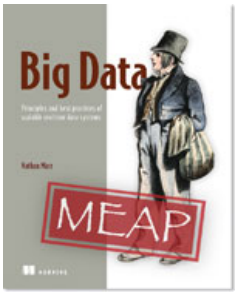
MANNING PUBLICATIONS



Big Data
Principles and best practices of scalable realtime data systems
By Nathan Marz

The power of these abstractions is in how they promote reuse and composability. In this article based on chapter 5, author Nathan Marz discusses various composition techniques possible with JCascalog.

You may also be interested in…

Composition Techniques with JCascalog

There are three main composition techniques we'll look at: predicate macros, functions that return dynamically created subqueries, and functions that return dynamically created predicate macros. These techniques take advantage of the fact that there's no barrier between the query tool and your general purpose programming language. You're able to manipulate your queries in a very fine-grained way.

Predicate macros

A predicate macro is a predicate operation that expands to another set of predicates. You've already seen one example of a predicate macro in the definition of Average from the beginning of this chapter. Let's take a look at that definition again:

```
public static PredicateMacroTemplate Average =
  PredicateMacroTemplate.build("?val").out("?avg")
        .predicate(new Count(), "?count")
        .predicate(new Sum(), "?val").out("?sum")
        .predicate(new Div(), "?sum", "?count").out("?avg");
```

The first line of the definition says that this operation takes in one variable called `?val` as input, and produces one variable called `?avg` as output. `Average` then consists of three predicates composed together: a count aggregation, a sum aggregation, and a division function. The intermediate variables `?count` and `?sum` are used to store the results of the aggregation before dividing them to compute the result.

Here's an example usage of `Average`:

```
new Subquery("?result")
      .predicate(INTEGER, "?n")
      .predicate(Average, "?n").out("?result");
```

When JCascalog sees that a predicate is a predicate macro, it expands it into its constituent predicates, like so:

```
new Subquery("?result")
      .predicate(INTEGER, "?n")
      .predicate(new Count(), "?count_gen1")
      .predicate(new Sum(), "?n").out("?sum_gen2")
      .predicate(new Div(), "?sum_gen2", "?count_gen1", "?result");
```

You can see that the `?n` and `?result` variables from the `Subquery` replace the `?val` and `?avg` variables used in definition of Average. And the `?count` and `?sum` intermediate vars are given unique names to guarantee they don't conflict with any other variables used in the surrounding `Subquery`.

This definition of `Average` is done as a PredicateMacroTemplate, which makes it easy to specify simple compositions like `Average` by specifying the composition as a template. Not everything can be specified with a template, however. For example, suppose you wanted to specify an aggregator that computes the unique count of a set of variables, like so:

```
new Subquery("?unique-followers-count")
      .predicate(FOLLOWS, "?person", "_")
      .predicate(new DistinctCount(), "?person")
          .out("?unique-followers-count");
```

Now suppose that you want this aggregator to work even if the number of tuples for a group is extremely large, large enough that it would cause memory issues to keep an in-memory set to compute the unique count. What you can do is make use of a feature called "secondary sorting" that can sort your group before it's given as input to your aggregator function. Then, to compute the distinct count, your code just needs to keep track of the last tuple seen in the group to determine whether or not to increment the count or not. The code to do the sorting and aggregation looks like this:

```
public static class DistinctCountAgg extends CascalogAggregator {
    static class State {
        int count = 0;
        Tuple last = null;
    }

    public void start(FlowProcess process, AggregatorCall call) {
        call.setContext(new State());
    }

    public void aggregate(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        Tuple t = call.getArguments().getTupleCopy();
        if(s.last==null || !s.last.equals(t)) {
            s.count++;
        }
        s.last = t;
    }

    public void complete(FlowProcess process, AggregatorCall call) {
        State s = (State) call.getContext();
        call.getOutputCollector().add(new Tuple(s.count));
    }
}

public static Subquery distinctCountManual() {
    return new Subquery("?unique-followers-count")
          .predicate(FOLLOWS, "?person", "_")
          .predicate(Option.SORT, "?person")
          .predicate(new DistinctCountAgg(), "?person")
              .out("?unique-followers-count");
}
```

`DistinctCountAgg` contains the logic to compute the unique count given a sorted input, and the `Option.SORT` predicate tells JCascalog how to sort the tuples for each group.

Of course, you don't want to have to specify the sort and aggregator each time you want to do a distinct count. What you want to do is compose these predicates together into a single abstraction. However, you can't do this with a PredicateMacroTemplate, like we used for Average, since any number of variables could be used for the distinct count.

The most general form of a predicate macro is as a function that takes in a list of input fields, a list of output fields, and returns a set of predicates. Here's the definition of `DistinctCount` as a regular `PredicateMacro`:

```
public static class DistinctCount implements PredicateMacro {
    public List<Predicate> getPredicates(Fields inFields,
            Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
```

```
        ret.add(new Predicate(Option.SORT, inFields));
        ret.add(new Predicate(new DistinctCountAgg(),
                              inFields, outFields));
        return ret;
    }
}
```

The input fields and output fields come from what is specified when the `PredicateMacro` is used within a subquery. Predicate macros are powerful because everything in JCascalog is represented uniformly as predicates. Predicate macros can thus arbitrarily compose predicates together, whether they're aggregators, secondary sorts, or functions.

Dynamically created subqueries

One of the most common techniques when using JCascalog is to write functions that create subqueries dynamically. That is, you write regular Java code that constructs a subquery according to some parameters.

This technique is powerful because subqueries can be used like any other source of data—they are simply a source of tuples, like how files on HDFS are a source of tuples. So you can use subqueries to break apart a larger query into subqueries, with each subquery handling an isolated portion of the overall computation.

Suppose, for example, you have text files on HDFS representing transaction data: an ID for the buyer, an ID for the seller, a timestamp, and a dollar amount.

The data is JSON-encoded and looks like this:

```
{"buyer": 123, "seller": 456, "amt": 50, "timestamp": 1322401523}
{"buyer": 1009, "seller": 12, "amt": 987, "timestamp": 1341401523}
{"buyer": 2, "seller": 98, "amt": 12, "timestamp": 1343401523}
```

You may have a variety of computations you want to run on that data, such as number of transactions within a time period, the total amount sold by each seller, or the total amount bought by each buyer. Each of these queries needs to do the same work of parsing the data from the text files, so you'd like to abstract that into its own subquery. What you need is a function that takes in a path on HDFS and returns a subquery that parses the data at that path. You can write this function like this:

```
public static class ParseTransactionRecord extends CascalogFunction {
    public void operate(FlowProcess process, FunctionCall call) {
        String line = call.getArguments().getString(0);
        Map parsed = (Map) JSONValue.parse(line);
        call.getOutputCollector().add(
            new Tuple(
                parsed.get("buyer"),
                parsed.get("seller"),
                parsed.get("amt"),
                parsed.get("timestamp")
                ));
    }
}

public static Subquery parseTransactionData(String path) {
    return new Subquery("?buyer", "?seller", "?amt", "?timestamp")
            .predicate(Api.hfsTextline(path), "?line")
            .predicate(new ParseTransactionRecord(), "?line")
                .out("?buyer", "?seller", "?amt", "?timestamp");
}
```

You can then reuse this abstraction for each query. For example, here's the query which computes the number of transactions for each buyer:

```
public static Subquery buyerNumTransactions(String path) {
    return new Subquery("?buyer", "?count")
            .predicate(parseTransactionData(path),
                "?buyer", "_", "_", "_")
            .predicate(new Count(), "?count");
```

```
}
```

This is a very simple example of creating subqueries dynamically, but it illustrates how subqueries can be composed together in order to enable abstracting away pieces of a more complicated computation. Let's look at another example in which the number of predicates is dynamic according to the arguments.

Suppose you have a set of retweet data, with each record indicating that some tweet is a retweet of some other tweet. You want to be able to query for all chains of retweets of a certain length. So for a chain of length 4, you want to know all retweets of retweets of retweets of tweets.

You start with pairs of tweet identifiers. The basic observation needed for transforming pairs into chains is to recognize that you can find chains of length 3 by joining pairs against themselves. Then you can find chains of length 4 by joining your chains of length 3 against your original pairs. For example, here's a query that returns chains of length 3 given an input generator of "pairs":

```
public static Subquery chainsLength3(Object pairs) {
    return new Subquery("?a", "?b", "?c")
            .predicate(pairs, "?a", "?b")
            .predicate(pairs, "?b", "?c");
}
```

And here's how you find chains of length 4:

```
public static Subquery chainsLength4(Object pairs) {
    return new Subquery("?a", "?b", "?c", "?d")
            .predicate(pairs, "?a", "?b")
            .predicate(pairs, "?b", "?c")
            .predicate(pairs, "?c", "?d");
}
```

To generalize this for chains of arbitrary length, you need a function that generates a subquery, setting up the appropriate number of predicates and setting the variable names correctly. This can be accomplished by writing some fairly simple Java code to create the subquery:

```
public static Subquery chainsLengthN(Object pairs, int n) {
    List<String> genVars = new ArrayList<String>();
    for(int i=0; i<n; i++) {
        genVars.add(Api.genNullableVar());
    }
    Subquery ret = new Subquery(genVars);
    for(int i=0; i<n-1; i++) {
        ret = ret.predicate(pairs, genVars.get(i), genVars.get(i+1));
    }
    return ret;
}
```

The function first generates unique variable names to be used within the query using the helper `Api.getNullableVar` function from JCascalog. It then iterates through the variables, creating predicates to set up the joins that will return the appropriate chains. Another interesting note about this function is that it's not specific to retweet data: in fact, it can take as input any subquery or source of data containing pairs and return a subquery that computes chains.

Let's look at one more example of a dynamically created subquery. Suppose you want to be able to get a random sample of some fixed number of elements from any set of data. For instance, you want to get a random 10000 elements out of a dataset of unknown size.

The simplest strategy to accomplish this in a distributed and scalable way is with the following algorithm:

1. Generate a random number for every record
2. Find the N elements with the smallest random numbers

JCascalog has a built-in aggregator called "Limit" for doing #2: Limit uses a strategy similar to parallel aggregators to find the smallest N elements. "Limit" will find the smallest N elements on each map task, then combine all the results from each map task to find the overall smallest N elements.

Here's a function that uses Limit to find a random N elements from a dataset of arbitrary size:

```
public static Subquery fixedRandomSample(Object data, int n) {
    List<String> inputVars = new ArrayList<String>();
    List<String> outputVars = new ArrayList<String>();
    for(int i=0; i < Api.numOutFields(data); i++) {
        inputVars.add(Api.genNullableVar());
        outputVars.add(Api.genNullableVar());
    }
    String randVar = Api.genNullableVar();
    return new Subquery(outputVars)
            .predicate(data, inputVars)
            .predicate(new RandLong(), randVar)
            .predicate(Option.SORT, randVar)
            .predicate(new Limit(n), inputVars).out(outputVars);
}
```

The function uses the `Api.numOutfields` introspection function to determine the number of fields in the input dataset. It then generates variables so that it can put together a subquery that represents the logic to do a fixed random sample. The `RandLong` function comes with JCascalog and simply generates a random long value. The secondary sort tells the Limit aggregator how to determine what the smallest elements are, and then Limit does the heavy lifting of finding the smallest records.

The cool thing about this algorithm is its scalability: it's able to parallelize the computation of the fixed sample without ever needing to centralize all the records in one place. And it was easy to express the algorithm, since using regular Java code you can construct a subquery to do the fixed sample for any input set of data.

Dynamically created predicate macros

Like how you can write functions that dynamically create subqueries, you can also create predicate macros dynamically. This is an extremely powerful technique that really showcases the advantages of having your query tool just be a library for your general purpose programming language.

Consider the following query:

```
new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new IncrementFunction(), "?a").out("?x")
    .predicate(new IncrementFunction(), "?b").out("?y")
    .predicate(new IncrementFunction(), "?c").out("?z");
```

This query reads a dataset containing triplets of numbers, and increments each field. There's some repetition in this query, since it has to explicitly apply the `IncrementFunction` to each field from the input data. It would be nice if you could eliminate this repetition, like so:

```
new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new Each(new IncrementFunction()), "?a", "?b", "?c")
            .out("?x", "?y", "?z");
```

Rather than repeat the use of `IncrementFunction` over and over, it's better if you could tell JCascalog to apply the function to each input/output field pair, automatically expanding that predicate to the three predicates in the first example.

This is exactly what predicate macros are good at, and you can define `Each` quite simply like so:

```
public static class Each implements PredicateMacro {
    Object _op;

    public Each(Object op) {
```

```
        _op = op;
    }

    public List<Predicate> getPredicates(Fields inFields,
            Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        for(int i=0; i<inFields.size(); i++) {
            Object in = inFields.get(i);
            Object out = outFields.get(i);
            ret.add(new Predicate(_op, Arrays.asList(in),
                    Arrays.asList(out)));
        }
        return ret;
    }
}
```

This definition of `Each` is parameterized with the predicate operation to use, and then it creates a predicate for each input/output field pair that it's given. The number of predicates generated is dynamic depending on the number of fields specified in the subquery predicate.

Let's look at another example of a dynamic subquery. We've already defined the `IncrementFunction`, which adds the value one to its argument. We defined `IncrementFunction` as its own function, but, in reality, it's really just the `Plus` function with one argument filled in to "1". It would be nice if you could abstract away the partial application of a predicate operation into its own operation, and defined Increment as something like this:

```
public static Object Increment = new Partial(new Plus(), 1);
```

`Partial` is a predicate macro that fills in some of the input fields. It allows you to rewrite the query that increments the triplets like this:

```
new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new Each(new Partial(new Plus(), 1)),
            "?a", "?b", "?c").out("?x", "?y", "?z");
```

After expanding all the predicate macros, this query expands to:

```
new Subquery("?x", "?y", "?z")
    .predicate(TRIPLETS, "?a", "?b", "?c")
    .predicate(new Plus(), 1, "?a").out("?x")
    .predicate(new Plus(), 1, "?b").out("?y")
    .predicate(new Plus(), 1, "?c").out("?z");
```

The definition of `Partial` is straightforward:

```
public static class Partial implements PredicateMacro {
    Object _op;
    List<Object> _args;

    public Partial(Object op, Object... args) {
        _op = op;
        _args = Arrays.asList(args);
    }

    public List<Predicate> getPredicates(Fields inFields,
            Fields outFields) {
        List<Predicate> ret = new ArrayList<Predicate>();
        List<Object> input = new ArrayList<Object>();
        input.addAll(_args);
        input.addAll(inFields);
        ret.add(new Predicate(_op, input, outFields));
        return ret;
    }
}
```

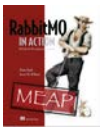
The predicate macro simply prepends any provided input fields to the input fields specified when the subquery is created.

Summary

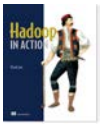
As you can see, dynamic predicate macros let you manipulate the construction of your subqueries in a very fine-grained way. In all of these composition techniques—predicate macros, dynamic subqueries, and dynamic predicate macros—regular Java code is used to create clean, reusable abstractions.

Here are some other Manning titles you might be interested in:

MongoDB in Action
Kyle Banker



RabbitMQ in Action
Alvaro Videla and Jason J.W. Williams



Hadoop in Action
Chuck Lam

Last updated: August 5, 2012