Joins in HBase

The standard SQL join syntax (with some limitations) is now supported by Phoenix to combine records from two or more tables based on their fields having common values.

For example, we have the following tables to store our temperature, humidity, pressure, and vibrations information during a manufacturing process.

- **Heat** and **temperature** are related, but not the same. Temperature is a measure of the average kinetic energy of the particles of a substance. The higher the temperature of an object, the higher is its kinetic energy. Kinetic energy is a type of energy associated with motion.
- **Humidity** is simply the amount of water vapor held in the air. Water vapor is the gaseous state of water. As the temperature of the air increases more water vapor can be held since the movement of molecules at higher temperatures prevents condensation from occurring.
- **Pressure** is typically measured in units of force per unit of surface area. … Instruments used to measure and display pressure in an integral unit are called pressure gauges or vacuum gauges. A manometer is a good example, as it uses a column of liquid to both measure and indicate pressure.
- **Vibrations** may influence the durability and reliability of machinery systems or structures and cause problems such as

damage, abnormal stopping and disaster. Vibration measurement is an important countermeasure to prevent these problems.

There are multiple ways this data might be modeled in a database. HBase is no exception. So let’s see how this might be done more effectually.

Data Model

The “Simulation” table:

id	prop_id	temperature	telemetry_id	humidity	pressu
1630781	053243	75.0	1245	70.6	246.0
1630782	958643	69.3	1247	73.3	244.6
1630783	053243	82.8	1249	84.8	245.3
1630784	958643	79.6	1247	85.7	244.6
1630785	534063	88.2	1246	84.5	247.2
1630786	636924	87.8	1245	85.0	246.7
1630787	534063	79.5	1250	78.8	248.9

1630788	347462	86.7	1250	83.5	244.3
1630789	636924	87.0	1247	83.2	245.1
1630790	534063	79.9	1247	86.4	244.8

> *Note: the `simulation` table represents readings taken during the day from different sources.*

The “Properties” table:

prop_id	type	location	latitude	longitude
053243	Cooker 12	Building 2	47.640792	-122.126258
534063	Shipping 1	Building 12	48.435268	-132.432566
958643	Weigher 321	Building 9	52.476970	-154.124370
347462	Cooker 11	Building 22	51.798355	-146.886545
636924	Shipping 4	Building 2	47.640792	-167.564355

The “Telemetry” table:

tel_id	name	version	vib_low	vib_hi	vib_unit	temp_low
1245	cooker-sensors	14	10.6	14.2	hz	88.3
1246	range-sensors	1	0.0	0.0	null	42.0

1247	network-sensors	12	12.8	13.4	hz	0.0
1248	building-sensors	3	41.0	41.2	hz	49.0
1249	balance-sensors	6	12.4	13.1	hz	75.0
1250	freight-weight	1	0.0	0.0	null	40.0

> *Note: these two tables represent the supporting data.*

The SQL

Now go into Phoenix:

```
    cd /usr/hdp/current/phoenix-client/bin
    ./sqlline.py localhost
```

Let’s create the tables:

```
CREATE TABLE simulation (
    id INTEGER NOT NULL,
    prop_id INTEGER,
    temperature DOUBLE,
    telemetry_id INTEGER,
    humidity DOUBLE,
    pressure DOUBLE,
```

```
    timestamp TIMESTAMP,
    CONSTRAINT pk PRIMARY KEY (id));

CREATE TABLE properties (
    prop_id INTEGER NOT NULL,
    type VARCHAR,
    location VARCHAR,
    latitude DOUBLE,
    longitude DOUBLE,
    CONSTRAINT pk PRIMARY KEY (prop_id));

CREATE TABLE telemetry (
    tel_id INTEGER NOT NULL,
    name VARCHAR,
    version INTEGER,
    temperature DOUBLE,
    vib_low DOUBLE,
    vib_hi DOUBLE,
    vib_unit CHAR(2),
    temp_low DOUBLE,
    temp_hi DOUBLE,
    temp_unit CHAR(2),
    CONSTRAINT pk PRIMARY KEY (tel_id));
```

Easier to copy:

```
CREATE TABLE telemetry (tel_id INTEGER NOT NULL,name VARCH
```

```
AR,version INTEGER,temperature DOUBLE,vib_low DOUBLE,vib_h
i DOUBLE,vib_unit CHAR(2),temp_low DOUBLE,temp_hi DOUBLE,t
emp_unit CHAR(2),CONSTRAINT pk PRIMARY KEY (tel_id));
CREATE TABLE properties (prop_id INTEGER NOT NULL,type VAR
CHAR,location VARCHAR,latitude DOUBLE,longitude DOUBLE,CON
STRAINT pk PRIMARY KEY (prop_id));
CREATE TABLE simulation (id INTEGER NOT NULL,prop_id INTEG
ER,temperature DOUBLE,telemetry_id INTEGER,humidity DOUBLE
,pressure DOUBLE,timestamp TIMESTAMP,CONSTRAINT pk PRIMARY
 KEY (id));
```

Try SQL on one of the tables:

```
    0: jdbc:phoenix:localhost> select * from properties;
    +---------+-------+-----------+-----------+-----------
-+
    | LOC_ID  | TYPE  | LOCATION  | LATITUDE  | LONGITUDE
 |
    +---------+-------+-----------+-----------+-----------
-+
    +---------+-------+-----------+-----------+-----------
-+
    No rows selected (0.081 seconds)
```

Now let’s populate the tables:

```
#
```

```
# TABLE STRUCTURE FOR: telemetry
#

UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (0, 'cooker-sensors', 7, 120.0, 45.0, 49.9, 'hz', 1
99.9, 380.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (1, 'range-sensors', 2, 120.0, 45.0, 49.9, 'hz', 19
9.0, 309.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (2, 'network-sensors', 6, 120.0, 45.0, 49.9, 'hz',
80.9, 99.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (3, 'building-sensors', 3, 120.0, 45.9, 99.9, 'hz',
 50.0, 109.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (4, 'balance-sensors', 3, 120.0, 45.0, 49.9, 'hz',
80.0, 109.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (5, 'freight-weight', 6, 120.0, 10.9, 99.9, 'hz', 8
0.0, 99.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
```

```
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (6, 'vibration-sensors', 7, 120.0, 10.0, 49.9, 'hz'
, 99.0, 109.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (7, 'calibration-sensors', 6, 120.0, 99.0, 120.9, '
hz', 99.9, 109.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (8, 'misrange-sensors', 6, 120.0, 190.0, 349.9, 'hz
', 99.9, 99.9, 'F');
UPSERT INTO telemetry (tel_id, name, version, temperature,
 vib_low, vib_hi, vib_unit, temp_low, temp_hi, temp_unit)
VALUES (9, 'visual-sensors', 1, 120.0, 10.9, 99.9, 'hz', 9
9.9, 99.9, 'F');

#
# TABLE STRUCTURE FOR: properties
#

UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (0, 'Cooker 12', 'Building 2', 47.64079
2, -122.126258);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (1, 'Shipping 1', 'Building 12', 48.435
2689, -167.564355);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (2, 'Weigher 321', 'Building 6', 52.476
```

```
970, -146.886545);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (3, 'Cooker 11', 'Building 3', 48.43526
89, -146.886545);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (4, 'Shipping 4', 'Building 5', 48.4352
689, -167.564355);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (5, 'Weigher 322', 'Building 6', 48.435
2689, -146.886545);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (7, 'Weigher 321', 'Building 12', 52.47
6970, -146.886545);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (8, 'Shipping 4', 'Building 1', 48.4352
689, -167.564355);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (9, 'Weigher 310', 'Building 4', 48.435
2689, -146.886545);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (19, 'Shipping 8', 'Building 8', 52.476
970, -167.564355);
UPSERT INTO properties (prop_id, type, location, latitude,
 longitude) VALUES (21, 'Cooker 17', 'Building 4', 48.4352
689, -146.886545);



#
```

```
# TABLE STRUCTURE FOR: simulation
#


UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (0, 19, 88.6,
1, 0.0, 100.2, '2017-04-02 17:24:24');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (1, 5, 6.0, 9,
 0.0, 106.6, '2017-04-23 04:55:47');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (2, 3, 23.8, 8
, 0.0, 98.3, '2017-06-11 16:28:37');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (3, 7, 8.7, 2,
 99.9, 100.6, '2017-09-08 09:27:27');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (4, 4, 99.9, 1
, 0.0, 102.9, '2017-10-20 21:02:39');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (5, 9, 99.9, 8
, 99.9, 77.7, '2017-03-09 05:30:34');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (6, 21, 99.9,
2, 0.0, 99.9, '2017-02-05 08:47:51');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (7, 1, 99.9, 6
, 23.8, 110.4, '2017-02-09 03:23:23');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
```

```
y_id, humidity, pressure, timestamp) VALUES (8, 0, 0.0, 4,
 99.9, 115.7, '2017-03-14 17:35:15');
UPSERT INTO simulation (id, prop_id, temperature, telemetr
y_id, humidity, pressure, timestamp) VALUES (9, 21, 99.9,
8, 38.6, 99.6, '2017-10-31 21:33:51');
```

You may get a combined view of the “Simulation” table and the “Properties” table by running the following join query:

```
    SELECT S.id, P.type, P.location, S.temperature, S.time
stamp
    FROM simulation AS S
    INNER JOIN properties AS P
    ON S.prop_id = P.prop_id;
```

Pretty simple join (although it’s on HBase!). Just to prove that, go into HBase (may want to open another session) and run:

```
hbase(main):003:0> scan 'PROPERTIES'
ROW                   COLUMN+CELL

 \x80\x00\x00\x00     column=0:\x00\x00\x00\x00, timestamp=
1534509439028, value=x
 \x80\x00\x00\x00     column=0:\x80\x0B, timestamp=15345094
39028, value=Cooker 12
 \x80\x00\x00\x00     column=0:\x80\x0C, timestamp=15345094
39028, value=Building 2
```

```
 \x80\x00\x00\x00    column=0:\x80\x0D, timestamp=15345094
39028, value=\xC0G\xD2\x05x\xE5\xC4\xEC
 \x80\x00\x00\x00    column=0:\x80\x0E, timestamp=15345094
39028, value=?\xA1w\xEBc\x90\xC9\x10
 \x80\x00\x00\x01    column=0:\x00\x00\x00\x00, timestamp=
1534509439040, value=x
 \x80\x00\x00\x01    column=0:\x80\x0B, timestamp=15345094
39040, value=Shipping 1
 \x80\x00\x00\x01    column=0:\x80\x0C, timestamp=15345094
39040, value=Building 12
 \x80\x00\x00\x01    column=0:\x80\x0D, timestamp=15345094
39040, value=\xC0H7\xB6\xE4-;\xA3
 \x80\x00\x00\x01    column=0:\x80\x0E, timestamp=15345094
39040, value=?\x9B\x0D\xF0\xCD\xC8uO
 \x80\x00\x00\x02    column=0:\x00\x00\x00\x00, timestamp=
1534509439052, value=x
 \x80\x00\x00\x02    column=0:\x80\x0B, timestamp=15345094
39052, value=Weigher 321
 \x80\x00\x00\x02    column=0:\x80\x0C, timestamp=15345094
39052, value=Building 6
 \x80\x00\x00\x02    column=0:\x80\x0D, timestamp=15345094
39052, value=\xC0J=\x0DZ[\x96*
 \x80\x00\x00\x02    column=0:\x80\x0E, timestamp=15345094
39052, value=?\x9D\xA3\xA1laR*
 ...
```

> *Note: what's different about HBase?*

Joining Tables with Indices

Secondary indices will be automatically utilized when running join queries. For example, if we create indices on the “Simulation” table and the “Properties” table respectively, which are defined as follows:

```
    CREATE INDEX iSimulation ON simulation (id) INCLUDE (p
rop_id, telemetry_id);
    10 rows affected (6.331 seconds)
```

Now do several more:

```
    CREATE INDEX i2Simulation ON simulation (prop_id) INCL
UDE (telemetry_id, temperature);
    CREATE INDEX iProperties ON properties (prop_id) INCLU
DE (location);
```

We can find out each item’s total sales value by joining the `telemetry` table and the `simulation` table and then grouping the joined result with `name` (and also adding some filtering conditions):

```
    SELECT T.name, avg(T.temp_low - T.temperature) AS MinT
emperature,
    avg(T.temp_hi - T.temperature) AS MaxTemperature
    FROM telemetry AS T, simulation AS S
    JOIN properties AS P
```

```
    ON S.prop_id = P.prop_id
    WHERE T.name = 'cooker-sensors'
    GROUP BY T.NAME;
```

The results should be like this:

```
    +-----------------+-----------------+-----------------
+
    |     T.NAME      | MINTEMPERATURE  | MAXTEMPERATURE
|
    +-----------------+-----------------+-----------------
+
    | cooker-sensors  | 79.8999         | 260.9
|
    +-----------------+-----------------+-----------------
+
    1 row selected (0.053 seconds)
```

The execution plan for this query (by running “EXPLAIN [the query]”) will be:

```
+-------------------------------------------------------
------------------------------+-----------------+---+
|                                                PLAN
                              | EST_BYTES_READ  | E |
+-------------------------------------------------------
------------------------------+-----------------+---+
```

```
| CLIENT 1-CHUNK PARALLEL 1-WAY FULL SCAN OVER TELEMETRY
                                  | null            | n |
|     SERVER FILTER BY NAME = 'cooker-sensors'
                                  | null            | n |
|     SERVER AGGREGATE INTO DISTINCT ROWS BY [T.NAME]
                                  | null            | n |
| CLIENT MERGE SORT
                                  | null            | n |
|     PARALLEL INNER-JOIN TABLE 0
                                  | null            | n |
|         CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL S
CAN OVER ISIMULATION              | null            | n |
|             PARALLEL INNER-JOIN TABLE 0 (SKIP MERGE)
                                  | null            | n |
|                 CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBI
N FULL SCAN OVER IPROPERTIES      | null            | n |
|                     SERVER FILTER BY FIRST KEY ONLY
                                  | null            | n |
+-------------------------------------------------------
--------------------------------+-----------------+---+
9 rows selected (0.069 seconds)
```

In this case, the index table “iProperties” is used in place of the data table “Properties” since the index table “iProperties” is indexed on column “prop_id” and will hence benefit the GROUP-BY clause in this query.

Meanwhile, the index table “iSimulation” is favored over the data table “Simulation” and another index table “iSimulation” because a range scan instead of a full scan can be applied as a result of the WHERE clause.

Grouped Joins and Derived Tables

Phoenix also supports complex join syntax such as grouped joins (or sub joins) and joins with derived-tables. You can group joins by using parenthesis to prioritize certain joins before other joins are executed.

You can also replace any one (or more) of your join tables with a subquery (derived table), which could be yet another join query.

For grouped joins, you can write something like:

```
    SELECT T.name, T.temperature, S.humidity, S.pressure
    FROM telemetry AS T
    LEFT JOIN
        (properties AS P
        INNER JOIN simulation as S
        ON P.prop_id = S.prop_id)
    ON T.tel_id = S.telemetry_id;
```

By replacing the sub join with a subquery (derived table), we get an equivalent query as:

```
    SELECT T.name, J.temperature, J.humidity, J.pressure
    FROM telemetry AS T
    LEFT JOIN
        (SELECT S.prop_id, temperature, humidity, pressure
        FROM simulation AS S
        INNER JOIN properties AS P
        ON S.prop_id = P.prop_id) AS J
    ON T.tel_id = J.prop_id;
```

As an alternative to the earlier example where we try to find out whether the sensors are within range, instead of using group-by after joining the two tables, we can join the “simulation” detail rows with the grouped rows from the “telemetry” table:

```
    SELECT J.name, J.MinTemperature, J.MaxTemperature
    FROM simulation AS S
    JOIN
        (SELECT
            avg(temp_low - temperature) AS MinTemperature,
            avg(temp_hi - temperature) AS MaxTemperature,
            tel_id,
            name
            FROM telemetry
            WHERE name = 'range-sensors'
            GROUP BY tel_id,name) AS J
    ON S.telemetry_id = J.tel_id;
```

Hash Join vs. Sort-Merge Join

Basic hash join usually outperforms other types of join algorithms, but it has its limitations too, the most significant of which is the assumption that one of the relations must be small enough to fit into memory.

Thus Phoenix now has both hash join and sort-merge join implemented to facilitate fast join operations as well as join between two large tables.

Phoenix currently uses the hash join algorithm whenever possible since it is usually much faster. However we have the hint `USE_SORT_MERGE_JOIN` for forcing the usage of sort-merge join in a query.

The choice between these two join algorithms, together with detecting the smaller relation for hash join, will be done automatically in the future under the guidance provided by table statistics.

Foreign Key to Primary Key Join Optimization

Oftentimes a join will occur from a child table to a parent table, mapping the foreign key of the child table to the primary key of the parent. So instead of doing a full scan on the parent table, Phoenix will drive a skip-scan or a range-scan based on the foreign key values it got from the child table result.

Phoenix will extract and sort multiple key parts from the join keys so that it can get the most accurate key hints/ranges possible for the parent table scan.

For example, we have a parent table “machine” and child table “inspection” defined as:

```
CREATE TABLE machine (
    region VARCHAR NOT NULL,
    local_id VARCHAR NOT NULL,
    name VARCHAR,
    prop_id INTEGER,
    startdate TIMESTAMP,
    CONSTRAINT pk PRIMARY KEY (region, local_id));

CREATE TABLE inspection (
    spec_id INTEGER NOT NULL,
    region VARCHAR,
    local_id VARCHAR,
    prop_id INTEGER,
    title VARCHAR,
    category VARCHAR,
    value_read DOUBLE,
    startdate TIMESTAMP,
    CONSTRAINT pk PRIMARY KEY (spec_id));
```

So create the tables.

And now let’s put some data in these:

```
#
# The machine table
#

UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('East Valley', '1280 8045PC', 'East Hannove
r Heating Unit', 21, '2017-10-31 21:33:51');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('East Valley', '1212 8045PC', 'American Eas
tern', 4, '2014-11-14 22:33:46');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('West Slope', '12234 8045PC', 'East Hannove
r Heating Unit', 5, '2015-08-22 13:33:33');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('East Hartford', '11376 8045PC', 'Schlumber
ger 034-FG', 2, '2012-10-14 12:33:14');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('West Valley', '5435 8045PC', 'Thomas Tools
', 19, '2017-06-30 08:33:34');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('Wilson', '1280 8045PC', 'East Hannover Hea
ting Unit', 7, '2015-12-31 07:33:55');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('Marquant', 'EASTERN 5043', 'Light Tower Re
ntals', 11, '2014-09-30 11:33:51');
```

```
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('Estonia', '30424 40353TT', 'Enfield', 21,
'2011-11-23 14:33:45');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('San Francisco', '1280 8045PC', 'East Hanno
ver Heating Unit', 6, '2015-12-31 13:33:53');
UPSERT INTO machine (region, local_id, name, prop_id, star
tdate) VALUES ('East Valley', '1280 8045PC', 'East Hannove
r Heating Unit', 5, '2016-01-12 01:33:13');

#
# The inspection table
#

UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (0, 'East
 Valley', '1280 8045PC', 3, 'Cooker Participation', 'Cooke
rs', 112, '2017-12-11 21:33:40');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (1, 'East
 Valley', '1280 8045PC', 4, 'Vibration PF-50', 'Cookers',
17.63, '2017-12-11 21:33:41');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (2, 'East
 Valley', '1280 8045PC', 6, 'Handling Provider 5560', 'Coo
kers', 349.6, '2017-12-11 21:33:46');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (3, 'East
```

```
 Valley', '1280 8045PC', 0, 'Chamber Form EC40', 'Cookers'
, 12.74, '2017-12-11 21:33:51');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (4, 'East
 Valley', '1280 8045PC', 7, 'Mag Drive 0343C', 'Cookers',
.554, '2017-12-11 21:33:55');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (5, 'East
 Valley', '1280 8045PC', 6, 'Handling Provider 5560', 'Coo
kers', 10.0, '2017-12-11 21:34:03');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (6, 'East
 Valley', '1280 8045PC', 5, 'Rosterization Maker RC4934',
'Cookers', 144.64, '2017-12-11 21:34:06');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (7, 'East
 Valley', '1280 8045PC', 4, 'Vibration PF-50', 'Cookers',
17.94, '2017-12-11 21:36:51');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (8, 'East
 Valley', '1280 8045PC', 5, 'Rosterization Maker RC4934',
'Cookers', 144.64, '2017-12-11 21:37:33');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (9, 'East
 Valley', '1280 8045PC', 4, 'Vibration PF-50', 'Cookers',
18.34, '2017-12-11 21:38:24');
UPSERT INTO inspection (spec_id, region, local_id, prop_id
, title, category, value_read, startdate) VALUES (10, 'Eas
```

```
t Valley', '1280 8045PC', 4, 'Vibration PF-50', 'Cookers',
 25.64, '2017-12-11 21:38:53');
```

Now a simple select should yield:

```
0: jdbc:phoenix:localhost> select * from inspection;
+----------+--------------+--------------+----------+-----
------------------------+-----------+-------------+-+
| SPEC_ID  |    REGION    |   LOCAL_ID   | PROP_ID  |
       TITLE            | CATEGORY  | VALUE_READ  | |
+----------+--------------+--------------+----------+-----
------------------------+-----------+-------------+-+
| 0        | East Valley  | 1280 8045PC  | 3        | Cook
er Participation        | Cookers   | 112.0       | |
| 1        | East Valley  | 1280 8045PC  | 4        | Vibr
ation PF-50             | Cookers   | 17.63       | |
| 2        | East Valley  | 1280 8045PC  | 6        | Hand
ling Provider 5560      | Cookers   | 349.6       | |
| 3        | East Valley  | 1280 8045PC  | 0        | Cham
ber Form EC40           | Cookers   | 12.74       | |
| 4        | East Valley  | 1280 8045PC  | 7        | Mag
Drive 0343C             | Cookers   | 0.554       | |
| 5        | East Valley  | 1280 8045PC  | 6        | Hand
ling Provider 5560      | Cookers   | 10.0        | |
| 6        | East Valley  | 1280 8045PC  | 5        | Rost
erization Maker RC4934  | Cookers   | 144.64      | |
| 7        | East Valley  | 1280 8045PC  | 4        | Vibr
```

```
ation PF-50              | Cookers   | 17.94       | |
| 8         | East Valley  | 1280 8045PC  | 5          | Rost
erization Maker RC4934  | Cookers   | 144.64      | |
| 9         | East Valley  | 1280 8045PC  | 4          | Vibr
ation PF-50              | Cookers   | 18.34       | |
| 10        | East Valley  | 1280 8045PC  | 4          | Vibr
ation PF-50              | Cookers   | 25.64       | |
+----------+--------------+--------------+----------+-----
-----------------------+-----------+-------------+-+
11 rows selected (0.039 seconds)
```

Now we'd like to find out all those machines which were inspected on December 11, 2017 and we see a trend out of specs:

```
SELECT M.name, M.region, I.value_read
FROM machine AS M
JOIN
    (SELECT region, local_id, value_read
     FROM inspection
     WHERE startdate >= to_date('2017-12-11')
     AND prop_id = 4
     GROUP BY region, local_id, value_read) AS I
ON M.region = I.region AND M.local_id = I.local_id;
```

The above statement will do a skip-scan over the "machine" table and will use both join key "region" and "local_id" for runtime `key hint` calculation.

Below is the execution time of this query with and without this optimization on the “machine” table of about 5000000 records and a “inspection” table of about 1000 records:

W/O Optimization	W/ Optimization
8.1s	0.4s

However, there are times when the foreign key values from the child table account for a complete primary key space in the parent table, thus using skip-scans would only be slower not faster.

Yet you can always turn off the optimization by specifying hint `NO_CHILD_PARENT_OPTIMIZATION`. Furthermore, table statistics will soon come in to help making smarter choices between the two schemes.

Configuration

As mentioned earlier, if we decide to use the hash join approach for our join queries, the prerequisite is that either of the relations can be small enough to fit into memory in order to be broadcast over all servers that have the data of concern from the other relation.

Aside from making sure that the region server heap size is big enough to hold the smaller relation, we might also need to pay a attention to a few configuration parameters that are crucial to running hash joins.

The servers-side caches are used to hold the hash table built upon the

smaller relation. The size and the living time of the caches are controlled by the following parameters.

> *Note: a relation can be a physical table, a view, a subquery, or a joined result of other relations in a multiple-join query.*

The pertinent settings are:

- ***phoenix.query.maxServerCacheBytes***
  Maximum size (in bytes) of the raw results of a relation before being compressed and sent over to the region servers.
  Attempting to serializing the raw results of a relation with a size bigger than this setting will result in a MaxServerCacheSizeExceededException.
  Default: 104,857,600
- ***phoenix.query.maxGlobalMemoryPercentage***
  Percentage of total heap memory (i.e. Runtime.getRuntime().maxMemory()) that all threads may use.
  The summed size of all living caches must be smaller than this global memory pool size. Otherwise, you would get an InsufficientMemoryException.
  Default: 15
- ***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.
Default: 30,000

Although changing parameters can sometimes be a solution to getting rid of the exceptions mentioned above, it is highly recommended that you first consider optimizing the join queries according to the information provided in the following section.

Optimizing Your Query

Now that we know if using hash join it is most crucial to make sure that there will be enough memory for the query execution, but other than rush to change the configuration immediately, sometimes all you need to do is to know a bit of the interiors and adjust the sequence of the tables that appear in your join query.

Below is a description of the default join order (without the presence of table statistics) and of which side of the query will be taken as the “smaller” relation and be put into server cache:

Join Order	**What Happens**
lhs INNER JOIN rhs	rhs will be built as hash table in server cache
lhs LEFT OUTER JOIN rhs	rhs will be built as hash table in server cache
lhs RIGHT OUTER JOIN rhs	**lhs** will be built as hash table in server cache

The join order is more complicated with multiple-join queries. You can try running “EXPLAIN join_query” to look at the actual execution plan.

For multiple-inner-join queries, Phoenix applies star-join optimization by default, which means the leading (left-hand-side) table will be scanned only once joining all right-hand-side tables at the same time.

You can turn off this optimization by specifying the hint `NO_STAR_JOIN` in your query if the overall size of all right-hand-side tables would exceed the memory size limit.

Let’s take a previous query for example:

```
    SELECT T.name, P.type, S.temperature, S.pressure
    FROM simulation AS S
    INNER JOIN telemetry AS T
        ON S.telemetry_id = T.tel_id
    INNER JOIN properties As P
        ON S.prop_id = P.prop_id;
```

The default join order (using star-join optimization) will be:

1. SCAN telemetry --> BUILD HASH[0]
   SCAN properties --> BUILD HASH[1]
2. SCAN simulation JOIN HASH[0], HASH[1] --> Final Resultset

Alternatively, if we use hint `NO_STAR_JOIN`:

```
    SELECT /*+ NO_STAR_JOIN*/ T.name, P.type, S.temperatur
e, S.pressure
    FROM simulation AS S
    INNER JOIN telemetry AS T
        ON S.telemetry_id = T.tel_id
    INNER JOIN properties As P
        ON S.prop_id = P.prop_id;
```

The join order will be:

1. SCAN telemetry --> BUILD HASH[0]
2. SCAN simulation JOIN HASH[0]; CLOSE HASH[0] --> BUILD HASH[1]
3. SCAN properties JOIN HASH[1] --> Final Resultset

It is also worth mentioning that not the entire dataset of the table should be counted into the memory consumption.

Instead, only those columns used by the query, and only the records that satisfy the predicates will be built into the server hash table.

You can see more on the Phoenix tuning guide for more help optimizing your queries.

Results

WOW! OK, think we're done? Not exactly, but that's a lot of SQL! Good job!

A really good book can do wonders with Lars George’s HBase The Definitive Guide. It can save you a ton of hours.

<button type=“button”>Go Back</button>

