Querying Complex Data with Impala

As noted previously, while Impala does support the use of complex data types in tables, it does so with some limitations. Remember, Impala supports the use of complex columns only in Parquet tables. Also, Impala does not support selecting a full complex column simply by including the bare name of the column in the SELECT list, as Hive does. If you issue SELECT * queries on a table with complex columns, the query will run but the complex columns will be omitted from the results.

To access elements within a complex column using Impala, you have to use different syntax depending on the complex type. The syntax for accessing ARRAYs and MAPs also is different from Hive's syntax.

Note: The syntax in this reading is for Impala only. See “Querying Complex Data with Hive” for the syntax to use with Hive.

Pseudocolumns

An important concept in working with complex data using Impala is *pseudocolumns*. To understand pseudocolumns, think of an ARRAY or MAP complex column as a table within a table. This inner table then has columns within it—those are the pseudocolumns. Every ARRAY has two columns named item and pos, and every MAP has two columns named key and value. As you'll see, you then treat the complex column as if it were a table, and use these pseudocolumns to access the elements within the column.

Querying ARRAYs with Impala

To query an element within an ARRAY, treat the array as a table named *tablename.arrayname*with the two pseudocolumns mentioned above: item and pos. The item pseudocolumn gives the value of each ARRAY element. The pos pseudocolumn gives the index of each element within the array. The array index starts at 0. For example, to get the first and second phone numbers in phones from the phones_array_parquet table, use this query:

```
SELECT item
    FROM phones_array_parquet.phones
    WHERE pos = 0 OR pos = 1;
```

Notice that you can use the item and pos pseudocolumns in the WHERE clause as well as the SELECT clause. You can also use them in other clauses, just as if they were regular columns.

Just as if this phones column were a regular table, the query results for this example will include one row for each phone number. This is different than Hive's behavior, which uses a comma-separated list, enclosed in brackets. In the “Complex Data in Practice” reading, you'll see how to produce similar output for Hive.

You often will want your query results to include the values from other columns in the actual table (such as the name column in phones_array_parquet) along with the items in the ARRAY column. The list of phone numbers, without the person who has each number, will probably not be useful! You can use join notation to return ARRAY elements along with scalar column values from corresponding table rows. Typically you use *implicit* join notation (also called *SQL-89-style* join notation) as shown in the following example. It’s also possible to use *explicit* join notation and to specify different join types, such as LEFT OUTER JOIN, but that’s used less often and is not described here. (For more about implicit join syntax, see Course 2, *Analyzing Big Data with SQL,* Week 6, “Alternative Join Syntax.”)

```
SELECT name, phones.item AS phone
    FROM phones_array_parquet, phones_array_parquet.phones;
```

In this example, the FROM clause includes the base table name, which is phones_array_parquet, and the qualified name of the ARRAY column, which is phones_array_parquet.phones. These are separated with a comma, which indicates an implicit join.

Notice that no join condition is specified, because the join condition is implied: the ARRAY elements are joined with the rows they came from.

The SELECT list can then include any columns from the base table along with the item and pos pseudocolumns from the ARRAY column. This example includes the name column and the item pseudocolumn in the SELECT list. It’s a good practice to qualify the item and pos pseudocolumns with the ARRAY column name as shown here (phones.item) but this is not strictly required.

This may seem a bit complicated, but users often find Impala’s familiar join syntax to be more straightforward than what's needed with Hive to get a separate row for each ARRAY element (again, see the “Complex Data in Practice” reading).

Querying MAPs with Impala

Querying a MAP column is similar to querying an ARRAY column, but the pseudocolumns are key and value, representing the keys and values of the MAP elements. For example, to get the home phone numbers, use this query:

```
SELECT value AS home
    FROM phones_map_parquet.phones
    WHERE key = 'home';
```

In this example, the Parquet table named phones_map_parquet contains a MAP column named phones. The MAP keys hold the label (home, work, or mobile) for each phone number, and the associated values hold the phone numbers themselves. To query this MAP column, you use the column name qualified with the table name in the FROM clause: FROM phones_map_parquet.phones. Then you can include one or both of the pseudocolumns in the SELECT list or in other clauses such as WHERE. This example returns the phone numbers (the values), and only phone numbers with the label 'home' will be returned.

As with ARRAY columns, use join notation to return MAP elements along with scalar column values from corresponding table rows.

```
SELECT name, phones.value AS home
    FROM phones_map_parquet, phones_map_parquet.phones
    WHERE phones.key = 'home';
```

In this example, the FROM clause includes the base table name, which is phones_map_parquet, and the qualified name of the MAP column, which is phones_map_parquet.phones. They are separated with a comma to indicate an implicit join. The SELECT list includes the name column and the phones.value pseudocolumn. As with ARRAY, it's a good practice to qualify the value and key pseudocolumns with the name of the MAP column, phones.

Querying STRUCTs with Impala

The Impala query syntax for STRUCT columns is exactly the same as Hive's:

To select a field from a STRUCT column, use the column name, a dot, and the field name (similar to how you can use the database name, a dot, and the table name to refer to a table in a different database from the active one). For example, this query selects the name column, and the state and zipcode fields from the address column:

```
SELECT name, address.state, address.zipcode
    FROM customers_addr_parquet;
```

Try It!

Do the following to run two queries on each of the three *Parquet* tables you created in “Creating Tables with Hive and Impala.” Use Impala for these exercises.

1. First, recall that you created these tables using Hive, so there's something you need to do before you can query the tables with Impala. Do you remember what that is? Figure that out and do it.
2. Try running SELECT * FROM *tablename*; for each table. Notice that the complex column is omitted from the results.
3. Then, run each of the examples in the reading above. Notice the NULL fields in each case. (Why does Bob have no row in the MAP example?)
4. *Optional:* Try some other queries for each table.