Complex Data in Practice

While there are several things you may want to do with complex data, here are three examples of practical applications. The techniques used for these applications are not immediately apparent from understanding basic queries, and they must be handled differently between Hive and Impala, so look for the new information provided here!

Counting Items in a Collection

ARRAYs and MAPs can contain any number of items; they do not have a fixed size. You can use the size function in a Hive query to return the number of items in an ARRAY or MAP, but Impala does not have such a function. The examples here show how to find the number of items in both engines.

Using Hive

Using the size function with Hive is fairly straightforward:

```
SELECT name, size(phones) AS num
    FROM customers_phones_array;
```

For this example, the column named phones is an ARRAY column. Using our example data from the previous readings, the ARRAY in each row of this column contains a different number of items. In the row for Alice, the ARRAY has three items; in the row for Bob, the ARRAY has one item; and in the row for Carlos, the ARRAY has two items.

The query uses the size function to return these numbers of items as a column with the alias num. Similarly, when you use the size function with a MAP column, it returns the numbers of key-value pairs. The size function is an example of what's called a *collection function*, because it's a function that operates on a *collection type*, which is another name for a complex type.

Using Impala

As noted above, Impala does not have a size function, nor does Impala support any other collection functions. To count the number of elements in each ARRAY or MAP using Impala, you need to use join notation and a GROUP BY clause to group by a column or columns that have unique row values. You can then use the COUNT function, or indeed any other aggregation function, to aggregate the elements in each ARRAY or MAP.

```
SELECT name, COUNT(*) AS num

    FROM phones_array_parquet, phones_array_parquet.phones

    GROUP BY name;
```

In this example, the goal is to return each customer's name and the number of phone numbers they have. In the SELECT list, the expression COUNT(*) AS num returns a column named num giving the number of records in each group, which is the number of phone numbers each customer has.

ConvertingARRAYs and MAPs to Records with Hive

Recall that Impala's method of querying ARRAY and MAP types provides a separate row for each element in the complex column, and you must use a join to include values from the other columns in the table. Hive's behavior is very different, whichmay seem unusual, because typically Impala aims for a high degree of compatibility with Hive's query syntax. In this case, Impala's syntax is intended to provide greater flexibility.

If you want to use Hive to break the individual items within an ARRAY or MAP into a table of results with one item per row, you can do this using the explode function. The explode function is an example of what's called a *table-generating* function; this is a class of functions that can transform a single input row into multiple output rows.

```
SELECT explode(phones) AS phone
    FROM customers_phones_array;
```

In this example, the explode function is applied to the ARRAY column named phones, and it returns a column with the alias phone. It returns one output record for each item in the ARRAY column. Using the same example data as in the previous readings, there are a total of six phone numbers in the ARRAY column (three for Alice, one for Bob, and two for Carlos). This means the output contains six records.

Since the MAP type has two parts to it, the key and the value, explode returns two values. You can use the same syntax *except*for the alias—if you use an alias, you must supply two values:

```
SELECT explode(phones) AS (type, number)
    FROM customers_phones_map;
```

The result set would again have six rows, each with two columns: type and number. If you omit the alias, the columns would be called key and value.

When you use a table-generating function like explode, you cannot include any other columns in the SELECT list of your query. However, you can overcome this limitation by using a *lateral view*, which first applies the table-generating function to the ARRAY or MAP column, then joins the resulting output with the rows of the table. Lateral view syntax is similar to explicit join syntax; in the FROM clause, include the name of the base table, then the keywords LATERAL VIEW, followed by the explode function applied to the ARRAY or MAP column.

```
SELECT name, phone
    FROM customers_phones_array
    LATERAL VIEW
```

```
    explode(phones) p AS phone;
```

In the example here, a lateral view is used to return values from the name column along with the individual phone numbers. *The table alias, p in this example, is required,* even if you don't use it anywhere else in the query.The column alias, AS phone in this example, is optional.

For the MAP version, again you need two column aliases—but without parentheses in this case:

```
SELECT name, type, number
    FROM customers_phones_map
    LATERAL VIEW
        explode(phones) p AS type, number;
```

Denormalizing Tables Using Complex Data

A potential use for complex data is denormalizing tables with a one-to-many relationship. Normalized tables (using Third Normal Form) cannot have a repeating groups—that is, a single row should not have multiple values for one type of data. For example, a toy company likely has many products; a table listing different toy makers (like the makers table in the toy database) would not include all Hasbro's products in the same row.

Instead, you typically have a second table in which each row holds one of those values along with a foreign key that identifies which row in the first table that value belongs with. In the toy company in example, the toys table has one row for each toy, and the maker_id column identifies which company from the makers table makes that particular toy. Joining the columns using the maker's identification number allows you to identify all the toys made by a particular company.

The complex column types allow analysts to reshape tables into a denormalized form. The resulting revised data model can support queries of the combined data elements from one table, without any join required. In big data, you can expect a query that does not require a join to be significantly faster than one that does require a join.

The rest of this section describes how to set up and query such a table, using the toy example. You don't need to memorize these steps or the functions involved.

Creating the Denormalized Table

The makers_with_toys table defined below could hold each toy maker with its information, *including* a list of its toys (and the MSRP, manufacturer's suggested retail price) in the same row, using an ARRAY with STRUCTs as its elements. (The table uses Parquet files so you can query it with Impala. You can create this table in either Hive or Impala.)

```
CREATE EXTERNAL TABLE toy.makers_with_toys (

    id INT,

    name STRING,

    city STRING,

    toys ARRAY<STRUCT <toy_name:STRING, price:DECIMAL(5,2)>>)

  STORED AS PARQUET;
```

Populating the Denormalized Table

The next step is to load the data into the table. Because Impala can't load data into Parquet files, this step *must* be completed with Hive.

Use the named_struct function to cast a row of the detail table (in this case, toys) into the STRUCT. Then use the collect_list function to collect multiple rows into the ARRAY. (These functions are probably new to you, because none of the courses in this specialization have introduced them before now. Consult the Hive documentation if you want to learn more about these functions.)

```
INSERT OVERWRITE TABLE toy.makers_with_toys
    SELECT m.id, m.name, m.city,
        collect_list(named_struct('toy_name', t.name,
                                  'price', t.price))
    FROM toy.makers m LEFT OUTER JOIN toy.toys t
        ON (m.id = t.maker_id)
    GROUP BY m.id, m.name, m.city;
```

Querying the Denormalized Table

You now can query the table with Hive or Impala, using the syntax for the engine you're using. For example, you can find the price of the most expensive toy for each maker using the following query with Impala. Remember to invalidate metadata before attempting to run this query with Impala:

```
SELECT name, MAX(toys.item.price) AS max_price
    FROM toy.makers_with_toys, toy.makers_with_toys.toys
```

```
    GROUP BY name;
```

Note that to query the elements in the ARRAY column (toys), you need to reference the pseudocolumn toys.item as the column, but that element is a STRUCT. So you then need to use .price to identify the element within that STRUCT.

Try It!

Try each of the examples above, using both the ARRAY and MAP tables when appropriate.