Data Design Patterns

Mahmoud Parsian

Ph.D. in Computer Science

Senior Architect @Illumina

Adjunct faculty @Santa Clara University

email: mahmoud.parsian@yahoo.com

last updated: 1/17/2022

Note: this is working document...

0. Introduction

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 formats:

- PDF version
- Marcdown version

Source code for Data Design Patterns are provided in GitHub.

Data Design Patterns can be categorized as:

- Summarization Patterns
- In-Mapper-Combiner Pattern
- 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]+y[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. Note that depending on your data requirements, combined data type can be a simple data type (such as integer, string, ...) or it can be collection (such as set, list, tuple, array, or dictionary) or it can be custom data type.

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. In-Mapper-Combiner Design Pattern

In this section, I will discuss In-Mapper-Combiner design patterns and show some examples for using the pattern.

In a typical MapReduce paradigm, mappers emit `(key, value)` pairs and once all mappers are done, the "sort and shuffle" phase prepare inputs (using output of mappers) for reducers in the form of `(key, Iterable<value>)`, finally, reducers consume these pairs and create final `(key, aggregated-vale)`. For example if mappers have emitted the following `(key, vlaue)` pairs:

```
(k1, v1), (k1, v2), (k1, v3), (k1, v4),
(k2, v5), (k2, v6)
```

Then "sort and shuffle" will prepare the following `(key, vlaue)` pairs to be consumed by reducers:

```
(k1, [v1, v2, v3, v4])
(k2, [v5, v6])
```

For some data applications, it is very possible to emit too many (key, value) pairs, which may create huge network traffic in the cluster. If a mapper creates the same key multiple times (with different values) for the input partition, then for some aggregation algorithms, it is possible to aggregate/combine these `(key, values)` and emit less `(key, value)` pairs. The simplest case is that counting DNA bases (count A's, T's, C's, G's). In a typical MapReduce paradigm, for a DNA string of "AAAATTTCCCGGAAATGG", a mapper will create the following `(key, value)` pairs:

```
(A, 1), (A, 1), (A, 1), (A, 1), (T, 1), (T, 1), (T, 1),
(C, 1), (C, 1), (C, 1), (G, 1), (G 1), (A, 1), (A, 1),
(A, 1), (T, 1), (G, 1), (G, 1)
```

For this specific algorithm (DNA base count), it is possible to combine values for the same key:

```
(A, 7), (T, 4), (C, 3), (G, 4)
```

Combining/merging/reducing 18 `(key, value)` pairs into 4 combined `(key, value)` pairs is called "In-Mapper-Combining": we combined values in the mapper processing: the advantage is we emit/create much less `(key, value)` pairs, which will ease the cluster network traffic. The "In-Mapper Combiner" emits `(A, 7)` instead of 7 pairs of `(A, 1)` and so on. The In-Mapper-Combiner design pattern is introduced to address some issues (to limit the number of `(key, value)` pairs generated by mappers) with MapReduce programming paradigm.

When do you need In-Mapper-Combiner design pattern? When your mapper generates too many `(key, value)` pairs and you have a chance to combine these `(key, value)` pairs into a smaller number of `(key, value)` pairs, then you may use In-Mapper-Combiner design pattern.

Informally, say that your mapper has created 3 keys with multiple `(key, value)` as:

```
key k1: (k1, u1), (k1, u2), (k1, u3), ...
key k2: (k2, v1), (k2, v2), (k2, v3), ...
key k3: (k3, t1), (k3, t2), (k3, t3), ...
```

Then In-Mapper-Combiner design pattern should combine these into the following `(key, value)` pairs:

```
(k1, combiner_function([u1, u2, u3, ...])
(k2, combiner_function([v1, v2, v3, ...])
(k3, combiner_function([t1, t2, t3, ...])
```

Where `combiner_function([a1, a2, a3, ...])` is a custom function, which combines/reduces `[a1, a2, a3, ...]` into a single value.

Applying In-Mapper-Combiner design pattern may result in a more efficient algorithm implementation from the performance point (for example reducing time complexity). The `combiner_function()` must guarantee that it is a semantic-preserving function: meaning that semantic/correctness of algorithms (with and without In-Mapper-Combiner) for mappers must not change at all. The In-Mapper-Combiner design pattern might substantially reduce both the number and size of `(key, value)` pairs that need to be shuffled from the mappers to the reducers.

Example: DNA Base Count Problem

What is a DNA Base Counting? The four bases in DNA molecule are adenine (A), cytosine (C), guanine (G), and thymine (T). So, a DNA String is comprised of 4 base letters {A, T, C, G}. DNA Base Count finds frequency of base letters for a given set of DNA strings.

Input Format: FASTA

There are multiple text formats to represent DNA. The FASTA is a text based format to represent DNA data. The FASTA file format is a widely used format for specifying biosequence information. A sequence in FASTA format begins with a single description line, followed by one or more lines of sequence data.

Therefore, a FASTA file has 2 kind of records:

- records which begins with ">", which is a description line (should be ignored for DNA base count)
- records which does not begin with ">", which is a DNA string

We will ignore the description records and focus only on DNA strings.

Example of two FASTA-formatted sequences in a file:

```
>NM_012514 Rattus norvegicus breast cancer 1 (Brca1), mRNA
CGCTGGTGCAACTCGAAGACCTATCTCCTTCCCGGGGGGGCTTCTCCGGCATTTAGGCCT
CGGCGTTTGGAAGTACGGAGGTTTTTCTCGGAAGAAAGTTCACTGGAAGTGGAAGAAATG
GATTTATCTGCTGTTCGAATTCAAGAAGTACAAAATGTCCTTCATGCTATGCAGAAAATC
TTGGAGTGTCCAATCTGTTTGGAACTGATCAAAGAACCGGTTTCCACACAGTGCGACCAC
ATATTTTGCAAATTTTGTATGCTGAAACTCCTTAACCAGAAGAAAGGACCTTCCCAGTGT
CCTTTGTGTAAGAATGAGATAACCAAAAGGAGCCTACAAGGAAGTGCAAGG
>NM_012515
TGTGGATCTTTCCAGAACAGCAGTTGCAATCACTATGTCTCAATCCTGGGTACCCGCCGT
GGGCCTCACTCTGGTGCCCAGCCTGGGGGGCTTCATGGGAGCCTACTTTGTGCGTGGTGA
GGGCCTCCGCTGGTATGCTAGCTTGCAGAAACCCTCCTGGCATCCGCCTCGCTGGACACT
CGCTCCCATCTGGGGCACACTGTATTCGGCCATGGGGTATGGCTCCTACATAATCTGGAA
AGAGCTGGGAGGTTTCACAGAGGAGGCTATGGTTCCCTTGGGTCTCTACACTGGTCAGCT
```

Note that for all three solutions, we will drop description records (which begins with ">" symbol) by using the `RDD.filter()` transformation.

Solution 1: Classic MapReduce Algorithm

In the canonical example of DNA Base counting, a `(key, value)` pair is emitted for every DNA base letter found, where key is a DNA base letter in {A, T, C, G} and value is 1 (frequency of one). This solution will create too many `(key, value)` pairs. After mapping is done, then we have several options for reduction of these `(key, value)` pairs. The options are:

- Use `groupByKey()`
- Use `reduceByKey()`
- Use `combineByKey()`

Pros of Solution 1

- Simple solution, which works
- Mappers are fast, no need for combining counters

Cons of Solution 1

- Too many `(key, value)` pairs are created
- Might cause cluster network traffic

Solution 2: In-Mapper-Combiner Algorithm

In this solution we will use In-Mapper-Combiner design pattern and per DNA string, we will emit at most four `(key, value)` pairs as:

```
(A, n1)
(T, n2)
(C, n3)
(G, n4)
```

where n1: is the total frequency of A's per mapper input n2: is the total frequency of T's per mapper input n3: is the total frequency of C's per mapper input n4: is the total frequency of G's per mapper input

To implement In-Mapper-Combiner design pattern, we will use Python's `collections.Counter()` to keep track of DNA base letter frequencies. The other option is to use four variables (initialized to zero), and then increment them as we interate/scan the DNA string. Since the number of keys are very small (4 of them), then it is easier to use 4 variables for counting, otherwise (when you have many keys) you should use a `collections.Counter()` to keep track of frequencies of keys.

Similar to the first solutions, we mat apply any of the following reducers to find the final DNA base count.

- Use `groupByKey()`
- Use `reduceByKey()`
- Use `combineByKey()`

Pros of Solution 2

- Much less `(key, value)` pairs are created compared to Solution 1
- In-Mapper-Combiner design pattern is applied
- Will not cause cluster network traffic, since there are not too manny `(key, value)`

Cons of Solution 2

- A dictionary is created per mapper, if we have too many mappers concurrently, then there might be an OOM error

Solution 3: Mapping Partitions Algorithm

This solution uses `RDD.mapPartitions()` transformation to solve DNA base count problem. In this solution we will emit four `(key, value)` pairs as:

```
(A, p1)
(T, p2)
(C, p3)
(G, p4)
```

where

```
p1: is the total frequency of A's per single partition
p2: is the total frequency of T's per single partition
p3: is the total frequency of C's per single partition
p4: is the total frequency of G's per single partition
```

Note that a single partition may have thousands or millions of FASTA records. For this solution, we will create a single `collections.Counter()` per partition (rather than per RDD element)

Pros of Solution 3

- Much less `(key, value)` pairs are created compared to Solution 1 and 2
- Map Partitions design pattern is applied
- Will not cause cluster network traffic, since there are not too manny `(key, value)`
- A single dictionary is created per partition. Since the number of partitions can be in hundereds or thousands, this will not be a problem at all
- This is the most scaled-out solution: basically we summarize DNA base counting per partition: from each partition, we emit four `(key, value)` pairs

Cons of Solution 3

- None

Summary and conclusion

In-Mapper-Combiner design pattern is one method to summarize the output of mappers and hence to possibly improve the speed of your MapReduce job by reducing the number of intermediary `(key, value)` pairs emitted from mappers to reducers. As we noted, there are several ways to implement In-Mapper-Combiner design pattern, which does depend on your mappers input and expected output. One immediate benefit of In-Mapper-Combiner design pattern is to drastically reduce the number of `(key, value)` pairs emitted from mappers to reducers.

Download FASTA Files

1. GeoSymbio Downloads
2. NCBI Downloads

3. 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.

4. Organization Patterns

The Organizational Patterns deals with reorganizing data to be used by other rendering applications. For example, you might have structured data in different formats and different data sources and you might join and merge these data into XML or JSON formats. The other example will be partition data (so called binning pattern) based on some categories (such as continent, country, ...).

4.1 The Structured to Hierarchical Pattern

The goal of this pattern is to convert structured data (in different formats and from different data sources) into hierarchical (XML or JSON) structure. You need to bring all data into a sigle location, so that you can convert it to hierarchical structure. In a nutshell, The structured to hierarchical pattern creates new hierarchical records (such as XML, JSON) from data that started in a very different structure (plain records). The main objective of the Structured to Hierarchical Pattern is to transform row-based data to a hierarchical format (such as JSON or XML).

For example, consider blog data commented by many users. A hierarchy will look something like:

```
    Posts
        Post-1
            Comment-11
            Comment-12
            Comment-13
        Post-2
            Comment-21
            Comment-22
            Comment-23
            Comment-24
        ...
```

Assume that there are two types of structured data, which can be joind to create A hierarchal structure (above).

Data Set 1:

```
<post_id><,><title><,><creator>
```

Example of Data Set 1 records:

```
p1,t1,creator1
p2,t2,creator2
p3,t3,creator3
...
```

Data Set 2:

```
<post_id><,><comment><,><commented_by>
```

Example of Data Set 2 records:

```
p1,comment-11,commentedby-11
p1,comment-12,commentedby-12
p1,comment-13,commentedby-13
p2,comment-21,commentedby-21
p2,comment-22,commentedby-22
p2,comment-23,commentedby-23
p2,comment-24,commentedby-24
...
```

Therefore, the goal is to join-and merge these 2 data sets so that we can create an XML for a single post such as:

```
<post id="p1">
    <title>t1</title>
    <creator>creator1</creator>
    <comments>
        <comment>comment-11</comment>
        <comment>comment-12</comment>
        <comment>comment-13</comment>
    </comments>
</post>
<post id="p2">
    <title>t2</title>
    <creator>creator2</creator>
    <comments>
        <comment>comment-21</comment>
        <comment>comment-22</comment>
        <comment>comment-23</comment>
        <comment>comment-24</comment>
    </comments>
</post>
...
```

I will provide two solutions: RDD-based and DataFrame-based.

RDD Solution

Step-1: This solution reads data sets and creates two RDDs with `post_id` as a key:

```
posts: RDD[(post_id, (title, creator))]
comments: RDD[(post_id, (comment, commented_by))]
```

Step-2: these two RDDs are joined by common key: `post_id`:

```
# joined: RDD[(post_id, ((title, creator),(comment, commented_by)))]
joined = posts.join(comments)
```

Step-3: Apply a reducer: group by `post_id`:

```
# grouped = RDD[(post_id, Iterable<((title, creator),(comment, commented_by))>)]
grouped = joined.groupByKey()
```

Step-4: the final step is iterate `grouped` elements and then create XML or JSON

```
xml_rdd = grouped.map(create_xml)

where

# element: (post_id, Iterable<((title, creator),(comment, commented_by))>)
def create_xml(element):
    xml = <perform concatenation of required items and create desired xml>
    return xml
#end-def
```

Complete example implementation is given as:
`structured_to_hierarchical_to_xml_rdd.py`

DataFrame Solution

In DataFrame solution, we read data sets and create DataFrames.

```
posts = DataFrame(["post_id", "title", "creator"])
comments = DataFrame(["post_id", "comment", "commented_by"])
```

Next, these two Dataframes are joined on common key `post_id` and then we select proper columns:

```
joined_and_selected = posts.join(comments, posts.post_id == comments.post_id)\
    .select(posts.post_id, posts.title, posts.creator, comments.comment)
```

Next, we group the result by ("post_id", "title", "creator") and concatenate `comment` column values.

```
grouped = joined_and_selected.groupBy("post_id", "title", "creator")
  .agg(F.collect_list("comment").alias("comments"))
```

To create XML, we use a UDF:

```
create_xml_udf = F.udf(lambda post_id, title, creator, comments:
    create_xml(post_id, title, creator, comments), StringType())
```

Finally, we apply UDF to proper columns to create XML:

```
df = grouped.withColumn("xml", \
   create_xml_udf(grouped.post_id, grouped.title, grouped.creator, grouped.comment
  .drop("title")\
  .drop("creator")\
  .drop("comments")
```

Complete example implementation is given as:
`structured_to_hierarchical_to_xml_dataframe.py`

4.2 The partitioning and binning patterns

Bucketing, binning, and categorization of data are used synonymously in technical papers and blogs. Data binning, also called discrete binning or bucketing, is a data pre-processing technique used to reduce the effects of minor observation errors. Binning is a way to group a number of more or less continuous values into a smaller number of "bins". For example, if you have data about a group of graduated students, with a number of years in education, then you might categorize it as HSDG (12 years), AA (14 years), BS (16 years), MS (18 years), PHD (21 years), MD (22+ years). Therefore, you have created 6 bins: `{HSDG, AA, BS, MS, PHD, MD}`. After bins are created, then you might use categorial values (HSDG, BS, ...) in your data queries.

Data binning -- also called Discrete binning or bucketing -- is a data pre-processing technique used to reduce the effects of minor observation errors. The original data values which fall into a given small interval, a bin, are replaced by a value representative of that interval, often the central value. For example if a car price value os so scattered, then you may use bucketing instead of actual car prices.

Spark's Bucketizer transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users.

Consider this example: there's no linear relationship between latitude and the housing values, but you may suspect that individual latitudes and housing values are related, but the relationship is not linear. Therefore you might bucketize the latitudes; for example you may create buckets as:

```
Bin-1:  32 < lattitude <= 33
Bin-2:  33 < lattitude <= 34
...
```

Binning technique can be applied on both categorical and numerical data. The following examples show both types of binning.

Numerical Binning Example:

value	Bin
0-10	Very Low
11-30	Low
31-70	Mid
71-90	High
91-100	Very High

Categorical Binning Example

value	Bin
India	Asia
China	Asia
Japan	Asia
Spain	Europe
Italy	Europe
Chile	South America
Brazil	South America

Binning is used genomics data as well: we bucketize human genome chromosomes (1, 2, 3, ..., 22, X, Y, MT). For instance chromosomes 1 has 250 million positions, which we may bucketize into 101 buckets as:

```
for id in (1, 2, 3, ..., 22, X, Y, MT):
  chr_position = (chromosome-<id> position)
  # chr_position range is from 1 to 250,000,000
  bucket = chr_position % 101
  # where
  #      0 =< bucket <= 100
```

Bucketing is a most straight forward approach for converting the continuous variables into categorical variable. To understand this, let's look at an example below. In PySpark the task of bucketing can be easily accomplished using the `Bucketizer` class.

To use the `Bucketizer` class, firstly, we shall accomplish the task of creating bucket borders. Let us define a list of bucket borders as the following example. Next, let us create a object of the `Bucketizer` class. Then we will apply the `transform` method to our defined Dataframe `dataframe` .

First, Let's create a sample dataframe for demo purpose:

```
>>> data = [('A', -99.99), ('B', -0.5), ('C', -0.3),
...   ('D', 0.0), ('E', 0.7), ('F', 99.99)]
>>> column_names =  ["id", "features"]
>>> dataframe = spark.createDataFrame(data, column_names)
>>> dataframe.show()
+---+--------+
| id|features|
+---+--------+
|  A|  -99.99|
|  B|    -0.5|
|  C|    -0.3|
|  D|     0.0|
|  E|     0.7|
|  F|   99.99|
+---+--------+
```

Next, we apply the `Bucketizer` to create buckets:

```
>>> bucket_borders=[-float("inf"), -0.5, 0.0, 0.5, float("inf")]
>>> from pyspark.ml.feature import Bucketizer
>>> bucketer = Bucketizer().setSplits(bucket_borders)
     .setInputCol("features").setOutputCol("bucket")
>>> bucketer.transform(dataframe).show()
+---+--------+------+
| id|features|bucket|
+---+--------+------+
|  A|  -99.99|   0.0|
|  B|    -0.5|   1.0|
|  C|    -0.3|   1.0|
|  D|     0.0|   2.0|
|  E|     0.7|   3.0|
|  F|   99.99|   3.0|
+---+--------+------+
```

5. Join Patterns

Covered in chapter 11 of Data Algorithms with Spark

6. 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?

7. Input/Output Patterns

in progress...

8. 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

6. J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce