Data Design Patterns


```
@author: Mahmoud Parsian
@email: mahmoud.parsian@yahoo.com
@last updated: 1/10/2022
```


The goal of this paper/chapter is to present Data Design Patterns in an informal way. The emphasis has been on pragmatism and practicality.

Data Design Patterns can be categorized as:

- Summarization Patterns
- Filtering Patterns
- Organization Patterns
- Join Patterns
- Meta Patterns
- Input and Output Patterns

1. Summarization Patterns

Typically, Numerical Summarizations are big part of Summarization Patterns. Numerical summarizations are patterns, which involves calculating aggregate statistical values (minimum, maximum, average, median, standard deviation, ...) over data. If data has keys (such as department identifier, gene identifiers, or patient identifiers), then the goal is to group records by a key field and then you calculate aggregates per group such as minimum, maximum, average, median, or standard deviation. If data does not have keys, then you compute the summarization over entire data without grouping.

The main purpose of summarization patters is to summarize lots of data into meaningful data structures such as tuples, lists, and dictionaries.

Some of the Numerical Summarizations patterns can be expressed by SQL.

For example, Let `gene_samples` be a table of `(gene_id, patient_id, biomarker_value)`. Further, assume that we have about

`100,000` unique `gen_id` (s), a patient (represented as `patient_id`) may have lots of gene records with an associated `biomarker_value` (s).

This Numerical Summarizations pattern corresponds to using `GROUP BY` in SQL for example:

```
SELECT MIN(biomarker_value), MAX(biomarker_value), COUNT(*)
   FROM gene_samples
      GROUP BY gene_id;
```

Therefore, in this example, we find a triplet `(min, max, count)` per `gene_id`.

In Spark, this summarization pattern can be implemented by using RDDs and DataFrames. In Spark, `(key, value)` pair RDDs are commonly used to `group by` a key (in our example `gene_id`) in order to calculate aggregates per group.

Let's assume that the input files are CSV file(s) and further assume that input records have the following format:

```
<gene_id><,><patient_id><,><biomarker_value>
```

RDD Example using groupByKey()

We load data from CSV file(s) and then create an `RDD[(key, value)]`, where key is `gene_id` and value is a `biomarker_value`. To solve it by the `reduceByKey()`, we first need to map it to a desired data type of `(min, max, count)`:

```
# rdd : RDD[(gene_id, biomarker_value)]
mapped = rdd.mapValues(lambda v: (v, v, 1))
```

Then we can apply `groupByKey()` transformation:

```
# grouped : RDD[(gene_id, Iterable<biomarker_value>)]
grouped = mapped.groupByKey()
# calculate min, mx, count for values
triplets = grouped.mapValues(lambda values: (min(values), max(values), len(values)
```

The `groupByKey()` might give OOM errors if you have too many values per key (`gene_id`) and `groupByKey()` does not use any combiners at all. Overall

`reduceByKey()` is a better scale-out solution than `groupByKey()`.

RDD Example using reduceByKey()

We load data from CSV file(s) and then create an `RDD[(key, value)]`, where key is `gene_id` and value is a `biomarker_value`. To solve it by the `reduceByKey()`, we first need to map it to a desired data type of `(min, max, count)`:

```
# rdd : RDD[(gene_id, biomarker_value)]
mapped = rdd.mapValues(lambda v: (v, v, 1))
```

Then we can apply `reduceByKey()` transformation:

```
# x = (min1, max1, count1)
# y = (min2, max2, count2)
reduced = mapped.reduceByKey(lambda x, y: (min(x[0], y[0]), max(x[1], y[1]), x[2]+
```

Spark's `reduceByKey()` merges the values for each key using an associative and commutative reduce function.

RDD Example using combineByKey()

We load data from CSV file(s) and then create an `RDD[(key, value)]`, where key is `gene_id` and value is a `biomarker_value`.

`RDD.combineByKey(createCombiner, mergeValue, mergeCombiners)` is a generic function to combine the elements for each key using a custom set of aggregation functions. Turns an `RDD[(K, V)]` into a result of type `RDD[(K, C)]`, for a "combined type" C.

Users provide three functions:

```
1. createCombiner, which turns a V into a C (e.g., creates a one-element list)
2. mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
3. mergeCombiners, to combine two C's into a single one (e.g., merges the lists)
```

here is the solution by `combineByKey()`:

```
# rdd : RDD[(gene_id, biomarker_value)]
combined = rdd.combineByKey(
   lambda v: (v, v, 1),
   lambda C, v: (min(C[0], v), max(C[1], v), C[2]+1),
   lambda C, D: (min(C[0], D[0]), max(C[1], D[1]), C[2]+D[2])
)
```

DataFrame Example

After reading input, we can create a DataFrame as: `DataFrame[(gene_id, patient_id, biomarker_value)]`.

```
# df : DataFrame[(gene_id, patient_id, biomarker_value)]
import pyspark.sql.functions as F
result = df.groupBy("gene_id")
          .agg(F.min("biomarker_value").alias("min"),
               F.max("biomarker_value").alias("max"),
               F.count("biomarker_value").alias("count")
              )
```

The other alternative solution is to use pure SQL: register your DataFrame as a table, and then fire a SQL query:

```
# register DataFrame as gene_samples
df.registerTempTable("gene_samples")
# find the result by SQL query:
result = spark.sql("SELECT MIN(biomarker_value),
                           MAX(biomarker_value),
                           COUNT(*)
                      FROM gene_samples
                         GROUP BY gene_id")
```

Note that your SQL statement will be executed as a series of mappers and reducers behind the Spark engine.

Data Without Keys - using DataFrames

You might have some numerical data without keys and then you might be interested in computing some statistics such as (min, max, count) on the entire data. In these situations, we

have more than couple of options: we can use `mapPartitions()` transformation or use `reduce()` action (depending on the format and nature of input data).

We can use Spark's built in functions to get aggregate statistics. Here's how to get mean and standard deviation.

```
# import required functions
from pyspark.sql.functions import col
from pyspark.sql.functions import mean as _mean
from pyspark.sql.functions import stddev as _stddev

# apply desired functions
collected_stats = df.select(
    _mean(col('numeric_column_name')).alias('mean'),
   _stddev(col('numeric_column_name')).alias('stddev')
).collect()

# extract the final results:
final_mean = collected_stats[0]['mean']
final_stddev = collected_stats[0]['stddev']
```

Data Without Keys - using RDDs

If you have to filter your numeric data and perform other calculations before omputing mean and std-dev, then you may use `RDD.mapPartitions()` transformations. The `RDD.mapPartitions(f)` transformation returns a new RDD by applying a function `f()` to each partition of this RDD. Finally, you may `reduce()` the result of `RDD.mapPartitions(f)` transformation.

To understand `RDD.mapPartitions(f)` transformation, let's assume that your input is a set of files, where each record has a set of numbers separated by comma (note that each record may have any number of numbers: for example one record may have 5 numbers and another record might have 34 numbers, etc.):

```
<number><,><number><,>...<,><number>
```

Suppose the goal is to find `(min, max, count, num_of_negatives, num_of_positives)` for the entire data set. One easy solution is to use `RDD.mapPartitions(f)`, where `f()` is a function which returns

`(min, max, count, num_of_negatives, num_of_positives)` per partition. Once `mapPartitions()` is done, then we can apply the final reducer to find the final `(min, max, count, num_of_negatives, num_of_positives)` for all partitions.

Let `rdd` denote `RDD[String]`, which represents all input records.

First we define our custom function `compute_stats()`, which accepts a partition and returns `(min, max, count, num_of_negatives, num_of_positives)` for a given partition.

```
def count_neg_pos(numbers):
    neg_count, pos_count = 0, 0
    # iterate numbers
    for num in numbers:
      if num > 0: pos_count += 1
      if num < 0: neg_count += 1
   #end-for
   return (neg_count, pos_count)
#end-def

def compute_stats(partition):
    first_time = True
    for e in partition:
        numbers = [int(x) for x in e.split(',') if x]
        neg_pos = count_neg_pos(numbers)
        #
        if (first_time):
            _min = min(numbers)
            _max = max(numbers)
            _count = len(numbers)
            _neg = neg_pos[0]
            _pos = neg_pos[1]
            first_time = False
        else:
            # it is not the first time:
            _min = min(_min, min(numbers))
            _max = max(_max, max(numbers))
            _count += len(numbers)
            _neg += neg_pos[0]
            _pos += neg_pos[1]
        #end-if
    #end-for
    return [(_min, _max, _count, _neg, _pos)]
#end-def
```

After defining `compute_stats(partition)` function, we can now apply the `mapPartitions()` transformation:

```
mapped = rdd.mapPartitions(compute_stats)
```

Now `mapped` is an `RDD[(int, int, int, int, int)]`

Next, we can apply the final reducer to find the final `(min, max, count, num_of_negatives, num_of_positives)` for all partitions:

```
tuple5 = mapped.reduce(lambda x, y: (min(x[0], y[0]),
                                     max(x[1], y[1]),
                                     x[2]+y[2],
                                     x[3]+y[3],
                                     x[4]+y[4])
                       )
```

Spark is so flexiable and powerful: therefore we might find multiple solutions for a given problem. But what is the optimal solution? This can be handled by testing your solutions against real data which you might use in the production environments. Test, test, and test.

2. Filtering Patterns

Filter patterns are a set of design patterns that enables us to filter a set of records (or elements) using different criteria and chaining them in a decoupled way through logical operations. One simple example will be to filter records if the salary of that record is less than 20000. Another example would be to filter records if the record does not contain a valid URL. This type of design pattern comes under structural pattern as this pattern combines multiple criteria to obtain single criteria.

for example, Python offers filtering as:

`filter(function, sequence)` where

`function` : function that tests if each element of a sequence true or not.

`sequence` : sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.

Returns: returns an iterator that is already filtered.

Simple example is given below:

```
# function that filters DNA letters
def is_dna(variable):
    dna_letters = ['A', 'T', 'C', 'G']
    if (variable in dna_letters):
        return True
    else:
        return False
#end-def

# sequence
sequence = ['A', 'B', 'T', 'T', 'C', 'G', 'M', 'R', 'A']

# using filter function
# filtered = ['A', 'T', 'T', 'C', 'G', 'A']
filtered = filter(is_dna, sequence)
```

PySpark offers filtering in large scale for RDDs and DataFrames.

Filter using RDD

Let `rdd` be an `RDD[(String, Integer)]`. Assume the goal is to keep (key, value) pairs if and only if the value is greater than 0. It is pretty straightforward to accomplish this in PySpark: by using the `RDD.filter()` transformation:

```
# rdd: RDD[(String, Integer)]
# filtered: RDD[(key, value)], where value > 0
# e = (key, value)
filtered = rdd.filter(lambda e: e[1] > 0)
```

Also, the filter implementation can be done by boolean predicate functions:

```
def greater_than_zero(e):
    # e = (key, value)
    if e[1] > 0:
        return True
    else:
        return False
#end-def

# filtered: RDD[(key, value)], where value > 0
filtered = rdd.filter(greater_than_zero)
```

Filter using DataFrame

Filtering records using DataFrame can be accomplished by `DataFrame.filter()` or you may use `DataFrame.where()`.

Consider a `df` as a `DataFrame[(emp_id, city, state)]`.

Then you may use the following as filtering patterns:

```
# SparkSession available as 'spark'.
>>> tuples3 = [('e100', 'Cupertino', 'CA'), ('e200', 'Sunnyvale', 'CA'),
               ('e300', 'Troy', 'MI'), ('e400', 'Detroit', 'MI')]
>>> df = spark.createDataFrame(tuples3, ['emp_id', 'city', 'state'])
>>> df.show()
+------+---------+-----+
|emp_id|     city|state|
+------+---------+-----+
|  e100|Cupertino|   CA|
|  e200|Sunnyvale|   CA|
|  e300|     Troy|   MI|
|  e400|  Detroit|   MI|
+------+---------+-----+

>>> df.filter(df.state != "CA").show(truncate=False)
+------+-------+-----+
|emp_id|city   |state|
+------+-------+-----+
|e300  |Troy   |MI   |
|e400  |Detroit|MI   |
+------+-------+-----+

>>> df.filter(df.state == "CA").show(truncate=False)
```

```
+------+---------+-----+
|emp_id|city     |state|
+------+---------+-----+
|e100  |Cupertino|CA   |
|e200  |Sunnyvale|CA   |
+------+---------+-----+

>>> from pyspark.sql.functions import col
>>> df.filter(col("state") == "MA").show(truncate=False)
+------+----+-----+
|emp_id|city|state|
+------+----+-----+
+------+----+-----+

>>> df.filter(col("state") == "MI").show(truncate=False)
+------+-------+-----+
|emp_id|city   |state|
+------+-------+-----+
|e300  |Troy   |MI   |
|e400  |Detroit|MI   |
+------+-------+-----+
```

You may also use `DataFrame.where()` function to filter rows:

```
>>> df.where(df.state == 'CA').show()
+------+---------+-----+
|emp_id|     city|state|
+------+---------+-----+
|  e100|Cupertino|   CA|
|  e200|Sunnyvale|   CA|
+------+---------+-----+
```

For more examples, you may read PySpark Where Filter Function | Multiple Conditions.

3. Organization Patterns

in progress...

4. Join Patterns

in progress...

5. Meta Patterns

Meta data is about a set of data that describes and gives information about other data. Meta patterns is about "patterns that deal with patterns". The term meta patterns is directly translated to "patterns about patterns." For example in MapReduce paradigm, "job chaining" is a meta pattern, which is piecing together several patterns to solve complex data problems. In MapReduce paradigm, another met pattern is "job merging", which is an optimization for performing several data analytics in the same MapReduce job, effectively executing multiple MapReduce jobs with one job.

Spark is a superset of MapReduce paradiagm and deals with meta patterns in terms of estimators, transformers and pipelines, which are discussed here:

- ML Pipelines
- Want to Build Machine Learning Pipelines?

6. Input/Output Patterns

in progress...

References

1. Spark with Python (PySpark) Tutorial For Beginners
2. Data Algorithms with Spark, author: Mahmoud Parsian
3. PySaprk Algorithms, author: Mahmoud Parsian
4. Apache PySpark Documentation
5. PySpark Tutorial, author: Mahmoud Parsian