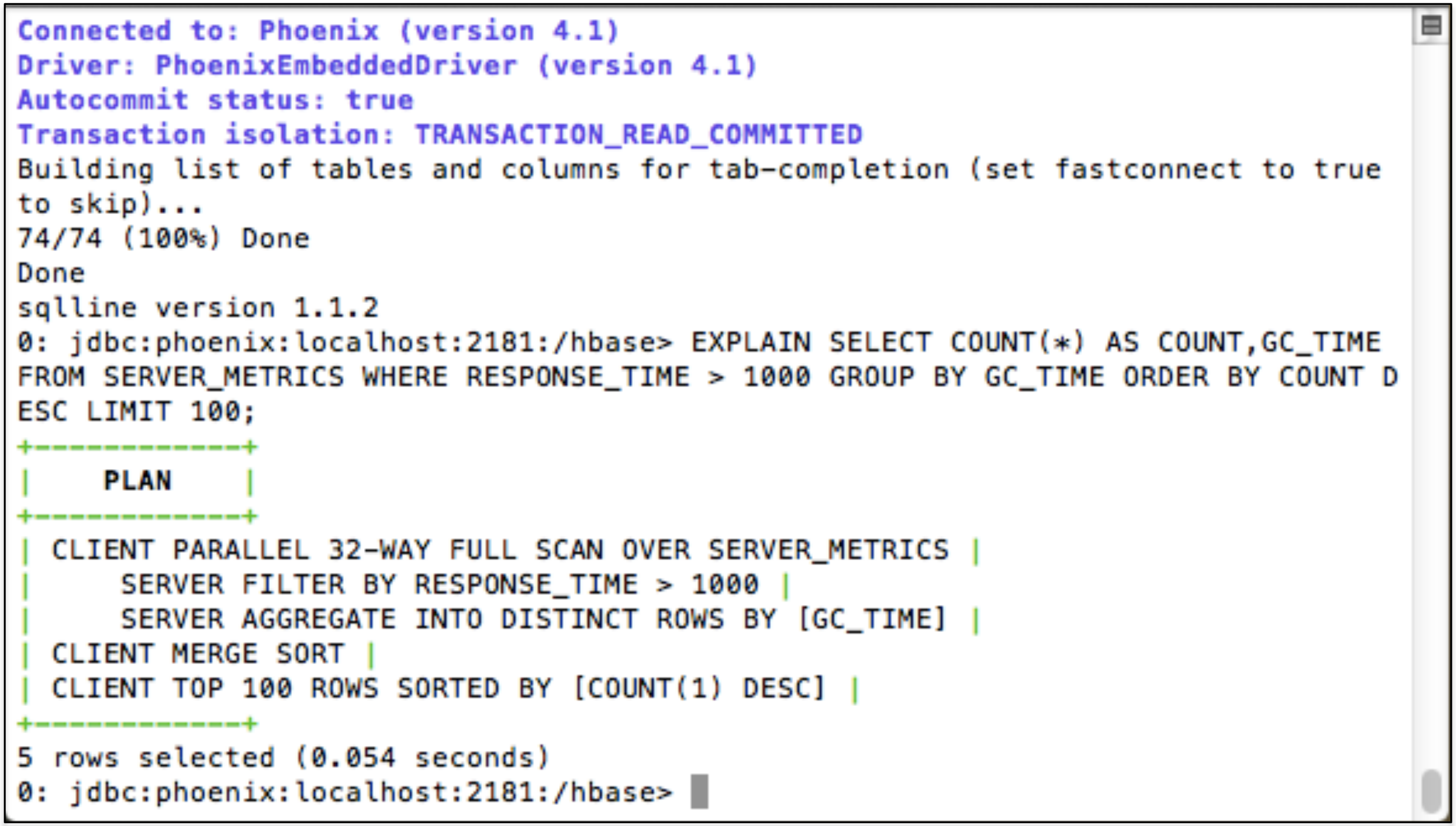
```
CREATE TABLE T (K VARCHAR PRIMARY KEY, …)
SALT_BUCKETS=32;
```

- For best results, N should approximate the number of RegionServers in the HBase cluster

APACHE Phoenix
"We put the SQL back in NoSQL"

Query Optimization

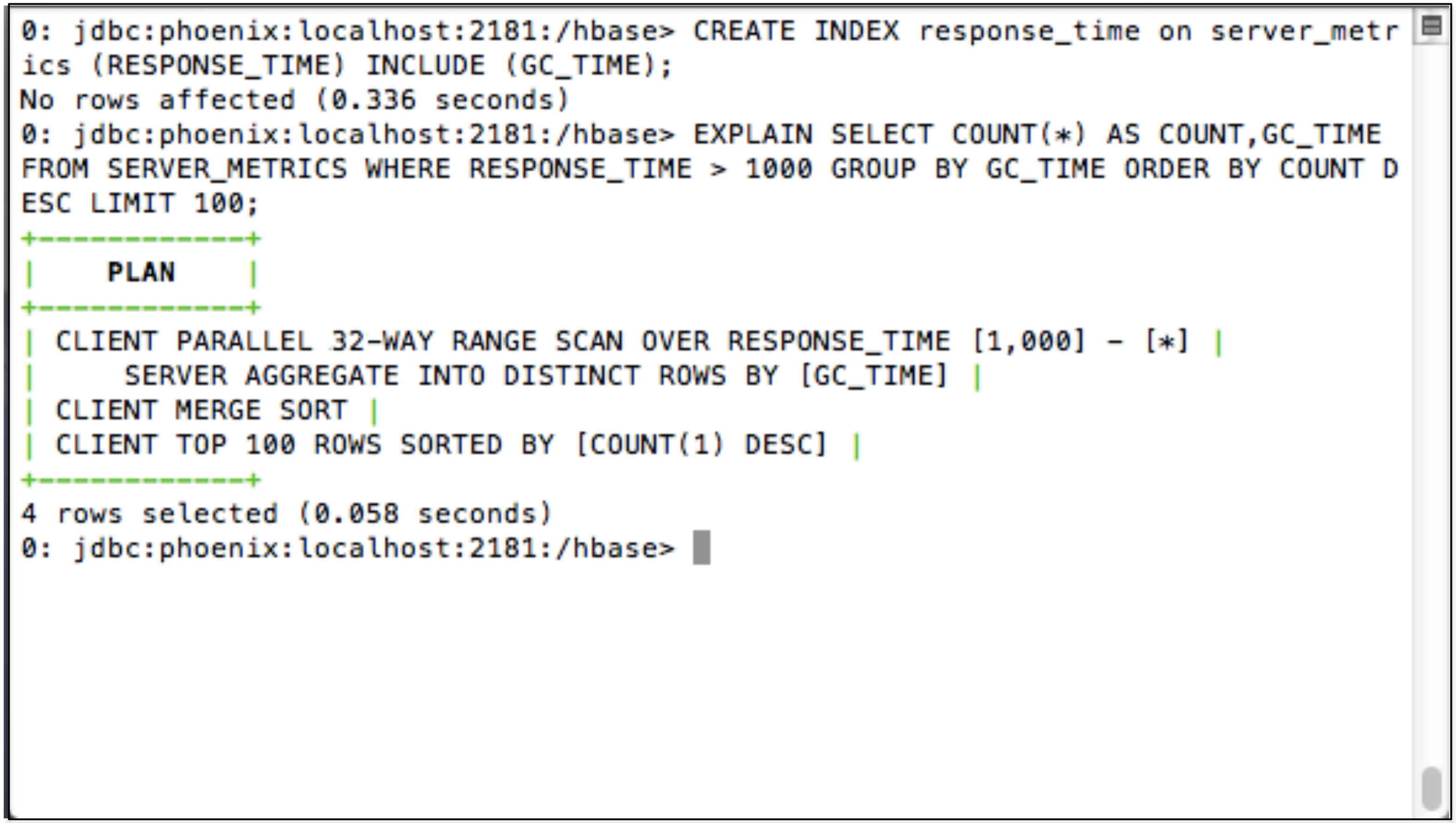
- Example query plan for a 32 region table



```
Connected to: Phoenix (version 4.1)
Driver: PhoenixEmbeddedDriver (version 4.1)
Autocommit status: true
Transaction isolation: TRANSACTION_READ_COMMITTED
Building list of tables and columns for tab-completion (set fastconnect to true
to skip)...
74/74 (100%) Done
Done
sqlline version 1.1.2
0: jdbc:phoenix:localhost:2181:/hbase> EXPLAIN SELECT COUNT(*) AS COUNT,GC_TIME
FROM SERVER_METRICS WHERE RESPONSE_TIME > 1000 GROUP BY GC_TIME ORDER BY COUNT D
ESC LIMIT 100;
+------------+
|    PLAN    |
+------------+
| CLIENT PARALLEL 32-WAY FULL SCAN OVER SERVER_METRICS |
|     SERVER FILTER BY RESPONSE_TIME > 1000 |
|     SERVER AGGREGATE INTO DISTINCT ROWS BY [GC_TIME] |
| CLIENT MERGE SORT |
| CLIENT TOP 100 ROWS SORTED BY [COUNT(1) DESC] |
+------------+
5 rows selected (0.054 seconds)
0: jdbc:phoenix:localhost:2181:/hbase>
```

Query Optimization

- With a secondary index on RESPONSE_TIME



```
0: jdbc:phoenix:localhost:2181:/hbase> CREATE INDEX response_time on server_metr
ics (RESPONSE_TIME) INCLUDE (GC_TIME);
No rows affected (0.336 seconds)
0: jdbc:phoenix:localhost:2181:/hbase> EXPLAIN SELECT COUNT(*) AS COUNT,GC_TIME
FROM SERVER_METRICS WHERE RESPONSE_TIME > 1000 GROUP BY GC_TIME ORDER BY COUNT D
ESC LIMIT 100;
+------------+
|    PLAN    |
+------------+
| CLIENT PARALLEL 32-WAY RANGE SCAN OVER RESPONSE_TIME [1,000] - [*] |
|     SERVER AGGREGATE INTO DISTINCT ROWS BY [GC_TIME] |
| CLIENT MERGE SORT |
| CLIENT TOP 100 ROWS SORTED BY [COUNT(1) DESC] |
+------------+
4 rows selected (0.058 seconds)
0: jdbc:phoenix:localhost:2181:/hbase> 
```

Query Optimization

- Client side rewriting
  - Parallel scanning with final client side merge sort
  - RPC batching
  - Use secondary indexes if available
  - Rewrites for multitenant tables
- Statistics
  - Use guideposts to increase intra-region parallelism
- Server side push down
  - Filters
  - Skip scans
  - Partial aggregation
  - TopN
  - Hash joins

Query Optimization

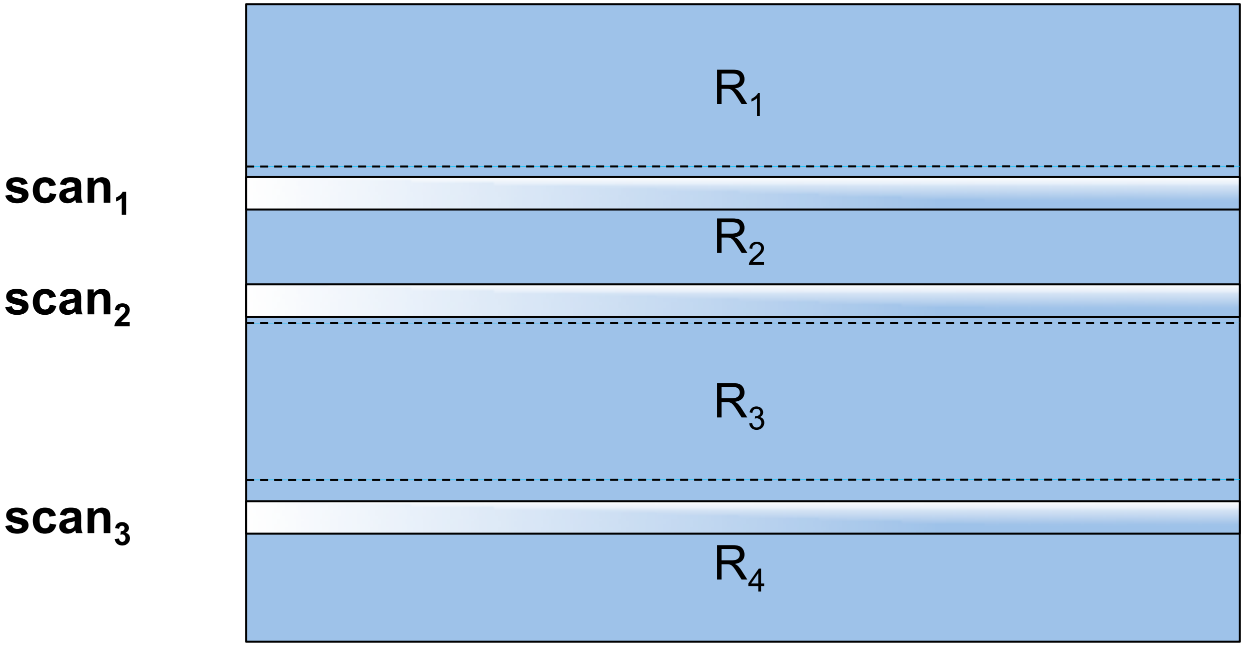
- Future work considers integrating Apache Calcite
  - http://calcite.incubator.apache.org
  - Cost based optimization
  - ~120 rewrite rules
  - Support for materialized views, lattices, tiles, etc.

Statistics

- As of 4.2, Phoenix collects a set of keys per region per column family that are equidistant by volume of intervening data
  - These keys are called *guideposts* and they act as hints for increased parallelization of queries over target regions
  - Helps avoid region scanner lease timeouts
- Collected automatically during major compaction and region splits

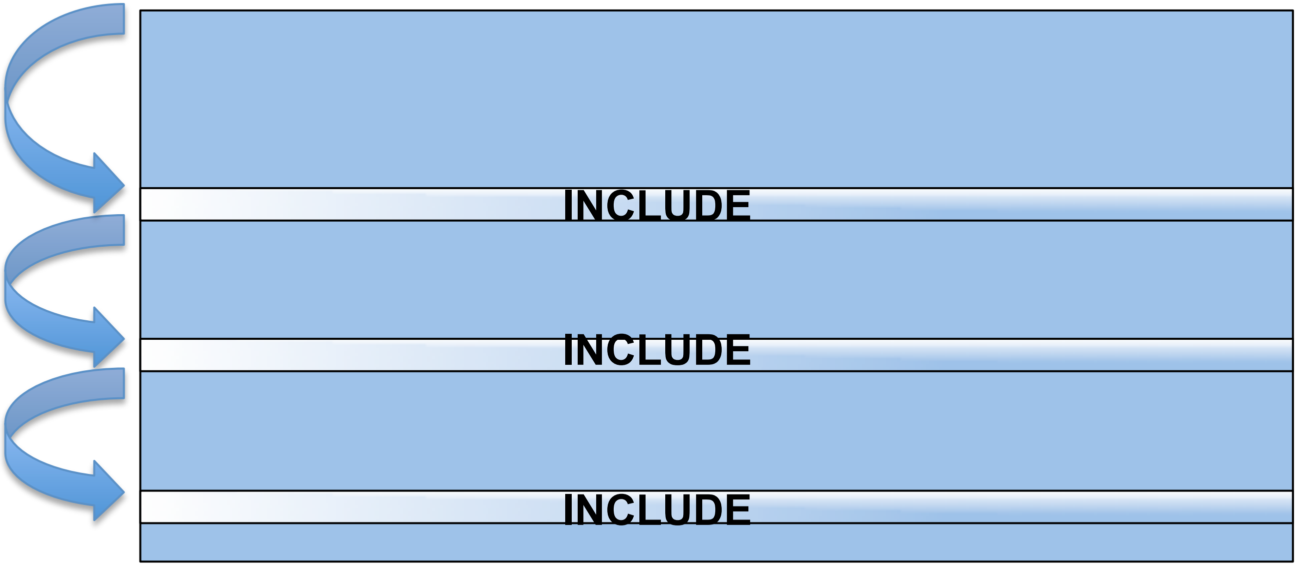
Skip Scans

- The optimizer identifies sub-regions of interest in the key space and chunks parallel scans by region boundaries (and guideposts if available)



Skip Scans

- Within a region, column value ranges are pushed down into a filter that uses SEEK_NEXT_HINT to quickly skip through data



APACHE Phoenix
"We put the SQL back in NoSQL"

Partial Aggregation

- Phoenix runs aggregations in parallel on the server, where the data lives
  - GROUP BY and/or aggregate functions
  - Only the aggregate values are returned for each grouping

SERVER METRICS

HOST	DATE	KV$_1$	KV$_2$	KV$_3$
SF1	Jun 2 10:10:10.234	239	234	674
SF1	Jun 3 23:05:44.975		23	234
SF1	Jun 9 08:10:32.147	256	314	341
SF1	Jun 9 08:10:32.147	235	256	
SF1	Jun 1 11:18:28.456		235	23
SF1	Jun 3 22:03:22.142	234		314
SF1	Jun 3 22:03:22.142	432	234	256
SF2	Jun 1 10:29:58.950	23	432	
SF2	Jun 2 14:55:34.104	314	876	23
SF2	Jun 3 12:46:19.123	256	234	314
SF2	Jun 3 12:46:19.123		432	
SF2	Jun 8 08:23:23.456	876	876	235
SF2	Jun 1 10:31:10.234	234	234	876
SF3	Jun 1 10:31:10.234	432	432	234
SF3	Jun 3 10:31:10.234		890	
SF3	Jun 8 10:31:10.234	314	314	235
SF3	Jun 1 10:31:10.234	256	256	876
SF3	Jun 1 10:31:10.234	235		234
SF3	Jun 8 10:31:10.234	876	876	432
SF3	Jun 9 10:31:10.234	234	234	
SF3	Jun 3 10:31:10.234		432	276
…	…	…	…	…

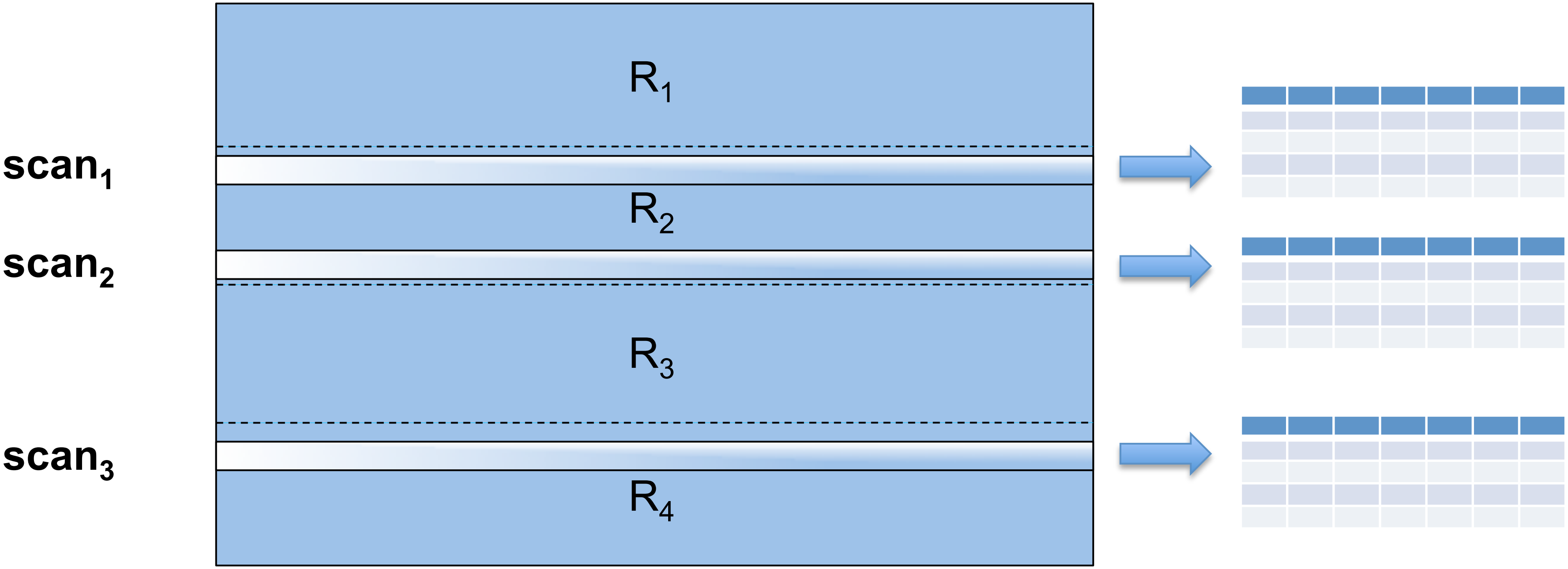
SERVER METRICS

HOST	AGGREGATE VALUES
SF1	3421
SF2	2145
SF3	9823

APACHE Phoenix
"We put the SQL back in NoSQL"

TopN

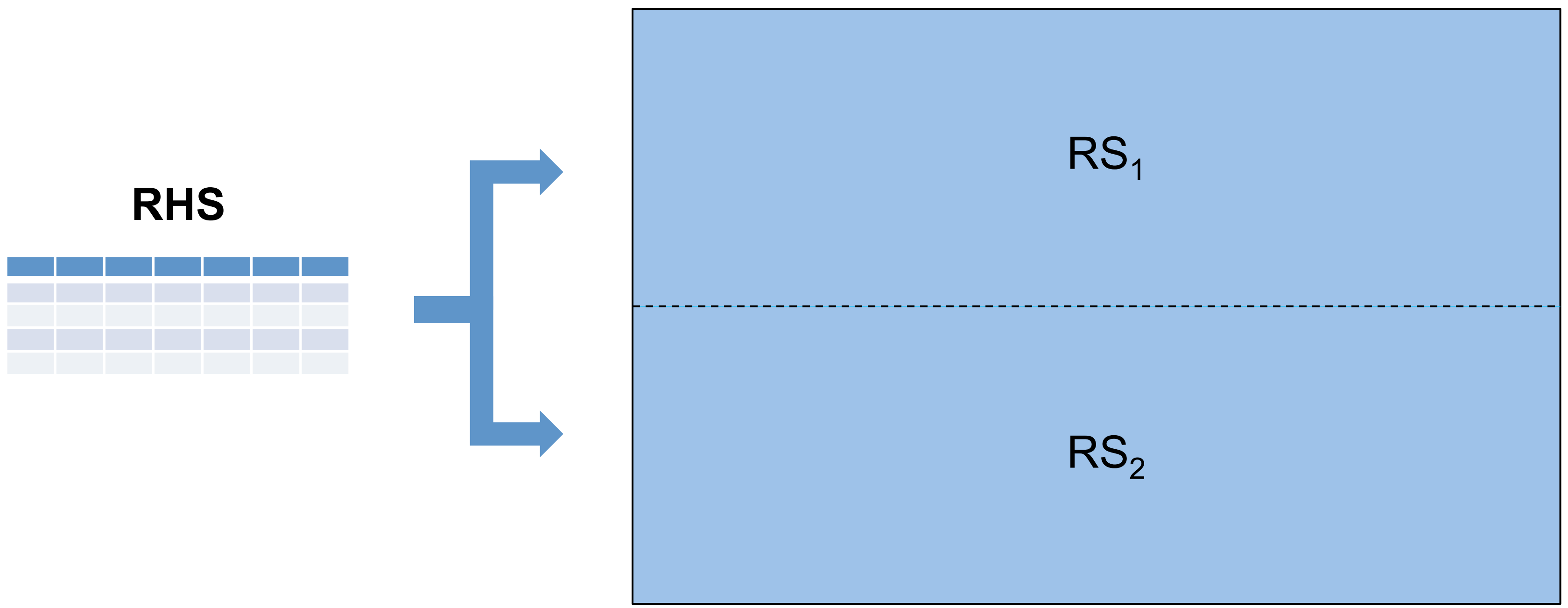
- Parallel scans are chunked by region boundaries (and guideposts if available)
- TopN coprocessor holds on to top N rows by chunk
- Client does a final merge sort



APACHE Phoenix
"We put the SQL back in NoSQL"

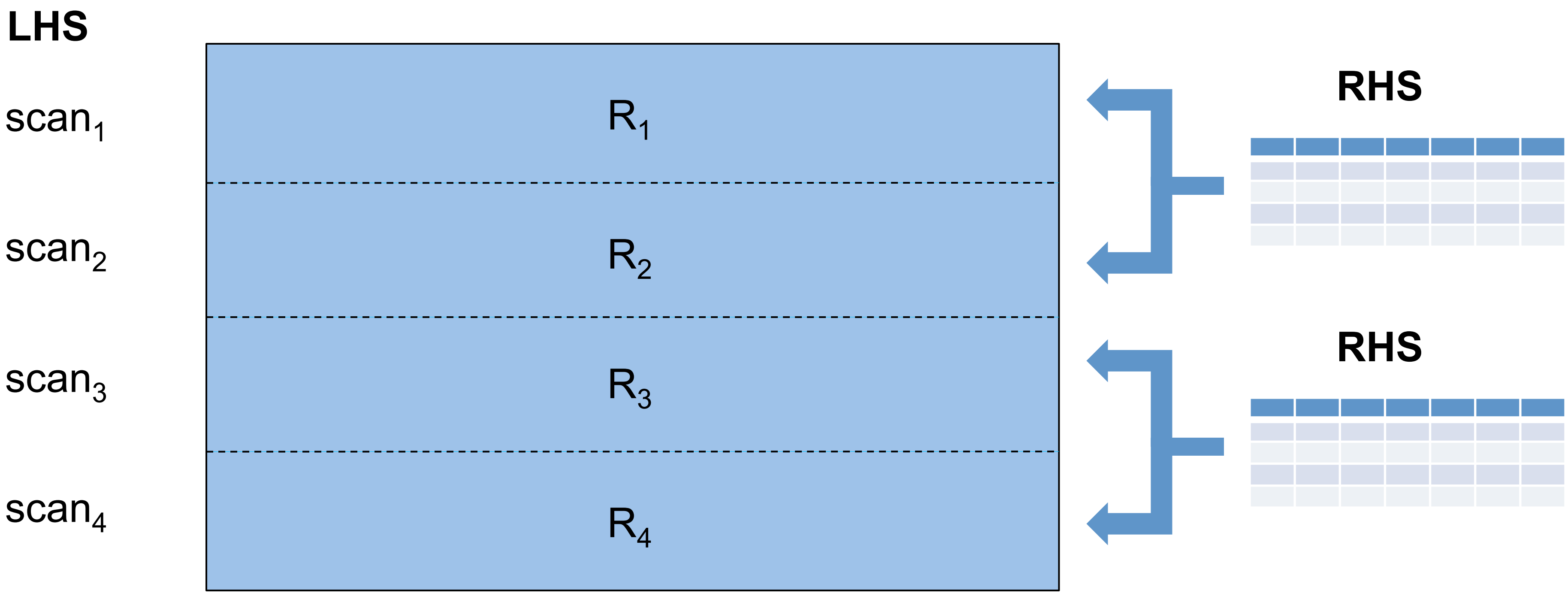
Hash Joins

- Separate query into LHS and RHS
- Execute RHS and broadcast the result to all RegionServers



Hash Joins

- Execute LHS and join in coprocessor



APACHE Phoenix
"We put the SQL back in NoSQL"

Roadmap

- Transactions, probably via Tephra*
- Many-to-many joins
- Cost-based query optimizer
  - Enhanced statistics collection, histograms
  - Apache Calcite integration
- Query server
  - Like Apache Hive's HiveServer2
  - Opens the door to on demand use of available Hive or Spark server-side resources
- OLAP extensions
  - WINDOW, PARTITION OVER, RANK, and other SQL-92 extensions

* - http://blog.cask.co/2014/07/meet-tephra-an-open-source-transaction-engine-2/

Roadmap

- Functional indexes
- Table sampling
- Surface native HBase multiversioning
- Security
  - GRANT and REVOKE using the HBase AccessController
  - Per cell labels and visibility expressions
  - Transparent encryption

End

(Questions?)