Spark Syntax

Eric Xiao

Table of Contents

Spark-Syntax

This is a public repo documenting all of the "best practices" of writing PySpark code from what I have learnt from working with `PySpark` for 3 years. This will mainly focus on the `Spark DataFrames and SQL` library.

Contributing/Topic Requests

If you notice an improvements in terms of typos, spellings, grammar, etc. feel free to create a PR and I'll review it , you'll most likely be right.

If you have any topics that I could potentially go over, please create an **issue** and describe the topic. You can create an issue here. I'll try my best to address it .

Acknowledgement

If you found this book helpful, please give a star on the github repo to show some love!

Huge thanks to Levon for turning everything into a gitbook. You can follow his github at https://github.com/tumregels.

Other Formats

- pdf
- epub
- mobi

Spark API Documents

I always find myself referencing the `PySpark API` documentation and have it opened as a seperate browser at work. A majority of your Spark application will be written with the functions found in the document.

It can be found with this link (I suggest you bookmark it):

PySpark latest API docs

Ask Google

When in doubt ask Google, there are a lot of crowdsourced questions and answers on Stack Overflow.

Companies that Contribute to Spark

Databricks and **Cloudera** contribute heavily to Spark and they provide a lot of good blogs about writting performant Spark code.

Note: The author of Spark, Matei Zaharia also cofounded Databricks the company.

Conference Talks

There are a lot of Spark conferences throughout the year where speakers from the companies above or the big tech companies come speak about their advances and experiences with Spark at scale. I find these talks very insightful into writing "real big data" applications. These talks also cover a broader subject matter like how to manage a large spark cluster, etc.

Example: Apache Spark - Spark + AI Summit San Francisco 2018

Spark Books

The O'reilly books on Spark is how I got into Spark. They are either written by some highly profiled people in the Spark community (Holden Karau) or the original members that created Spark back in the AmpLabs days (Matei Zaharia).

The two that I would recommend are:

- Learning Spark: Lightning-Fast Big Data Analysis
    - Back in the early days of Spark, this was the only book out there. I started off with this book. It gives a nice overview of everything in Spark.
    - It might be a bit outdated but none-the-less it will give you an appreciation for how far Spark has come.
    - This is written by Holden Karau and Matei Zahario most noticably.
- Spark: The Definitive Guide: Big Data Processing Made Simple

- This book is more up-to-date as it talks more in-depth about `Spark SQL` and the `DataFrame`s API.
- This book is written by Matei Zahario most noticably.

Spark Release Docs

Spark is an open-source project under Apache, and releases new features regularly. If you want to be up-to-date with the newest features I recommend following their releases/news:

Spark Releases

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Create a DataFrame

```
schema = T.StructType([
    T.StructField("pet_id", T.IntegerType(), False),
    T.StructField("name", T.StringType(), True),
    T.StructField("age", T.IntegerType(), True),
])

data = [
    (1, "Bear", 13),
    (2, "Chewie", 12),
    (2, "Roger", 1),
]

pet_df = spark.createDataFrame(
    data=data,
    schema=schema
)

pet_df.toPandas()
```

	pet_id	name	age
0	1	Bear	13
1	2	Chewie	12
2	2	Roger	1

Background

There are 3 datatypes in spark `RDD`, `DataFrame` and `Dataset`. As mentioned before, we will focus on the `DataFrame` datatype.

- This is most performant and commonly used datatype.
- `RDD`s are a thing of the past and you should refrain from using them unless you can't do the transformation in `DataFrame`s.
- `Dataset`s are a thing in `Spark scala`.

If you have used a `DataFrame` in Pandas, this is the same thing. If you haven't, a dataframe is similar to a `csv` or `excel` file. There are columns and rows that you can perform transformations on. You can search online for better descriptions of what a `DataFrame` is.

What Happened?

For any `DataFrame (df)` that you work with in Spark you should provide it with 2 things:

1. a `schema` for the data. Providing a `schema` explicitly makes it clearer to the reader and sometimes even more performant, if we can know that a column is `nullable`. This means providing 3 things:
   - the `name` of the column
   - the `datatype` of the column
   - the `nullability` of the column
2. the data. Normally you would read data stored in `gcs`, `aws` etc and store it in a `df`, but there will be the off-times that you will need to create one.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T
```

The above also shows you the "best practices" for importing these components into your program.

*some of the above imports will be explained later, just know this is how you should import these functions into your Spark application.

These are the essential `imports` that you will need for any `PySpark` program.

`SparkSession`
The `SparkSession` is how you begin a Spark application. This is where you provide some configuration for your Spark program.

`pyspark.sql.functions`
You will find that all your data wrangling/analysis will mostly be done by chaining together multiple `functions`. If you find that you get your desired transformations with the base functions, you should:

1. Look through the API docs again.
2. Ask Google.
3. Write a `user defined function` (`udf`).

`pyspark.sql.types`
When working with spark, you will need to define the type of data for each column you are working with.

The possible types that Spark accepts are listed here: Spark types

Hello World

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 3 - Reading your First Dataset")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Create a `SparkSession`. No need to create `SparkContext` as you automatically get it as part of the `SparkSession`.

Read in Data (CSV)

```
# define the structure of your data inside the CSV file
```

```
def get_csv_schema(*args):
    return T.StructType([
        T.StructField(*arg)
        for arg in args
    ])

# read in your csv file with enforcing a schema
def read_csv(fname, schema):
    return spark.read.csv(
        path=fname,
        header=True,
        schema=get_csv_schema(*schema)
    )
```

```
import os

data_path = "/data"
pets_path = "/pets.csv"
base_path = os.path.dirname(os.getcwd())

path = base_path + data_path + pets_path
df = read_csv(
    fname=path,
    schema=[
        ("id", T.LongType(), False),
        ("breed_id", T.LongType(), True),
        ("name", T.StringType(), True),
        ("birthday", T.TimestampType(), True),
        ("color", T.StringType(), True)
    ]
)
```

```
df.toPandas()
```

	id	species_id	name	birthday	color
0	1	1	King	2014-11-22 12:30:31	brown
1	2	3	Argus	2016-11-22 10:05:10	None

What Happened?

Here we read in a `csv` file and put it into a `DataFrame (DF)`: this is one of the three datasets that Spark allows you to use. The other two are `Resilient Distributed Dataset (RDD)` and `Dataset`. `DF`s have replaced `RDD`s as more features have been brought out in version `2.x` of Spark. You should be able to perform anything with `DataFrames` now, if not you will have to work with `RDD`s, which I will not cover.

Spark gives you the option to automatically infer the schema and types of columns in your dataset. But you should always specify a `schema` for the data that you're reading in. For each column in the `csv` file we specified:

- the `name` of the column
- the `data type` of the column
- if `null` values can appear in the column

Conclusion

Congratulations! You've read in your first dataset in Spark. Next we'll look at how you can perform transformations on this dataset :).

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 4 - More Comfortable with SQL?")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path

df = spark.read.csv(path, header=True)
df.toPandas()
```

	id	species_id	name	birthday	color
0	1	1	King	2014-11-22 12:30:31	brown
1	2	3	Argus	2016-11-22 10:05:10	None

Register DataFrame as a SQL Table

```
df.createOrReplaceTempView("pets")
```

What Happened?

The first step in making a `df` queryable with `SQL` is to **register** the table as a sql table.

This particular function will **replace** any previously registered **local** table named `pets` as a result. There are other functions that will register a dataframe with slightly different behavior. You can check the reference docs if this isn't the desired behavior: docs

Let Write a SQL Query!

```
df_2 = spark.sql("""
SELECT
    *
FROM pets
WHERE name = 'Argus'
""")

df_2.toPandas()
```

	id	species_id	name	birthday	color
0	2	3	Argus	2016-11-22 10:05:10	None

What Happened?

Once your `df` is registered, call the spark `sc` function on your `spark session` object. It takes a `sql string` as an input and outputs a new `df`.

Conclusion?

If you're more comfortable with writing `sql` than python/spark code, then you can do so with a spark `df`! We do this by:

1. Register the `df` with `df.createOrReplaceTempView('table')`.
2. Call the `sql` function on your `spark session` with a `sql string` as an input.
3. You're done!

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 1.1 - Struct Types")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+------+
| id|breed_id|nickname|           birthday|age|color|weight|
+---+--------+--------+-------------------+---+-----+------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|  10.0|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|   5.5|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|   3.4|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|
+---+--------+--------+-------------------+---+-----+------+
```

Struct Types

What are they used for? TODO

```
(
    pets
```

```
    .withColumn('struct_col', F.struct('nickname', 'birthday', 'age', 'color'))
    .withColumn('nickname_from_struct', F.col('struct_col').nickname)
    .show()
)
```

```
+---+--------+--------+-------------------+---+-----+------+--------------------+---------
-----------+
| id|breed_id|nickname|           birthday|age|color|weight|          struct_col|nickname_
from_struct|
+---+--------+--------+-------------------+---+-----+------+--------------------+---------
-----------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|  10.0|[King, 2014-11-22...|
       King|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|   5.5|[Argus, 2016-11-2...|
      Argus|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|[Chewie, 2016-11-...|
     Chewie|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|   3.4|[Maple, 2018-11-2...|
      Maple|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|[, 2019-01-01 10:...|
       null|
+---+--------+--------+-------------------+---+-----+------+--------------------+---------
-----------+
```

What Happened?

We created a `struct` type column consisting of the columns `'nickname', 'birthday', 'age', 'color'` . Then we accessed a member `nickname` from the struct.

Summary

- TODO: Fix a use-case.
- It is pretty easy creating and accessing `struct` datatypes.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.1.2 - Arrays and Lists")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

def get_csv_schema(*args):
    return T.StructType([
        T.StructField(*arg)
        for arg in args
    ])

def read_csv(fname, schema):
    return spark.read.csv(
        path=fname,
        header=True,
        schema=get_csv_schema(*schema)
    )

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = read_csv(
    fname=path,
    schema=[
        ("id", T.LongType(), False),
        ("breed_id", T.LongType(), True),
        ("nickname", T.StringType(), True),
        ("birthday", T.TimestampType(), True),
        ("age", T.LongType(), True),
        ("color", T.StringType(), True),
        ("weight", T.DecimalType(), True),
```

```
    ]
)
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+------+
| id|breed_id|nickname|           birthday|age|color|weight|
+---+--------+--------+-------------------+---+-----+------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|    10|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|     6|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|     3|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|
+---+--------+--------+-------------------+---+-----+------+
```

Arrays and Lists

Case 1: Reading in Data that contains `Arrays`

TODO

Case 2: Creating Arrays

```
(
    pets
    .withColumn('array column', F.array([
        F.lit(1),
        F.lit("Bob"),
        F.lit(datetime(2019,2,1)),
    ]))
    .show()
)
```

```
+---+--------+--------+-------------------+---+-----+------+--------------------+
| id|breed_id|nickname|           birthday|age|color|weight|        array column|
+---+--------+--------+-------------------+---+-----+------+--------------------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|    10|[1, Bob, 2019-02-...|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|     6|[1, Bob, 2019-02-...|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|[1, Bob, 2019-02-...|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|     3|[1, Bob, 2019-02-...|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|[1, Bob, 2019-02-...|
+---+--------+--------+-------------------+---+-----+------+--------------------+
```

What Happened?

We will explain in the later chapter what the `F.lit()` function does, but for now understand that in order to create an array type you need to call the `F.array()` function and for each array element call `F.lit()` on.

Summary

- It's pretty simple to create an array in Spark, you will need to call 2 functions: `F.array()` and `F.lit()`.
- Each element of the array needs to be of type `F.lit()`

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 1.3 - Maps and Dictionaries")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

def get_csv_schema(*args):
    return T.StructType([
        T.StructField(*arg)
        for arg in args
    ])

def read_csv(fname, schema):
    return spark.read.csv(
        path=fname,
        header=True,
        schema=get_csv_schema(*schema)
    )

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+------+
| id|breed_id|nickname|           birthday|age|color|weight|
+---+--------+--------+-------------------+---+-----+------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|  10.0|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|   5.5|
```

```
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|   3.4|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|
+---+--------+--------+-------------------+---+-----+------+
```

Maps and Dictionaries

Case 1: Creating a Mapping from Existing Columns

```
(
    pets
    .fillna({
        'nickname': 'Unknown Name',
        'age':      'Unknown Age',
    })
    .withColumn('{nickname:age}', F.create_map(F.col('nickname'), F.col('age')))
    .withColumn('{nickname:age} 2', F.create_map('nickname', 'age'))
    .show()
)
```

```
+---+--------+-----------+-------------------+---+-----+------+--------------------+-----
---------------+
| id|breed_id|    nickname|           birthday|age|color|weight|      {nickname:age}|    {
nickname:age} 2|
+---+--------+-----------+-------------------+---+-----+------+--------------------+-----
---------------+
|  1|       1|        King|2014-11-22 12:30:31|  5|brown|  10.0|         [King -> 5]|
    [King -> 5]|
|  2|       3|       Argus|2016-11-22 10:05:10| 10| null|   5.5|       [Argus -> 10]|
  [Argus -> 10]|
|  3|       1|      Chewie|2016-11-22 10:05:10| 15| null|    12|      [Chewie -> 15]|
 [Chewie -> 15]|
|  3|       2|       Maple|2018-11-22 10:05:10| 17|white|   3.4|       [Maple -> 17]|
  [Maple -> 17]|
|  4|       2|Unknown Name|2019-01-01 10:05:10| 13| null|    10|[Unknown Name -> 13]|[Unkn
own Name -> 13]|
+---+--------+-----------+-------------------+---+-----+------+--------------------+-----
---------------+
```

What Happened?

You can create a column of map types using either `columnary expressions` (we'll learn what column expressions are later) or column names.

Case 2: Creating a Mapping from Constant Values

```
(
    pets
    .fillna({
        'nickname': 'Unknown Name',
```

```
        'age':      'Unknown Age',
    })
    .withColumn('{nickname:age}', F.create_map(F.lit('key'), F.lit('value')))
    .show()
)
```

```
+---+--------+------------+-------------------+---+-----+------+--------------+
| id|breed_id|    nickname|           birthday|age|color|weight|{nickname:age}|
+---+--------+------------+-------------------+---+-----+------+--------------+
|  1|       1|        King|2014-11-22 12:30:31|  5|brown|  10.0|[key -> value]|
|  2|       3|       Argus|2016-11-22 10:05:10| 10| null|   5.5|[key -> value]|
|  3|       1|      Chewie|2016-11-22 10:05:10| 15| null|    12|[key -> value]|
|  3|       2|       Maple|2018-11-22 10:05:10| 17|white|   3.4|[key -> value]|
|  4|       2|Unknown Name|2019-01-01 10:05:10| 13| null|    10|[key -> value]|
+---+--------+------------+-------------------+---+-----+------+--------------+
```

What Happened?

You can create a column of map types of literals using the `columnary expression` `F.lit()`, we will learn this later on. Notice that each key/value needs to be a `columnal expression`? This will be a common theme throughout Spark.

Summary

- It is very simple to create map data in Spark.
- You can do so with both existing columns or constant values.
- If constant values are used, then each value must be a `columnary expression`.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 1.4 - Decimals and Why did my Decimals overflow")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

def get_csv_schema(*args):
    return T.StructType([
        T.StructField(*arg)
        for arg in args
    ])

def read_csv(fname, schema):
    return spark.read.csv(
        path=fname,
        header=True,
        schema=get_csv_schema(*schema)
    )

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

Decimals and Why did my Decimals overflow

Some cases where you would deal with `Decimal` types are if you are talking about money, height, weight, etc. Working with `Decimal` types may appear simple at first but there are some nuances that will sneak up behind you. We will go through some ways to get around these as they are hard to debug.

Here is some simple jargon that we will use in the following examples:

- `Integer` : The set of numbers including all the whole numbers and their opposites (the positive whole numbers, the negative whole numbers, and zero). ie. -1, 0, 1, 2, etc.
- `Irrational Number` : The set including all numbers that are non- terminating, non- repeating decimals. ie. 2.1, 10.5, etc.
- `Precision` : the maximum total number of digits.
- `Scale` : the number of digits on the right of dot.

Source:

- link
- link

Case 1: Working With `Decimal` s in Python

```
print("Example 1 - {}".format(Decimal(20)))
print("Example 2 - {}".format(Decimal("20.2")))
print("Example 3 - {}".format(Decimal(20.5)))
print("Example 4 - {}".format(Decimal(20.2)))
```

```
Example 1 - 20
Example 2 - 20.2
Example 3 - 20.5
Example 4 - 20.199999999999992894572642398998141288757324218 75
```

What Happened?

Let's break down the examples above.

Example 1:

Here we provided a whole number, so nothing special.

Example 2:

Here we provided a string representing an irrational number. The `precision` and `scale` were preserved.

Example 3:

Here we provided an irrational number. The `precision` and `scale` were preserved.

Example 4:

Here we provided an irrational number, but this isn't what we expected? Well this is because it's impossible to provide an exact representation of `20.2` on the computer! If you want to know more about this you can look up "IEEE floating point representation". We will keep this in mind for later on.

IEEE Floating Point Represenatation (OUT OF SCOPE OF HANDBOOK)

When you were little have you ever tried to divide 10 by 3, did the result ever terminate? No. You were left with 3.33333... This is a similar case with trying to represent some `irrational numbers` on a computer.

> "There's only 10 types of people in the world: those who get binary and those who don't."

Let's think about how a computer works, if we go down to the lowest level, everything is stored as a binary value either a 0 or 1. Every whole number is easily representable, `11` is 3, `101` is 5, etc. But when it comes to `irrational numbers` there has been extensive work and established standards to representing these numbers most correctly. One of these standards are called the `IEEE-754 32-bit Single-Precision Floating-Point Numbers`.

IEEE-754 32-bit Single-Precision Floating-Point Numbers

From the picture above we can see that to represent a single irrational or floating point number, you only have 32 bits (or 64 in other cases) and a set number of bits for each portion of the number. Only 1 bit is ever needed for the `signed` bit, then the number of bits will vary for the whole (exponent) number and then decimal fractional) values.

I won't show you how the whole number and floating point values are computed, you can refer to other references for this. For example, this is a very good example: link.

Let's look at our examples above:

example	signed	exponent	fraction
20.2	0	010000011	01000011001100110011010
20.5	0	010000011	01001000000000000000000

If you did the calculations for `20.2` you will notice that the fraction portion never actually divides nicely. Like I mentioned with the simply example above with trying to divide 10 by 3 nicely, it's impoosible you will always have some leftover/remainder values. But with `20.5` we can see that it divides nicely.

Case 2: Reading in Decimals in Spark (Incorrectly)

```
pets = read_csv(
    fname=path,
    schema=[
        ("id", T.LongType(), False),
        ("breed_id", T.LongType(), True),
        ("nickname", T.StringType(), True),
        ("birthday", T.TimestampType(), True),
        ("age", T.LongType(), True),
        ("color", T.StringType(), True),
        ("weight", T.DecimalType(), True),
    ]
)
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+------+
| id|breed_id|nickname|           birthday|age|color|weight|
+---+--------+--------+-------------------+---+-----+------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|    10|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|     6|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|    12|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|     3|
|  4|       2|    null|2019-01-01 10:05:10| 13| null|    10|
+---+--------+--------+-------------------+---+-----+------+
```

What Happened?

What happened to our `scalar` values, they weren't read in? This is because the default arguments to the `T.Decimal()` function are `DecimalType(precision=10, scale=0)`. So to read in the data correctly we need to override these default arguments.

Case 2: Reading in Decimals in Spark (Correctly)

```
pets = read_csv(
    fname=path,
    schema=[
        ("id", T.LongType(), False),
        ("breed_id", T.LongType(), True),
        ("nickname", T.StringType(), True),
        ("birthday", T.TimestampType(), True),
        ("age", T.LongType(), True),
        ("color", T.StringType(), True),
        ("weight", T.DecimalType(10,2), True),
    ]
)
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+------+
| id|breed_id|nickname|           birthday|age|color|weight|
+---+--------+--------+-------------------+---+-----+------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown| 10.00|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|  5.50|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null| 12.00|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|  3.40|
|  4|       2|    null|2019-01-01 10:05:10| 13| null| 10.00|
+---+--------+--------+-------------------+---+-----+------+
```

Case 3: Reading in Large Decimals in Spark

```
spark.createDataFrame(
    data=[
        (100,),
        (2 ** 63,)
```

```
    ],
    schema=['data']
).show()
```

```
+----+
|data|
+----+
| 100|
|null|
+----+
```

What Happened?

Why is the second value null? The second value overflows the max value of a decimal and never makes it to Spark (Scala).

If you see this error then you will need to check your input data as there might be something wrong there.

Case 3: Setting Values in a DataFrame (Incorrectly)

```
(
    pets
    .withColumn('decimal_column', F.lit(Decimal(20.2)))
    .show()
)
```

```
---------------------------------------------------------------------------

AnalysisException                         Traceback (most recent call last)

<ipython-input-7-14e51ddcba87> in <module>()
      1 (
      2     pets
----> 3     .withColumn('decimal_column', F.lit(Decimal(20.2)))
      4     .show()
      5 )


/usr/local/lib/python2.7/site-packages/pyspark/sql/functions.pyc in _(col)
     40     def _(col):
     41         sc = SparkContext._active_spark_context
---> 42         jc = getattr(sc._jvm.functions, name)(col._jc if isinstance(col, Column) e
lse col)
     43         return Column(jc)
     44     _.__name__ = name


/usr/local/lib/python2.7/site-packages/py4j/java_gateway.pyc in __call__(self, *args)
   1255         answer = self.gateway_client.send_command(command)
   1256         return_value = get_return_value(
```

```
-> 1257             answer, self.gateway_client, self.target_id, self.name)
   1258
   1259         for temp_arg in temp_args:


/usr/local/lib/python2.7/site-packages/pyspark/sql/utils.pyc in deco(*a, **kw)
     67                                             e.java_exception.getStackTrace()))
     68             if s.startswith('org.apache.spark.sql.AnalysisException: '):
---> 69                 raise AnalysisException(s.split(': ', 1)[1], stackTrace)
     70             if s.startswith('org.apache.spark.sql.catalyst.analysis'):
     71                 raise AnalysisException(s.split(': ', 1)[1], stackTrace)


AnalysisException: u'DecimalType can only support precision up to 38;'
```

What Happened?

Remember our python examples above? Well because the `precision` of the Spark `T.DecimalType` is 38 digits, the value went over the maximum value of the Spark type.

Case 3: Setting Values in a DataFrame (Correctly)

```
(
    pets
    .withColumn('decimal_column', F.lit(Decimal("20.2")))
    .show()
)
```

```
+---+--------+--------+-------------------+---+-----+------+--------------+
| id|breed_id|nickname|           birthday|age|color|weight|decimal_column|
+---+--------+--------+-------------------+---+-----+------+--------------+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown| 10.00|          20.2|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|  5.50|          20.2|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null| 12.00|          20.2|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|  3.40|          20.2|
|  4|       2|    null|2019-01-01 10:05:10| 13| null| 10.00|          20.2|
+---+--------+--------+-------------------+---+-----+------+--------------+
```

What Happened?

If we provide the irrational number as a string, this solves the problem.

Case 4: Performing Arthimetrics with `DecimalType` s (Incorrectly)

```
pets = spark.createDataFrame(
    data=[
        (Decimal('113.790000000000000000'), Decimal('2.54')),
        (Decimal('113.790000000000000000'), Decimal('2.54')),
    ],
```

```
    schema=['weight_in_kgs','conversion_to_lbs']
)

pets.show()
```

```
+--------------------+--------------------+
|       weight_in_kgs|   conversion_to_lbs|
+--------------------+--------------------+
|113.7900000000000...|2.540000000000000000|
|113.7900000000000...|2.540000000000000000|
+--------------------+--------------------+
```

```
(
    pets
    .withColumn('weight_in_lbs', F.col('weight_in_kgs') * F.col('conversion_to_lbs'))
    .show()
)
```

```
+--------------------+--------------------+-------------+
|       weight_in_kgs|   conversion_to_lbs|weight_in_lbs|
+--------------------+--------------------+-------------+
|113.7900000000000...|2.540000000000000000|   289.026600|
|113.7900000000000...|2.540000000000000000|   289.026600|
+--------------------+--------------------+-------------+
```

What Happened?

This used to overflow... Guess they updated it .

Case 4: Performing Arthimetrics Operations with DecimalTypes (Correctly)

```
(
    pets
    .withColumn('weight_in_kgs', F.col('weight_in_kgs').cast('Decimal(20,2)'))
    .withColumn('conversion_to_lbs', F.col('conversion_to_lbs').cast('Decimal(20,2)'))
    .withColumn('weight_in_lbs', F.col('weight_in_kgs') * F.col('conversion_to_lbs'))
    .show()
)
```

```
+-------------+-----------------+-------------+
|weight_in_kgs|conversion_to_lbs|weight_in_lbs|
+-------------+-----------------+-------------+
|       113.79|             2.54|     289.0266|
|       113.79|             2.54|     289.0266|
+-------------+-----------------+-------------+
```

What Happened?

Before doing the calculations, we truncated (with the help of the `cast` function, which we will learn about in the later chapters) all of the values to be only 2 `scalar` digits at most. This is how you should perform your arithmetic operations with `Decimal Types`. Ideally you should know the minimum number of `scalar` digits needed for each datatype.

Summary

- We learned that you should always initial `Decimal` types using string represented numbers, if they are an Irrational Number.
- When reading in `Decimal` types, you should explicitly override the default arguments of the Spark type and make sure that the underlying data is correct.
- When performing arithmetic operations with decimal types you should always truncate the scalar digits to the lowest number of digits as possible, if you haven't already.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2 - Performing your First Transformations")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Transformation

```
(
    pets
    .withColumn('birthday_date', F.col('birthday').cast('date'))
    .withColumn('owned_by', F.lit('me'))
    .withColumnRenamed('id', 'pet_id')
    .where(F.col('birthday_date') > datetime(2015,1,1))
).toPandas()
```

	pet_id	breed_id	nickname	birthday	age	color	birthday_date	ov
0	2	3	Argus	2016-11-22 10:05:10	10	None	2016-11-22	me
1	3	1	Chewie	2016-11-22 10:05:10	15	None	2016-11-22	me

What Happened?

- We renamed the `primary key` of our `df`
- We truncated the precision of our date types.
- we filtered our dataset to a smaller subset.
- We created a new column describing who own these pets.

Summary

We performed a variety of spark transformations to transform our data, we will go through these transformations in detailed in the following section.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.1 - Looking at Your Data")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Looking at Your Data

Spark is lazily evaluated. To look at your data you must perform a `take` operation to trigger your transformations to be evaluated. There are a couple of ways to perform a `take` operation that we'll go through here, and their performance.

For example, the `toPandas()` is a `take` operation which you've already seen in many places.

Option 1 - `collect()`

```
pets.collect()
```

```
[Row(id=u'1', breed_id=u'1', nickname=u'King', birthday=u'2014-11-22 12:30:31', age=u'5',
color=u'brown'),
 Row(id=u'2', breed_id=u'3', nickname=u'Argus', birthday=u'2016-11-22 10:05:10', age=u'10'
, color=None),
 Row(id=u'3', breed_id=u'1', nickname=u'Chewie', birthday=u'2016-11-22 10:05:10', age=u'15
', color=None)]
```

What Happened?

When you call `collect` on a `dataframe`, it will trigger a `take` operation, bring all the data to the driver node and then return all rows as a lists of `Row` objects.

Note

This should not be advised unless you **have to** look at all the rows of your dataset, you should usually sample a subset of the data. This call will execution **all** of the transformations that you have specified on **all** the data.

Option 2 - `head()/take()/first()`

```
pets.head(n=1)
```

```
[Row(id=u'1', breed_id=u'1', nickname=u'King', birthday=u'2014-11-22 12:30:31', age=u'5',
color=u'brown')]
```

What Happened?

When you call `head(n)` on a `dataframe`, it will trigger a `take` operation and return the first `n` rows of the result dataset. The different operations will return different number of rows.

Note

- If the data is **unsorted**, spark will perform the **all** the transformations on a selected amount of partitions until the number of rows are satified. This is much optimal based on how much and large your dataset is.
- If the data is **sorted**, spark will perform the same as a `collect` and perform `all` of the `transformations` on `all` of the data.

By `sorted` we mean, if any sort of "sorting of the data" is done during the transformations, such as `sort()`, `orderBy()`, etc.

Option 3 - `toPandas()`

```
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

What Happened?

When you call a `toPandas()` on a `dataframe`, it will trigger a `take` operation and return all of the rows.

This is as performant as the `collect()` function, but the most readible in my opinion.

Option 4 - `show()`

```
pets.show()
```

```
+---+--------+--------+-------------------+---+-----+
| id|breed_id|nickname|           birthday|age|color|
+---+--------+--------+-------------------+---+-----+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|
+---+--------+--------+-------------------+---+-----+
```

What Happened?

When you call a `show()` on a `dataframe`, it will trigger a `take` operation return up to 20 rows.

This is as performant as the `head()` function and more readible. (I still perfer `toPandas()`).

Summary

- We learnt about various functions that allow you to look at your data.
- Some functions are less performant than others based on if the resultant data is sorted or not.
- Try to refrain from looking at all the data, unless you are required to.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.1 - Looking at Your Data")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Selecting a Subset of Columns

When you're working with raw data, you are usually only interested in a subset of columns. This means you should get into the habit of only selecting the columns you need before any spark transformations.

Why?

If you do not, and you are working with a wide dataset this will cause your spark application to do more work than it should do. This is because all the extra columns will be `shuffled` between the worker during the execution of the transformations.

This will really kill you, if you have string columns that have really large amounts of text within them.

Note

Spark is sometimes smart enough to know which columns aren't being used and perform a `Project Pushdown` to drop the unneeded columns. But it's better practice to do the selection first.

Option 1 - `select()`

```
(
    pets
    .select("id", "nickname", "color")
    .toPandas()
)
```

	id	nickname	color
0	1	King	brown
1	2	Argus	None
2	3	Chewie	None

What Happened?

Similar to a `sql select` statement, it will only keep the columns specified in the arguments in the resulting `df` . a `list` object can be passed as the argument as well.

If you have a wide dataset and only want to work with a small number of columns, a `select` would be less lines of code.

Note

If the argument `*` is provided, all the columns will be selected.

Option 2 - `drop()`

```
(
    pets
    .drop("breed_id", "birthday", "age")
    .toPandas()
)
```

	id	nickname	color
0	1	King	brown

1	2	Argus	None
2	3	Chewie	None

What Happened?

This is the opposite of a `select` statement it will drop an of the columns specified.

If you have a wide dataset and will need a majority of the columns, a `drop` would be less lines of code.

Summary

- Work with only the subset of columns required for your spark application, there is no need do extra work.
- Depending on the number of columns you are going to work with, a `select` over a `drop` would be better and vice-versa.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.3 - Creating New Columns")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Creating New Columns and Transforming Data

When we are data wrangling, transforming data, we will using assign the result to a new column. We will explore the `withColumn()` function and other transformation functions to achieve this our end results.

We will also look into how we can rename a column with `withColumnRenamed()`, this is useful for making a join on the same `column`, etc.

Case 1: New Columns - `withColumn()`

```
(
    pets
    .withColumn('nickname_copy', F.col('nickname'))
    .withColumn('nickname_capitalized', F.upper(F.col('nickname')))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	nickname_copy	nick
0	1	1	King	2014-11-22 12:30:31	5	brown	King	KIN
1	2	3	Argus	2016-11-22 10:05:10	10	None	Argus	ARG
2	3	1	Chewie	2016-11-22 10:05:10	15	None	Chewie	CHE

What Happened?

We duplicated the `nickname` column as `nickname_copy` using the `withColumn()` function. We also created a new column where all the letters of the `nickname` are `capitalized` with chaining multiple spark functions together.

We will look into more advanced column creation in the next section. There we will go into more details what a `column expression` is and what the purpose of `F.col()` is.

Case 2: Renaming Columns - `withColumnRenamed()`

```
(
    pets
    .withColumnRenamed('id', 'pet_id')
    .toPandas()
)
```

	pet_id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

What Happened?

We renamed and replaced the `id` column with `pet_id` .

Summary

- We learned how to create new columns from old ones by chaining spark functions and using `withColumn()`.
- We learned how to rename columns using `withColumnRenamed()`.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

from datetime import date
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.4 - Constant Values")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Constant Values

There are many instances where you will need to create a `column` expression or use a constant value to perform some of the spark transformations. We'll explore some of these.

Case 1: Creating a Column with a constant value (`withColumn()`) (wrong)

```
pets.withColumn('todays_date', date.today()).toPandas()
```

```
---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)

<ipython-input-4-f87e239cb534> in <module>()
----> 1 pets.withColumn('todays_date', date.today()).toPandas()


/usr/local/lib/python2.7/site-packages/pyspark/sql/dataframe.pyc in withColumn(self, colNa
me, col)
   1846
   1847         """
-> 1848         assert isinstance(col, Column), "col should be Column"
   1849         return DataFrame(self._jdf.withColumn(colName, col._jc), self.sql_ctx)
   1850


AssertionError: col should be Column
```

What Happened?

Spark functions that have a `col` as an argument will usually require you to pass in a `Column` expression. As seen in the previous section, `withColumn()` worked fine when we gave it a column from the current `df`. But this isn't the case when we want set a column to a constant value.

If you get an `AssertionError: col should be Column` that is usually the case, we'll look into how to fix this.

Case 1: Creating a Column with a constant value (`withColumn()`) (correct)

```
pets.withColumn('todays_date', F.lit(date.today())).toPandas()
```

	id	breed_id	nickname	birthday	age	color	todays_date
0	1	1	King	2014-11-22 12:30:31	5	brown	2019-02-14
1	2	3	Argus	2016-11-22 10:05:10	10	None	2019-02-14
2	3	1	Chewie	2016-11-22 10:05:10	15	None	2019-02-14

What Happened?

With `F.lit()` you can create a `column` expression that you can now assign to a new column in your dataframe.

More Examples

```
(
    pets
    .withColumn('age_greater_than_5', F.col("age") > 5)
    .withColumn('height', F.lit(150))
    .where(F.col('breed_id') == 1)
    .where(F.col('breed_id') == F.lit(1))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	age_greater_than_5
0	1	1	King	2014-11-22 12:30:31	5	brown	False
1	3	1	Chewie	2016-11-22 10:05:10	15	None	True

What Happened?

(We will look into equilities statements later.)

The above contains constant values (column `height`) and column expressions (columns using `F.col()`) so a `F.lit()` is not required.

Summary

- You need to use `F.lit()` to assign constant values to columns.
- Equality expressions with `F.col()` is also another way to have a column expressions.
- When in doubt, always use column expressions `F.lit()`.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.5 - Casting Columns to Different Type")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Casting Columns in Different Types

Sometimes your data can be read in as all `unicode` / `string` in which you will need to cast them to the correct type. Or Simply you want to change the type of a column as a part of your transformation.

Option 1 - `cast()`

```
(
```

```
    pets
    .select('birthday')
    .withColumn('birthday_date', F.col('birthday').cast('date'))
    .withColumn('birthday_date_2', F.col('birthday').cast(T.DateType()))
    .toPandas()
)
```

	birthday	birthday_date	birthday_date_2
0	2014-11-22 12:30:31	2014-11-22	2014-11-22
1	2016-11-22 10:05:10	2016-11-22	2016-11-22
2	2016-11-22 10:05:10	2016-11-22	2016-11-22

What Happened?

There are 2 ways that you can `cast` a column.

1. Use a string (`cast('date')`).
2. Use the spark types (`cast(T.DateType())`).

I tend to use a string as it's shorter, one less import and in more editors there will be syntax highlighting for the string.

Summary

- We learnt about two ways of casting a column.
- The first way is a bit more cleaner IMO.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.6 - Filtering Data")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None

Filtering Data

Again another commonly used function in data analysis, filtering out unwanted rows.

Option 1 - `where()`

```
(
    pets
```

```
    .where(F.col('breed_id') == 1)
    .filter(F.col('color') == 'brown')
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown

What Happened?

Similar to the functions we have seen so far, there are multiple functioned that get `alias` to different names that perform the same transformation. IMO I perfor `where` as it's a bit more intuitive and closer to the `sql` syntax.

Note:

Notice how we don't have to wrap `1` or `brown` in a `F.lit()` function as these conditions are columnary expressions.

We will look into how to perform more complex conditions in `2.1.7` that contain more than 1 condition.

Option 2 - `isin()`

```
(
    pets
    .where(F.col('nickname').isin('King', 'Argus'))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None

What Happened?

If you want to know if a column can be of many values then you can use the `isin()` function. This function takes in both a list of values of comma seperated values. This is again very similar to `sql` syntax.

Summary

- We learnt of two filter functions in Spark `where()` and `isin()` .
- Using `isin` you can see if a column can contain multiple values.
- These functions are named similarly to a `sql` language.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.7 - Equality Statements in Spark and Comparison with Nulls")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white

Filtering Data

When you want ot filter data with more than just one expression, there are a couple of gotchas that you will need to be careful of.

Case 1: Multiple Conditions

```
(
    pets
    .where(
        (F.col('breed_id') == 1) &
        (F.col('color') == 'brown') &
        F.col('color').isin('brown')
    )
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown

What Happened?

When there exists more than 1 condition you will to wrap each condition in `()` brackets and as well provide bitwise operations instead of logical operations in Python.

Why?

This is because in the spark internals they had to overwrite the `logical operations` and was only left with the `bitwise operations`. This is to my best knowledge, I could be wrong.

Case 2: Nested Conditions

```
(
    pets
    .where(
        (
            F.col('breed_id').isin([1, 2]) &
            F.col('breed_id').isNotNull()
        ) |
        (F.col('color') == 'white')
    )
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	3	1	Chewie	2016-11-22 10:05:10	15	None
2	3	2	Maple	2018-11-22 10:05:10	17	white

What Happened?

Similar to before, nested conditions will need to be wrapped with `()` as well.

Case 3: Equality Statements with `Null` Values, (use `isNotNull()` and `isNull()`)

```
(
    pets
    .withColumn('result', F.col('color') != 'white')
    .withColumn(
        'result_2',
        (F.col('color') != 'white') &
        (F.col('color').isNotNull())
    )
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	result	result_2
0	1	1	King	2014-11-22 12:30:31	5	brown	True	True
1	2	3	Argus	2016-11-22 10:05:10	10	None	None	False
2	3	1	Chewie	2016-11-22 10:05:10	15	None	None	False
3	3	2	Maple	2018-11-22 10:05:10	17	white	False	False

What Happened?

If you do not come from a `sql` background any comparison with `Null` will also be `Null`, unless you specifically use the `Null` comparisons.

The 2 `Null` comparisons are `isNotNull()` and `isNull()`.

Summary

- In spark when using a more involved conditional expression, you will need to wrap each condition with `()` brackets and use the **bitwise operations** in Python.
- Be explicit with you're performing conditional transformations on columns that can be `Null`.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.8 - Case Statements")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white
4	4	2	None	2019-01-01 10:05:10	13	None

Case Statements

Case statements are usually used for performing stateful calculations.

ie.

- if `x` then `a`

- if `y` then `b`
- everything else `c`

Using Switch/Case Statements in `Spark`

```
(
    pets
    .withColumn(
        'oldness_value',
        F.when(F.col('age') <= 5, 'young')
         .when((F.col('age') > 5) & (F.col('age') <= 10), 'middle age')
         .otherwise('old')
    )
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	oldness_value
0	1	1	King	2014-11-22 12:30:31	5	brown	young
1	2	3	Argus	2016-11-22 10:05:10	10	None	middle age
2	3	1	Chewie	2016-11-22 10:05:10	15	None	old
3	3	2	Maple	2018-11-22 10:05:10	17	white	old
4	4	2	None	2019-01-01 10:05:10	13	None	old

What Happened?

Based on the age of the pet, we classified if they are either `young`, `middle age` or `old`. **Please don't take offense, this is merely an example.**

We mapped the logic of:

- If their age is younger than or equal to 5, then they are considered `young`.
- If their age is greater than 5 but younger than or equal to 10 , then they are considered `middle age`.
- Anyone older is considered `old`.

Summary

- We learned how to map values based on case statements and a deafult value if all conditions are not satified.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.9 - Filling in Null Values")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white
4	4	2	None	2019-01-01 10:05:10	13	None

Filling in Null Values

Working with real world data, some information can be missing but can be interprelated from other columns or set with default values. These interprelated values or deafult value will thus fill in those missing values. Here we will show you how to.

Option 1 - Fill in All Missing Values with a Default Value

```
(
    pets
    .fillna('Unknown')
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	Unknown
2	3	1	Chewie	2016-11-22 10:05:10	15	Unknown
3	3	2	Maple	2018-11-22 10:05:10	17	white
4	4	2	Unknown	2019-01-01 10:05:10	13	Unknown

What Happened?

Using `fillna` we attempt to fill in all `Null` values with the value `'Unknown'`. This will be fine if all the `Null` values are strings, but it won't work if for say the `age` column is missing values. We look at how to specify different values for different columns next.

Option 2 - Fill in All Missing Values with a Mapping

```
(
    pets
    .fillna({
        'nickname': 'Unknown Nickname',
        'color':    'Unknown Color',
    })
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	Unknown Color
2	3	1	Chewie	2016-11-22 10:05:10	15	Unknown Color
3	3	2	Maple	2018-11-22 10:05:10	17	white
4	4	2	Unknown Nickname	2019-01-01 10:05:10	13	Unknown Color

What Happened?

You have the option of filling in each column with a diffferent value. This provides more flexibility as most times the columns will be different types and a single deafult value won't be sufficient enough.

Option 2 - `coalesce()`

```
(
    pets
    .withColumn('bogus', F.coalesce(F.col('color'), F.col('nickname'), F.lit('Default')))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	bogus
0	1	1	King	2014-11-22 12:30:31	5	brown	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None	Argus
2	3	1	Chewie	2016-11-22 10:05:10	15	None	Chewie
3	3	2	Maple	2018-11-22 10:05:10	17	white	white
4	4	2	None	2019-01-01 10:05:10	13	None	Default

What Happened?

Another way to fill in a column with values is using `coalesce()`. This function will try to fill in the specified columns by looking at the given arguments in order from left to right, until one of the arguments is not null and use that. If all else fails, you can provide a "default" value as your last arugment (remembering that it should be a columnar expression).

In our example, it will attempt to fill in the `bogus` column with values from the `color` column first, if that is null then try the `nickname` column next, and if both are null it will use the deafult value `Default`.

Summary

- We looked at a generic way of filling in a columns with a single default value.
- We looked at providing a mapping of `{column:value}` to fill in each column seperately.
- Lastly we looked at how to fill in a column using other columns and default values in an order of precedency.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.10 - Spark Functions aren't Enough, I Need my Own!")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white

Spark Functions aren't Enough, I Need my Own!

What happens when you can't find functions that can perform what you want? Try looking again . But if this is really the case, your last resort can be to implement an `udf` short for `user defined function`.

These are functions written in python code that take a subset of columns as the input and returns a new column back. There are multiple steps in creating a `udf`, we'll walk through one and decompose it step by step.

Option 1: Steps

```
# Step 1: Create your own function
def uppercase_words(word, cuttoff_length=2):
    return word.upper()[:cuttoff_length] if word else None

s = 'Hello World!'
print(s)
print(uppercase_words(s, 20))

# Step 2: Register the udf as a spark udf
uppercase_words_udf = F.udf(uppercase_words, T.StringType())

# Step 3: Use it!
(
    pets
    .withColumn('nickname_uppercase', uppercase_words_udf(F.col('nickname')))
    .withColumn('color_uppercase', uppercase_words_udf(F.col('color')))
    .withColumn('color_uppercase_trimmed', uppercase_words_udf(F.col('color'), F.lit(3)))
    .toPandas()
)
```

```
Hello World!
HELLO WORLD!
```

	id	breed_id	nickname	birthday	age	color	nickname_uppercase
0	1	1	King	2014-11-22 12:30:31	5	brown	KI
1	2	3	Argus	2016-11-22 10:05:10	10	None	AR
2	3	1	Chewie	2016-11-22 10:05:10	15	None	CH
3	3	2	Maple	2018-11-22 10:05:10	17	white	MA

What Happened?

Although the upper function is defined in the spark `fuctions` library it still serves as a good example. Let's breakdown the steps involved:

1. Create the function that you want (`uppercase_words`), remembering that only `spark columnar`

`objects` are accepted as input arguments to the function. This means if you want to use other values, you will need to cast it to a column object using `F.lit()` from the previous sections.
2. Register the python function as a spark function, and specify the spark return type. The format is like so `F.udf(python_function, spark_return_type)`.
3. Now you can use the function!

Option 2: 1 Less Step

```
from pyspark.sql.functions import udf

# Step 1: Create and register your own function
@udf('string', 'int')
def uppercase_words(word, cuttoff_length=2):
    return word.upper()[:cuttoff_length] if word else None

# Step 2: Use it!
(
    pets
    .withColumn('color_uppercase', uppercase_words_udf(F.col('color')))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	color_uppercase
0	1	1	King	2014-11-22 12:30:31	5	brown	BR
1	2	3	Argus	2016-11-22 10:05:10	10	None	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None	None
3	3	2	Maple	2018-11-22 10:05:10	17	white	WH

What Happened?

The `udf` function can also be used as a decorator to register your python functions as spark functions.

Where the inputs are the types of the arguments to the `udf`.

The Ugly Part of `udf` s

TL;DR `Spark function` s are executed on the JVM, while `Python UDF` s are executed in Python. This will require extra python memory for your spark application (will explain in Chapter 6) and more passing of data between the JVM and Python.

If your function can be performed with the spark `functions`, you should alway use the spark `functions`. `udf`s perform very poorly compared to the spark `functions`. This is a greate response that encapsulates the reason as to why:

> The main reasons are already enumerated above and can be reduced to a simple fact that Spark DataFrame is natively a JVM structure and standard access methods are implemented by simple calls to Java API. UDF from the other hand are implemented in Python and require moving data back and forth.
>
> While PySpark in general requires data movements between JVM and Python, in case of low level RDD API it typically doesn't require expensive serde activity. Spark SQL adds additional cost of serialization and serialization as well cost of moving data from and to unsafe representation on JVM. The later one is specific to all UDFs (Python, Scala and Java) but the former one is specific to non-native languages.
>
> Unlike UDFs, Spark SQL functions operate directly on JVM and typically are well integrated with both Catalyst and Tungsten. It means these can be optimized in the execution plan and most of the time can benefit from codgen and other Tungsten optimizations. Moreover these can operate on data in its "native" representation.
>
> So in a sense the problem here is that Python UDF has to bring data to the code while SQL expressions go the other way around.

source: https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance

Option 3: Pandas Vectorized `UDF`s

```
# TODO: https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html
```

Summary

- We learnt how to use a python function within spark, called `udf`s.
- We learnt how to pass non-column objects into the function by using knowledge gained from previous chapters.
- We learnt about the bad parts of `udf`s and their performance issues.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.11  - Unionizing Multiple Dataframes")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white

Unionizing Multiple Dataframes

There are a couple of situations where you would want to perform an union transformation.

Case 1: Collecting Data from Various Sources

When you're collecting data from multiple sources, some point in your spark application you will need to reconcile all the different sources into the same format and work with a single source of truth. This will require you to `union` the different datasets together.

Case 2: Perfoming Different Transformations on your Dataset

Sometimes you would like to perform seperate transformations on different parts of your data based on your task. This would involve breaking up your dataset into different parts and working on them individually. Then at some point you might want to stitch they back together, this would again be a `union` operation.

Case 1 - `union()` (the Wrong Way)

```
pets_2 = pets.select(
    'breed_id',
    'id',
    'age',
    'color',
    'birthday',
    'nickname'
)

(
    pets
    .union(pets_2)
    .where(F.col('id').isin(1,2))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	1	1	5	brown	2014-11-22 12:30:31	King
3	1	3	15	None	2016-11-22 10:05:10	Chewie
4	2	3	17	white	2018-11-22 10:05:10	Maple

Case 1 - Another Wrong Way

```
pets_3 = pets.select(
    '*',
    '*'
)
```

```
pets_3.show()

(
    pets
    .union(pets_3)
    .where(F.col('id').isin(1,2))
    .toPandas()
)
```

```
+---+--------+--------+-------------------+---+-----+---+--------+--------+--------------
----+---+-----+
| id|breed_id|nickname|           birthday|age|color| id|breed_id|nickname|           birt
hday|age|color|
+---+--------+--------+-------------------+---+-----+---+--------+--------+--------------
----+---+-----+
|  1|       1|    King|2014-11-22 12:30:31|  5|brown|  1|       1|    King|2014-11-22 12:3
0:31|  5|brown|
|  2|       3|   Argus|2016-11-22 10:05:10| 10| null|  2|       3|   Argus|2016-11-22 10:0
5:10| 10| null|
|  3|       1|  Chewie|2016-11-22 10:05:10| 15| null|  3|       1|  Chewie|2016-11-22 10:0
5:10| 15| null|
|  3|       2|   Maple|2018-11-22 10:05:10| 17|white|  3|       2|   Maple|2018-11-22 10:0
5:10| 17|white|
+---+--------+--------+-------------------+---+-----+---+--------+--------+--------------
----+---+-----+
```

```
---------------------------------------------------------------------------

AnalysisException                         Traceback (most recent call last)

<ipython-input-5-c8157f574918> in <module>()
      8 (
      9     pets
---> 10     .union(pets_3)
     11     .where(F.col('id').isin(1,2))
     12     .toPandas()


/usr/local/lib/python2.7/site-packages/pyspark/sql/dataframe.pyc in union(self, other)
   1336         Also as standard in SQL, this function resolves columns by position (not b
y name).
   1337         """
-> 1338         return DataFrame(self._jdf.union(other._jdf), self.sql_ctx)
   1339
   1340     @since(1.3)


/usr/local/lib/python2.7/site-packages/py4j/java_gateway.pyc in __call__(self, *args)
   1255         answer = self.gateway_client.send_command(command)
   1256         return_value = get_return_value(
-> 1257             answer, self.gateway_client, self.target_id, self.name)
```

```
   1258
   1259         for temp_arg in temp_args:


/usr/local/lib/python2.7/site-packages/pyspark/sql/utils.pyc in deco(*a, **kw)
     67                                             e.java_exception.getStackTrace()))
     68             if s.startswith('org.apache.spark.sql.AnalysisException: '):
---> 69                 raise AnalysisException(s.split(': ', 1)[1], stackTrace)
     70             if s.startswith('org.apache.spark.sql.catalyst.analysis'):
     71                 raise AnalysisException(s.split(': ', 1)[1], stackTrace)


AnalysisException: u"Union can only be performed on tables with the same number of columns
, but the first table has 6 columns and the second table has 12 columns;;\n'Union\n:- Rela
tion[id#10,breed_id#11,nickname#12,birthday#13,age#14,color#15] csv\n+- Project [id#10, br
eed_id#11, nickname#12, birthday#13, age#14, color#15, id#10, breed_id#11, nickname#12, bi
rthday#13, age#14, color#15]\n   +- Relation[id#10,breed_id#11,nickname#12,birthday#13,age
#14,color#15] csv\n"
```

What Happened?

This actually worked out quite nicely, I forgot this was the case actually. **Spark will only allow you to union `df` that have the exact number of columns and where the column datatypes are exactly the same.**

Case 1

Because we infered the schema and datatypes from the csv file it was able to union the 2 dataframes, but the results doesn't make sense at all; The columns don't match up.

Case 2

We created a new dataframe with twice the numnber of columns and tried to union it with the original `df`, spark threw an error as it doesn't know what to do when the number of columns don't match up.

Case 2 - `union()` (the Right Way)

```
(
    pets
    .union(pets_2.select(pets.columns))
    .union(pets_3.select(pets.columns))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white

4	1	1	King	2014-11-22 12:30:31	5	brown
5	2	3	Argus	2016-11-22 10:05:10	10	None
6	3	1	Chewie	2016-11-22 10:05:10	15	None
7	3	2	Maple	2018-11-22 10:05:10	17	white
8	1	1	King	2014-11-22 12:30:31	5	brown
9	2	3	Argus	2016-11-22 10:05:10	10	None
10	3	1	Chewie	2016-11-22 10:05:10	15	None
11	3	2	Maple	2018-11-22 10:05:10	17	white

What Happened?

The columns match perfectly! How? **For each of the new `df` that you would like to union with the original `df` you will `select` the column from the original `df` during the union.** This will:

1. Guarantees the ordering of the columns, as a `select` will select the columns in order of which they are listed in.
2. Guarantees that only the columns of the original `df` is selected, from the previous sections, we know that `select` will only the specified columns.

Summary

- Always always be careful when you are `union`ing `df` together.
- When you `union` `df`s together you should ensure:
  1. The number of columns are the same.
  2. The columns are the exact same.
  3. The columns are in the same order.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 2.12 - Performing Joins (clean one)")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

```
pets = spark.createDataFrame(
    [
        (1, 1, 'Bear'),
        (2, 1, 'Chewie'),
        (3, 2, 'Roger'),
    ], ['id', 'breed_id', 'nickname']
)

pets.toPandas()
```

	id	breed_id	nickname
0	1	1	Bear
1	2	1	Chewie
2	3	2	Roger

```
breeds = spark.createDataFrame(
    [
        (1, 'Pitbull', 10),
        (2, 'Corgie', 20),
    ], ['id', 'name', 'average_height']
)

breeds.toPandas()
```

	id	name	average_height
0	1	Pitbull	10
1	2	Corgie	20

Performing Joins

There are typically two types of joins in sql:

1. `Inner Join` is where 2 tables are joined on the basis of common columns mentioned in the ON clause.

   ie. `left.join(right, left[lkey] == right[rkey])`

1. `Natural Join` is where 2 tables are joined on the basis of all common columns.

   ie. `left.join(right, 'key')`

source: https://stackoverflow.com/a/8696402

Question: Which is better? Is it just a style choice?

Option 1: Inner Join (w/Different Keys)

```
join_condition = pets['breed_id'] == breeds['id']

df = pets.join(breeds, join_condition)

df.toPandas()
```

	id	breed_id	nickname	id	name	average_height
0	1	1	Bear	1	Pitbull	10
1	2	1	Chewie	1	Pitbull	10
2	3	2	Roger	2	Corgie	20

What Happened:

- We have 2 columns named `id`, but they refer to different things.
- We can't uniquely reference these 2 columns (easily, still possible).
- Pretty long `join expression`.

This is not ideal. Let's try `renaming` it before the join?

Option 2: Inner Join (w/Same Keys)

```
breeds = breeds.withColumnRenamed('id', 'breed_id')
join_condition = pets['breed_id'] == breeds['breed_id']

df = pets.join(breeds, join_condition)

df.toPandas()
```

	id	breed_id	nickname	breed_id	name	average_height

0	1	1	Bear	1	Pitbull	10
1	2	1	Chewie	1	Pitbull	10
2	3	2	Roger	2	Corgie	20

What Happened:

- We have 2 columns named `breed_id` which mean the same thing!
- Duplicate columns appear in the result.
- Still pretty long `join expression`.

This is again not ideal.

Option 3: Natural Join

```
df = pets.join(breeds, 'breed_id')
df.toPandas()
```

	breed_id	id	nickname	name	average_height
0	1	1	Bear	Pitbull	10
1	1	2	Chewie	Pitbull	10
2	2	3	Roger	Corgie	20

What Happened:

- No duplicated column!
- No extra column!
- A single string required for the `join expression` (list of column/keys, if joining on multiple column/keys join).

Summary

Preforming a `natural join` was the most elegant solution in terms of `join expression` and the resulting `df`.

NOTE: These rules also apply to the other join types (ie. `left` and `right`).

**Some might argue that you will need both join keys in the result for further transformations such as filter only the left or right key, but I would recommend doing this before the join, as this is more performant.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import types as T

from pyspark.sql import functions as F

from datetime import datetime
from decimal import Decimal
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 3.1 - One to Many Rows")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext

import os

data_path = "/data/pets.csv"
base_path = os.path.dirname(os.getcwd())
path = base_path + data_path
```

```
pets = spark.read.csv(path, header=True)
pets.toPandas()
```

	id	breed_id	nickname	birthday	age	color
0	1	1	King	2014-11-22 12:30:31	5	brown
1	2	3	Argus	2016-11-22 10:05:10	10	None
2	3	1	Chewie	2016-11-22 10:05:10	15	None
3	3	2	Maple	2018-11-22 10:05:10	17	white
4	4	2	None	2019-01-01 10:05:10	13	None

One to Many Rows

Very commonly you might have a column where it is an `array` type and you want to flatten that array out into multiple rows? Well let's look at how we can do that and some practical applications of it.

Multiple People Interested in Babysitting the Same Pet

Case 1: Get a Table with just the People Interested

Question to answer:

We have a couple of people interested in our little friend named "King", we want to create a new dataset containing a single row for each people interested in King. How can I do this?

```
pets_2 = (
    pets
    .where(F.col('id') == 1)
    .withColumn(
        'people_interested',
        F.array([
            F.lit('John'),
            F.lit('Doe'),
            F.lit('Bob'),
            F.lit('Billy')
        ])
    )
)

pets_2.toPandas()
```

	id	breed_id	nickname	birthday	age	color	people_interested
0	1	1	King	2014-11-22 12:30:31	5	brown	[John, Doe, Bob, Billy]

```
(
    pets_2
    .withColumn('people_interested', F.explode(F.col('people_interested')))
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	people_interested
0	1	1	King	2014-11-22 12:30:31	5	brown	John
1	1	1	King	2014-11-22 12:30:31	5	brown	Doe
2	1	1	King	2014-11-22 12:30:31	5	brown	Bob
3	1	1	King	2014-11-22 12:30:31	5	brown	Billy

What Happened?

1. So we first created a column that contained a list of names of the people interested.
2. Next we split the list so that there is a row per person interested.

Case 2: Get a Table with just the People Interested and the Number of Days

Question to answer:

We have a couple of people interested in our little friend named "King" and the number of days they would like to babysit for, we want to create a new dataset containing a single row for each people interested in King. How can I do this?

```
pets_2 = (
    pets
    .where(F.col('id') == 1)
    .withColumn(
        'people_interested',
        F.array([
            F.create_map([F.lit('John'), F.lit(5)]),
            F.create_map([F.lit('Doe'), F.lit(3)]),
            F.create_map([F.lit('Bob'), F.lit(7)]),
            F.create_map([F.lit('Billy'), F.lit(9)])
        ])
    )
)

pets_2.toPandas()
```

	id	breed_id	nickname	birthday	age	color	people_interested
0	1	1	King	2014-11-22 12:30:31	5	brown	[{u'John': 5}, {u'Doe': 3}, {u'Bob': 7}, {u'Bi...

```
(
    pets_2
    .withColumn('people_interested', F.explode(F.col('people_interested')))
    .select(
        "*",
        F.explode('people_interested').alias('person', 'days')
    )
    .toPandas()
)
```

	id	breed_id	nickname	birthday	age	color	people_interested	pe
0	1	1	King	2014-11-22 12:30:31	5	brown	{u'John': 5}	Jo
1	1	1	King	2014-11-22	5	brown	{u'Doe': 3}	Do

				12:30:31				
2	1	1	King	2014-11-22 12:30:31	5	brown	{u'Bob': 7}	Bo
3	1	1	King	2014-11-22 12:30:31	5	brown	{u'Billy': 9}	Bi

What Happened?

1. So we first created a column that contained a list of dictionary mapping the name of the person interested with the number of days they were interested.
2. Here we had to explode twice, first to get a row per person interested (table grew longer), then to split each field of the dictioanry to get the name of the person and the number of days (table grew wider).

Summary

- We looked at some pretty complex transformations that involved using the `explode` function which decomposed an `array` / `map` object into multiple rows/columns.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Section 3.2 - Range Join Conditions (WIP)")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)


sc = spark.sparkContext
```

```
geo_loc_table = spark.createDataFrame([
    (1, 10, "foo"),
    (11, 36, "bar"),
    (37, 59, "baz"),
], ["ipstart", "ipend", "loc"])

geo_loc_table.toPandas()
```

	ipstart	ipend	loc
0	1	10	foo
1	11	36	bar
2	37	59	baz

```
records_table = spark.createDataFrame([
    (1, 11),
    (2, 38),
    (3, 50),
],["id", "inet"])

records_table.toPandas()
```

	id	inet
0	1	11
1	2	38
2	3	50

Range Join Conditions

> A naive approach (just specifying this as the range condition) would result in a full cartesian product and a filter that enforces the condition (tested using Spark 2.0). This has a horrible effect on performance, especially if DataFrames are more than a few hundred thousands records.

source: http://zachmoshe.com/2016/09/26/efficient-range-joins-with-spark.html

> The source of the problem is pretty simple. When you execute join and join condition is not equality based the only thing that Spark can do right now is expand it to Cartesian product followed by filter what is pretty much what happens inside `BroadcastNestedLoopJoin`

source: https://stackoverflow.com/questions/37953830/spark-sql-performance-join-on-value-between-min-and-max?answertab=active#tab-top

Option #1

```
join_condition = [
    records_table['inet'] >= geo_loc_table['ipstart'],
    records_table['inet'] <= geo_loc_table['ipend'],
]

df = records_table.join(geo_loc_table, join_condition, "left")

df.toPandas()
```

	id	inet	ipstart	ipend	loc
0	1	11	11	36	bar
1	2	38	37	59	baz
2	3	50	37	59	baz

```
df.explain()
```

```
== Physical Plan ==
BroadcastNestedLoopJoin BuildRight, LeftOuter, ((inet#252L >= ipstart#245L) && (inet#252L
<= ipend#246L))
:- Scan ExistingRDD[id#251L,inet#252L]
+- BroadcastExchange IdentityBroadcastMode
   +- Scan ExistingRDD[ipstart#245L,ipend#246L,loc#247]
```

Option #2

```
from bisect import bisect_right
from pyspark.sql.functions import udf
```

```
from pyspark.sql.types import LongType

geo_start_bd = spark.sparkContext.broadcast(map(lambda x: x.ipstart, geo_loc_table
    .select("ipstart")
    .orderBy("ipstart")
    .collect()
))

def find_le(x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(geo_start_bd.value, x)
    if i:
        return geo_start_bd.value[i-1]
    return None

records_table_with_ipstart = records_table.withColumn(
    "ipstart", udf(find_le, LongType())("inet")
)

df = records_table_with_ipstart.join(geo_loc_table, ["ipstart"], "left")

df.toPandas()
```

	ipstart	id	inet	ipend	loc
0	37	2	38	59	baz
1	37	3	50	59	baz
2	11	1	11	36	bar

```
df.explain()
```

```
== Physical Plan ==
*(4) Project [ipstart#272L, id#251L, inet#252L, ipend#246L, loc#247]
+- SortMergeJoin [ipstart#272L], [ipstart#245L], LeftOuter
   :- *(2) Sort [ipstart#272L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(ipstart#272L, 200)
   :     +- *(1) Project [id#251L, inet#252L, pythonUDF0#281L AS ipstart#272L]
   :        +- BatchEvalPython [find_le(inet#252L)], [id#251L, inet#252L, pythonUDF0#281L]
   :           +- Scan ExistingRDD[id#251L,inet#252L]
   +- *(3) Sort [ipstart#245L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(ipstart#245L, 200)
         +- Scan ExistingRDD[ipstart#245L,ipend#246L,loc#247]
```

Library Imports

```
from datetime import datetime

from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Initial Datasets

```
pets = spark.createDataFrame(
    [
        (1, 1, 'Bear', 5),
        (2, 1, 'Chewie', 10),
        (3, 2, 'Roger', 15),
    ], ['id', 'breed_id', 'nickname', 'age']
)

pets.toPandas()
```

	id	breed_id	nickname	age
0	1	1	Bear	5
1	2	1	Chewie	10
2	3	2	Roger	15

```
groupby_columns = ['breed_id']
```

Option 1: Using a Dictionary

```
df_1 = (
    pets
    .groupby(groupby_columns)
    .agg({
```

```
        "*": "count",
        "age": "sum",
    })
)

df_1.toPandas()
```

	breed_id	count(1)	sum(age)
0	1	2	15
1	2	1	15

What Happened:

- Very similar to `pandas` `agg` function.
- The resultant column names are a bit awkward to use after the fact.

Option 2: Using List of Columns

```
df_2 = (
    pets
    .groupby(groupby_columns)
    .agg(
        F.count("*"),
        F.sum("age"),
    )
)

df_2.toPandas()
```

	breed_id	count(1)	sum(age)
0	1	2	15
1	2	1	15

What Happened:

- Here we use the Spark `agg` functions.
- Again, the resultant column names are a bit awkward to use after the fact.

Option 3: Using List of Columns, with Aliases

```
df_3 = (
    pets
    .groupby(groupby_columns)
    .agg(
        F.count("*").alias("count_of_breeds"),
        F.sum("age").alias("total_age_of_breeds"),
    )
)
```

```
df_3.toPandas()
```

	breed_id	count_of_breeds	total_age_of_breeds
0	1	2	15
1	2	1	15

What Happened:

- Here we use the Spark `agg` functions and `alias`ed the resultant columns to new names.
- This provides cleaner column names that we can use later on.

Summary

I encourage using option #3.

This creates more elegant and meaning names for the new aggregate columns.

A `withColumnRenamed` can be performed after the aggregates, but why not do it with an `alias`? It's easier as well.

Introduction

There are use cases where we would like to get the `first` or `last` of something within a `group` or particular `grain`.

It is natural to do something in SQL like:

```
select
    col_1,
    first(col_2) as first_something,
    last(col_2) as first_something
from table
group by 1
order by 1
```

Which leads us to writing spark code like this `df.orderBy().groupBy().agg()`. This has unexpected behaviours in spark and can be different each run.

Library Imports

```
from datetime import datetime

from pyspark.sql import SparkSession
from pyspark.sql import functions as F, Window
```

Create a `SparkSession`. No need to create `SparkContext` as you automatically get it as part of the `SparkSession`.

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Initial Datasets

```
pets = spark.createDataFrame(
    [
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 'Bear', 5),
        (2, 1, datetime(2010, 1, 1, 1 ,1, 1), 'Chewie', 15),
        (3, 1, datetime(2015, 1, 1, 1 ,1, 1), 'Roger', 10),
    ], ['id', 'breed_id', 'birthday', 'nickname', 'age']
)
```

```
pets.toPandas()
```

	id	breed_id	birthday	nickname	age
0	1	1	2018-01-01 01:01:01	Bear	5
1	2	1	2010-01-01 01:01:01	Chewie	15
2	3	1	2015-01-01 01:01:01	Roger	10

Option 1: Wrong Way

Result 1

```
df_1 = (
    pets
    .orderBy('birthday')
    .groupBy('breed_id')
    .agg(F.first('nickname').alias('first_breed'))
)

df_1.toPandas()
```

	breed_id	first_breed
0	1	Chewie

Result 2

```
df_2 = (
    pets
    .orderBy('birthday')
    .groupBy('breed_id')
    .agg(F.first('nickname').alias('first_breed'))
)

df_2.toPandas()
```

	breed_id	first_breed
0	1	Chewie

Option 2: Window Object, Right Way

```
window = Window.partitionBy('breed_id').orderBy('birthday')

df_3 = (
    pets
```

```
    .withColumn('first_breed', F.first('nickname').over(window.rowsBetween(Window.unbounde
dPreceding, Window.unboundedFollowing)))
    .withColumn('rn', F.row_number().over(window.rowsBetween(Window.unboundedPreceding, Wi
ndow.currentRow)))
)

df_3.toPandas()
```

	id	breed_id	birthday	nickname	age	first_breed	rn
0	2	1	2010-01-01 01:01:01	Chewie	15	Chewie	1
1	3	1	2015-01-01 01:01:01	Roger	10	Chewie	2
2	1	1	2018-01-01 01:01:01	Bear	5	Chewie	3

Summary

Ok so my example didn't work locally lol, but trust me it that `orderBy()` in a statement like this: `orderBy().groupBy()` doesn't maintain it's order!

reference: https://stackoverflow.com/a/50012355

For anything aggregation that needs an ordering performed (ie. `first`, `last`, etc.), we should avoid using `groupby()` s and instead we should use a `window` object.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql import functions as F

from datetime import datetime
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Initial Datasets

```
pets = spark.createDataFrame(
    [
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 'Bear', 5),
        (2, 1, datetime(2015, 1, 1, 1 ,1, 1), 'Chewie', 10),
        (3, 1, datetime(2015, 1, 1, 1 ,1, 1), 'Roger', 15),
    ], ['id', 'breed_id', 'birthday', 'nickname', 'age']
)

pets.toPandas()
```

	id	breed_id	nickname	age
0	1	1	Bear	5
1	2	1	Chewie	10
2	3	1	Roger	15

Scenario #1

No `orderBy` specified for `window` object.

```
window_1 = Window.partitionBy('breed_id')

df_1 = pets.withColumn('foo', (F.sum(F.col('age')).over(window_1)))
```

```
df_1.toPandas()
```

	id	breed_id	nickname	age	foo
0	1	1	Bear	5	30
1	2	1	Chewie	10	30
2	3	1	Roger	15	30

Scenario #2

`orderBy` with no `rowsBetween` specified for `window` object.

```
window_2 = (
    Window
    .partitionBy('breed_id')
    .orderBy(F.col('id'))
)

df_2 = pets.withColumn('foo', (F.sum(F.col('age')).over(window_2)))

df_2.toPandas()
```

	id	breed_id	nickname	age	foo
0	1	1	Bear	5	5
1	2	1	Chewie	10	15
2	3	1	Roger	15	30

Scenario #3

`orderBy` with a `rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)` specified for `window` object.

```
window_3 = (
    Window
    .partitionBy('breed_id')
    .orderBy(F.col('id'))
    .rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
)

df_3 = pets.withColumn('foo', (F.sum(F.col('age')).over(window_3)))

df_3.toPandas()
```

	id	breed_id	nickname	age	foo
0	1	1	Bear	5	30

1	2	1	Chewie	10	30
2	3	1	Roger	15	30

Why is This?

```
df_1.explain()
```

```
== Physical Plan ==
Window [sum(age#3L) windowspecdefinition(breed_id#1L, specifiedwindowframe(RowFrame, unbou
ndedpreceding$(), unboundedfollowing$())) AS foo#9L], [breed_id#1L]
+- *(1) Sort [breed_id#1L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(breed_id#1L, 200)
      +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2,age#3L]
```

```
df_2.explain()
```

```
== Physical Plan ==
Window [sum(age#3L) windowspecdefinition(breed_id#1L, id#0L ASC NULLS FIRST, specifiedwind
owframe(RangeFrame, unboundedpreceding$(), currentrow$())) AS foo#216L], [breed_id#1L], [i
d#0L ASC NULLS FIRST]
+- *(1) Sort [breed_id#1L ASC NULLS FIRST, id#0L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(breed_id#1L, 200)
      +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2,age#3L]
```

```
df_3.explain()
```

```
== Physical Plan ==
Window [sum(age#3L) windowspecdefinition(breed_id#1L, id#0L ASC NULLS FIRST, specifiedwind
owframe(RowFrame, unboundedpreceding$(), unboundedfollowing$())) AS foo#423L], [breed_id#1
L], [id#0L ASC NULLS FIRST]
+- *(1) Sort [breed_id#1L ASC NULLS FIRST, id#0L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(breed_id#1L, 200)
      +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2,age#3L]
```

TL;DR

By looking at the **Physical Plan**, the default behaviour for `Window.partitionBy('col_1').orderBy('col_2')` without a `.rowsBetween()` is to do `.rowsBetween(Window.unboundedPreceding, Window.currentRow)`.

Looking at the scala code we can see that this is indeed the default and intended behavior, https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/expressions/Window.scala#L36-L38.

```
  * @note When ordering is not defined, an unbounded window frame (rowFrame, unboundedPrece
ding,
  *       unboundedFollowing) is used by default. When ordering is defined, a growing windo
w frame
  *       (rangeFrame, unboundedPreceding, currentRow) is used by default.
```

Problem: This will cause problems if you're care about all the rows in the partitions.

When dealing with big data, some datasets will have a much higher frequent of "events" than others.

An example table could be a table that tracks each pageview, it's not uncommon for someone to visit a site at the same time as someone else, espically a very popular site such as google.

I will illustrate how you can deal with these types of events, when you need to order by time.

Library Imports

```
from datetime import datetime

from pyspark.sql import SparkSession
from pyspark.sql import functions as F, Window
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Option 1: Only ordering by date column

```
window = (
    Window
    .partitionBy('breed_id')
    .orderBy('birthday')
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)
)
```

```
pets = spark.createDataFrame(
    [
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 45),
        (2, 1, datetime(2018, 1, 1, 1 ,1, 1), 20),
    ], ['id', 'breed_id', 'birthday', 'age']
)

pets.withColumn('first_pet_of_breed', F.first('id').over(window)).toPandas()
```

	id	breed_id	birthday	age	first_pet_of_breed
0	1	1	2018-01-01 01:01:01	45	1
1	2	1	2018-01-01 01:01:01	20	1

```
pets = spark.createDataFrame(
    [
        (2, 1, datetime(2018, 1, 1, 1 ,1, 1), 20),
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 45),
    ], ['id', 'breed_id', 'birthday', 'age']
)

pets.withColumn('first_pet_of_breed', F.first('id').over(window)).toPandas()
```

	id	breed_id	birthday	age	first_pet_of_breed
0	2	1	2018-01-01 01:01:01	20	2
1	1	1	2018-01-01 01:01:01	45	2

What Happened:

- By changing the order of rows (this would happen with larger amounts of data stored on different partitions), we got a different value for "first" value.
- `datetime`s can only be accurate to the second and if data is coming in faster than that, it is ambiguous to order by the date column.

Option 2: Order by `date` and `id` Column

Window Object

```
window_2 = (
    Window
    .partitionBy('breed_id')
    .orderBy('birthday', 'id')
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)
)
```

```
pets = spark.createDataFrame(
    [
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 45),
        (2, 1, datetime(2018, 1, 1, 1 ,1, 1), 20),
    ], ['id', 'breed_id', 'birthday', 'age']
)

pets.withColumn('first_pet_of_breed', F.first('id').over(window_2)).toPandas()
```

	id	breed_id	birthday	age	first_pet_of_breed
0	1	1	2018-01-01 01:01:01	45	1
1	2	1	2018-01-01 01:01:01	20	1

```
pets = spark.createDataFrame(
    [
        (2, 1, datetime(2018, 1, 1, 1 ,1, 1), 20),
        (1, 1, datetime(2018, 1, 1, 1 ,1, 1), 45),
    ], ['id', 'breed_id', 'birthday', 'age']
)

pets.withColumn('first_pet_of_breed', F.first('id').over(window_2)).toPandas()
```

	id	breed_id	birthday	age	first_pet_of_breed
0	1	1	2018-01-01 01:01:01	45	1
1	2	1	2018-01-01 01:01:01	20	1

What Happened:

- We get the same "first" value in both incidents, which is what we expect.

TL;DR

In databases, the `id` (primary key) column of a table is usually monotonically increasing. Therefore if we are dealing with frequently arriving data we can additionally sort by `id` along the `date` column.

Section 1.1 - Understanding Distributed Systems and how Spark Works

Working with Spark requires a different kind of thinking. Your code isn't executing in a sequential manor anymore, it's being executing in parallel. To write performant parallel code, you will need to think about how you can perform as different/same tasks at the same while minimizing the blocking of other tasks. Hopefully by understanding how spark and distributed systems work you can get into the right mindset and write clean parallel spark code.

Understanding Distributed Systems/Computing

Sequential Applications

In a generic application the code path is run in a sequential order. As in the code will execute from the top of the file to the bottom of the file, line by line.

Multiprocess/threaded Applications

In a `multi-processed/threaded` application the code path will diverge. The application will assign a portions of the code to `threads/processes` which will handle these tasks in an `asynchronous` manner. Once these tasks are completed the `threads/processes` will signal the main application and the code path will conform again.

These `threads/processes` are allocated a certain amount of memory and processing power on a single machine where the main application is running.

Think about how your computer can handle multiple applications at once. This is multiple processes running on a single machine.

Distributed Computing (Clusters and Nodes)

Spark is a distributed computing library that can be used in either Python, Scala, Java or R. When we say "distributed computing" we mean that our application runs on multiple machines/computers called `Nodes` which are all part of a single `Cluster` .

This is very similar to how a `multi-processed` application would work, just with more processing juice. Each `Node` is essentially a computer running multiple process running at once.

Master/Slave Architecture

From the image in the last section we can see there are 2 types of `Nodes` , a single `Driver/Master Node` and multiple `Worker Nodes` .

Each machine is assigned a portion of the overall work in the application. Through communication and message passing the machines attempt to compute each portion of the work in parallel. When the work is dependent on another portion of work, then it will have to wait until that work is computed and passed to the worker. When every portion of the work is done, it is all sent back to the `Master Node`. The coordination of the communication and message passing is done by the `Master Node` talking to the `Worker Nodes`.

You can think of it as a team lead with multiple engineers, UX, designers, etc. The lead assigns tasks to everyone. The designers and UX collaborate to create an effective interface that is user friendly, once they are done they report back to the lead. The lead will pass this information to engineers and they can start coding the interface, etc.

Lazy Execution

When you're writing a spark application, no work is actually being done until you perform a `collect` action. As seen in some examples in the previous chapters, a `collect` action is when you want to see the results of your spark transformations in the form of a `toPandas()`, `show()`, etc. This triggers the `Driver` to start the distribution of work, etc.

For now this is all you need to know, we will look into why Spark works this way and why it's a desired design decision.

MapReduce

When the `Driver Node` actually starts to do some work, it communications and distributes work using a technique called "MapReduce". There are two essential behaviors of a MapReduce application, `map` and `reduce`.

Map

When the `Driver Node` is distributing the work it `maps` the 1) data and 2) transformations to each `Worker Node`. This allows the `Worker Nodes` to perform the work (transformations) on the associated data in parallel.

Ex.

```
# initialize your variables
x = 5
y = 10

# do some transformations to your variables
x = x * 2
y = y + 2
```

Here we can see that the arithmetic operations performed on `x` and `y` can be done independently, so we can do those 2 operations in parallel on two different `Worker Nodes`. So Spark will `map` `x` and the operation `x * 2` to a `Worker Node` and `y` and `y + 2` to another `Worker Node`.

Think about this but on a larger scale, we have 1 billion rows of numbers that we want to increment by 1. We will map portions of the data to each `Worker Node` and the operation `+ 1` to each `Worker Node`.

Reduce

When work can't be done in parallel and is dependent on some previous work, the post transformed `data` is sent back to the `Driver Node` from all the `Worker Nodes`. There the new data may be redistributed to the `Worker Nodes` to resume execution or execution is done on the `Driver Node` depending on the type of work.

Ex.

```
# initialize your variables
x = 5
y = 10

# do some transformations to your variables
x = x * 2
y = y + 2

# do some more transformations
z = x + y
```

Similar to the example above, but here we see that the last transformation `z = x + y` depends on the previous transformations. So we will need to collect all the work done on the `Worker Nodes` to the `Driver Node` and perform the final transformation.

Key Terms

Driver Node

Learnt above.

Worker

Learnt above.

Executor

A process launched from an application on a `Worker Node`, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

Jobs

Job A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action.

Stages

Smaller set of tasks inside any job.

Tasks

Unit of work that will be sent to one executor.

Source

Source

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Initial Datasets

```
pets = spark.createDataFrame(
    [
        (1, 1, 'Bear'),
        (2, 1, 'Chewie'),
        (3, 2, 'Roger'),
    ], ['id', 'breed_id', 'nickname']
)

pets.toPandas()
```

	id	breed_id	nickname
0	1	1	Bear
1	2	1	Chewie
2	3	2	Roger

```
breeds = spark.createDataFrame(
    [
        (1, 'Pitbull', 10),
        (2, 'Corgie', 20),
    ], ['breed_id', 'name', 'average_height']
)

breeds.toPandas()
```

	breed_id	name	average_height

0	1	Pitbull	10
1	2	Corgie	20

Filter Pushdown

> `Filter pushdown` improves performance by reducing the amount of data shuffled during any dataframes transformations.

Depending on your filter logic and where you place your filter code. Your Spark code will behave differently.

Case #1: Filtering on Only One Side of the Join

```
df = (
    pets
    .join(breeds, 'breed_id', 'left_outer')
    .filter(F.col('nickname') == 'Chewie')
)

df.toPandas()
```

	breed_id	id	nickname	name	average_height
0	1	2	Chewie	Pitbull	10

```
df.explain()
```

```
== Physical Plan ==
*(4) Project [breed_id#1L, id#0L, nickname#2, name#7, average_height#8L]
+- SortMergeJoin [breed_id#1L], [breed_id#6L], LeftOuter
   :- *(2) Sort [breed_id#1L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(breed_id#1L, 200)
   :     +- *(1) Filter (isnotnull(nickname#2) && (nickname#2 = Chewie))
   :        +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2]
   +- *(3) Sort [breed_id#6L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(breed_id#6L, 200)
         +- Scan ExistingRDD[breed_id#6L,name#7,average_height#8L]
```

What Happened:

Because the column `nickname` is only present in the `left` side of the join, only the `left` side of the join was `filtered` before the join.

Case #2: Filter on Both Sides of the Join

```
df = (
    pets
```

```
    .join(breeds, 'breed_id', 'left_outer')
    .filter(F.col('breed_id') == 1)
)

df.toPandas()
```

	breed_id	id	nickname	name	average_height
0	1	1	Bear	Pitbull	10
1	1	2	Chewie	Pitbull	10

```
df.explain()
```

```
== Physical Plan ==
*(4) Project [breed_id#1L, id#0L, nickname#2, name#7, average_height#8L]
+- SortMergeJoin [breed_id#1L], [breed_id#6L], LeftOuter
   :- *(2) Sort [breed_id#1L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(breed_id#1L, 200)
   :     +- *(1) Filter (isnotnull(breed_id#1L) && (breed_id#1L = 1))
   :        +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2]
   +- *(3) Sort [breed_id#6L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(breed_id#6L, 200)
         +- Scan ExistingRDD[breed_id#6L,name#7,average_height#8L]
```

What Happened:

The column `breed_id` is present in `both` sides of the join, but only the `left` side was `filtered` before the join.

Case #3: Filter on Both Sides of the Join #2

```
df = (
    pets
    .join(breeds, 'breed_id')
    .filter(F.col('breed_id') == 1)
)

df.toPandas()
```

	breed_id	id	nickname	name	average_height
0	1	1	Bear	Pitbull	10
1	1	2	Chewie	Pitbull	10

```
df.explain()
```

```
== Physical Plan ==
```

```
*(5) Project [breed_id#1L, id#0L, nickname#2, name#7, average_height#8L]
+- *(5) SortMergeJoin [breed_id#1L], [breed_id#6L], Inner
   :- *(2) Sort [breed_id#1L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(breed_id#1L, 200)
   :     +- *(1) Filter (isnotnull(breed_id#1L) && (breed_id#1L = 1))
   :        +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2]
   +- *(4) Sort [breed_id#6L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(breed_id#6L, 200)
         +- *(3) Filter (isnotnull(breed_id#6L) && (breed_id#6L = 1))
            +- Scan ExistingRDD[breed_id#6L,name#7,average_height#8L]
```

What Happened:

The column `breed_id` is present in `both` sides of the join, and spark was able to figure out that it should perform a `filter` on both sides before the join.

Case #4: Filter on Both Sides of the Join, Filter Beforehand

```
df = (
    pets
    .join(
        breeds.filter(F.col('breed_id') == 1),
        'breed_id',
        'left_outer'
    )
    .filter(F.col('breed_id') == 1)
)

df.toPandas()
```

	breed_id	id	nickname	name	average_height
0	1	1	Bear	Pitbull	10
1	1	2	Chewie	Pitbull	10

```
df.explain()
```

```
== Physical Plan ==
*(5) Project [breed_id#1L, id#0L, nickname#2, name#7, average_height#8L]
+- SortMergeJoin [breed_id#1L], [breed_id#6L], LeftOuter
   :- *(2) Sort [breed_id#1L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(breed_id#1L, 200)
   :     +- *(1) Filter (isnotnull(breed_id#1L) && (breed_id#1L = 1))
   :        +- Scan ExistingRDD[id#0L,breed_id#1L,nickname#2]
   +- *(4) Sort [breed_id#6L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(breed_id#6L, 200)
         +- *(3) Filter (isnotnull(breed_id#6L) && (breed_id#6L = 1))
            +- Scan ExistingRDD[breed_id#6L,name#7,average_height#8L]
```

What Happened:

The column `breed_id` is present in `both` sides of the join, and both sides were `filtered` before the join.

Summary

- To improve join performance, we should always try to push the `filter` before the joins.
- Spark might be smart enough to figure that the `filter` can be performed on both sides, but not always.
- You should alway check to see if your Spark DAG is performant during a join and if any `filter`s can be pushed before the joins.

A `skewed dataset` is defined by a dataset that has a class imbalance, this leads to poor or failing spark jobs that often get a `OOM` (out of memory) error.

When performing a `join` onto a `skewed dataset` it's usually the case where there is an imbalance on the `key` (s) on which the join is performed on. This results in a majority of the data falls onto a single partition, which will take longer to complete than the other partitions.

Some hints to detect skewness is:

1. The `key` (s) consist mainly of `null` values which fall onto a single partition.
2. There is a subset of values for the `key` (s) that makeup the high percentage of the total keys which fall onto a single partition.

We go through both these cases and see how we can combat it.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Situation 1: Null Keys

Inital Datasets

```
customers = spark.createDataFrame([
    (1, None),
    (2, None),
    (3, 1),
], ["id", "card_id"])

customers.toPandas()
```

	id	card_id
0	1	NaN
1	2	NaN

2	3	1.0

```
cards = spark.createDataFrame([
    (1, "john", "doe", 21),
    (2, "rick", "roll", 10),
    (3, "bob", "brown", 2)
], ["card_id", "first_name", "last_name", "age"])

cards.toPandas()
```

	card_id	first_name	last_name	age
0	1	john	doe	21
1	2	rick	roll	10
2	3	bob	brown	2

Option #1: Join Regularly

```
df = customers.join(cards, "card_id", "left")

df.toPandas()
```

	card_id	id	first_name	last_name	age
0	NaN	1	None	None	NaN
1	NaN	2	None	None	NaN
2	1.0	3	john	doe	21.0

```
df = customers.join(cards, "card_id")

df.toPandas()
```

	card_id	id	first_name	last_name	age
0	1	3	john	doe	21

What Happened:

- Rows that didn't join up were brought to the join.
- For a `left join`, they will get `Null` values for the right side columns, what's the point of being them in?
- For a `inner join`, they rows will get dropped, so again what's the point of being them in?

Results:

- We brought more rows to the join than we had to. These rows get normally get put onto a single partition.
- If the data is large enough and the percentage of keys that are null is high. The program could OOM out.

Option #2: Filter Null Keys First, then Join, then Union

```
def null_skew_helper(left, right, key):
    """
    Steps:
        1. Filter out the null rows.
        2. Create the columns you would get from the join.
        3. Join the tables.
        4. Union the null rows to joined table.
    """
    df1 = left.where(F.col(key).isNull())
    for f in right.schema.fields:
            df1 = df1.withColumn(f.name, F.lit(None).cast(f.dataType))

    df2 = left.where(F.col(key).isNotNull())
    df2 = df2.join(right, key, "left")

    return df1.union(df2.select(df1.columns))


df = null_skew_helper(customers, cards, "card_id")

df.toPandas()
```

	id	card_id	first_name	last_name	age
0	1	NaN	None	None	NaN
1	2	NaN	None	None	NaN
2	3	1.0	john	doe	21.0

```
df.explain()
```

```
== Physical Plan ==
Union
:- *(1) Project [id#0L, null AS card_id#23L, null AS first_name#26, null AS last_name#30,
null AS age#35L]
:  +- *(1) Filter isnull(card_id#1L)
:     +- Scan ExistingRDD[id#0L,card_id#1L]
+- *(5) Project [id#0L, card_id#1L, first_name#5, last_name#6, age#7L]
   +- SortMergeJoin [card_id#1L], [card_id#4L], LeftOuter
      :- *(3) Sort [card_id#1L ASC NULLS FIRST], false, 0
      :  +- Exchange hashpartitioning(card_id#1L, 200)
      :     +- *(2) Filter isnotnull(card_id#1L)
      :        +- Scan ExistingRDD[id#0L,card_id#1L]
      +- *(4) Sort [card_id#4L ASC NULLS FIRST], false, 0
```

```
          +- Exchange hashpartitioning(card_id#4L, 200)
             +- Scan ExistingRDD[card_id#4L,first_name#5,last_name#6,age#7L]
```

What Happened:

- We seperated the data into 2 sets:
  - one where the `key`s are not `null`.
  - one where the `key`s are `null`.
- We perform the join on the set where the keys are not null, then union it back with the set where the keys are null. (This step is not necessary when doing an inner join).

Results:

- We brought less data to the join.
- We read the data twice; more time was spent on reading data from disk.

Option #3: Cache the Table, Filter Null Keys First, then Join, then Union

Helper Function

```
def null_skew_helper(left, right, key):
    """
    Steps:
        1. Cache table.
        2. Filter out the null rows.
        3. Create the columns you would get from the join.
        4. Join the tables.
        5. Union the null rows to joined table.
    """
    left = left.cache()

    df1 = left.where(F.col(key).isNull())
    for f in right.schema.fields:
            df1 = df1.withColumn(f.name, F.lit(None).cast(f.dataType))

    df2 = left.where(F.col(key).isNotNull())
    df2 = df2.join(right, key, "left")

    return df1.union(df2.select(df1.columns))
```

```
df = null_skew_helper(customers, cards, "card_id")

df.toPandas()
```

	id	card_id	first_name	last_name	age
0	1	NaN	None	None	NaN
1	2	NaN	None	None	NaN

2	3	1.0	john	doe	21.0

```
df.explain()
```

```
== Physical Plan ==
Union
:- *(1) Project [id#0L, null AS card_id#68L, null AS first_name#71, null AS last_name#75,
null AS age#80L]
:  +- *(1) Filter isnull(card_id#1L)
:     +- *(1) InMemoryTableScan [card_id#1L, id#0L], [isnull(card_id#1L)]
:           +- InMemoryRelation [id#0L, card_id#1L], true, 10000, StorageLevel(disk, memor
y, deserialized, 1 replicas)
:                 +- Scan ExistingRDD[id#0L,card_id#1L]
+- *(5) Project [id#0L, card_id#1L, first_name#5, last_name#6, age#7L]
   +- SortMergeJoin [card_id#1L], [card_id#4L], LeftOuter
      :- *(3) Sort [card_id#1L ASC NULLS FIRST], false, 0
      :  +- Exchange hashpartitioning(card_id#1L, 200)
      :     +- *(2) Filter isnotnull(card_id#1L)
      :        +- *(2) InMemoryTableScan [id#0L, card_id#1L], [isnotnull(card_id#1L)]
      :              +- InMemoryRelation [id#0L, card_id#1L], true, 10000, StorageLevel(di
sk, memory, deserialized, 1 replicas)
      :                    +- Scan ExistingRDD[id#0L,card_id#1L]
      +- *(4) Sort [card_id#4L ASC NULLS FIRST], false, 0
         +- Exchange hashpartitioning(card_id#4L, 200)
            +- Scan ExistingRDD[card_id#4L,first_name#5,last_name#6,age#7L]
```

What Happened:

- Similar to option #2, but we did a `InMemoryTableScan` instead of two reads of the data.

Results:

- We brought less data to the join.
- We did 1 less read, but we used more memory.

Summary

All to say:

- It's definitely better to bring less data to a join, so performing a filter for `null keys` before the join is definitely suggested.
- For `left join` s:
  - By doing a union, this will result in an extra read of data or memory usage.
  - Decide what you can afford; the extra read vs memory usage and `cache` the table before the `filter` .

Always check the spread the values for the `join key` , to detect if there's any skew and pre filters that can be performed.

A `skewed dataset` is defined by a dataset that has a class imbalance, this leads to poor or failing spark jobs that often get a `OOM` (out of memory) error.

When performing a `join` onto a `skewed dataset` it's usually the case where there is an imbalance on the `key` (s) on which the join is performed on. This results in a majority of the data falls onto a single partition, which will take longer to complete than the other partitions.

Some hints to detect skewness is:

1. The `key` (s) consist mainly of `null` values which fall onto a single partition.
2. There is a subset of values for the `key` (s) that makeup the high percentage of the total keys which fall onto a single partition.

We go through both these cases and see how we can combat it.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Situation 2: High Frequency Keys

Inital Datasets

```
customers = spark.createDataFrame([
    (1, "John"),
    (2, "Bob"),
], ["customer_id", "first_name"])

customers.toPandas()
```

	customer_id	first_name
0	1	John
1	2	Bob

```
orders = spark.createDataFrame([
    (i, 1 if i < 95 else 2, "order #{}".format(i)) for i in range(100)
], ["id", "customer_id", "order_name"])

orders.toPandas().tail(6)
```

	id	customer_id	order_name
94	94	1	order #94
95	95	2	order #95
96	96	2	order #96
97	97	2	order #97
98	98	2	order #98
99	99	2	order #99

Option 1: Inner Join

```
df = customers.join(orders, "customer_id")

df.toPandas().tail(10)
```

	customer_id	first_name	id	order_name
90	1	John	90	order #90
91	1	John	91	order #91
92	1	John	92	order #92
93	1	John	93	order #93
94	1	John	94	order #94
95	2	Bob	95	order #95
96	2	Bob	96	order #96
97	2	Bob	97	order #97
98	2	Bob	98	order #98
99	2	Bob	99	order #99

```
df.explain()
```

```
== Physical Plan ==
*(5) Project [customer_id#122L, first_name#123, id#126L, order_name#128]
+- *(5) SortMergeJoin [customer_id#122L], [customer_id#127L], Inner
   :- *(2) Sort [customer_id#122L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(customer_id#122L, 200)
```

```
:     +- *(1) Filter isnotnull(customer_id#122L)
:        +- Scan ExistingRDD[customer_id#122L,first_name#123]
+- *(4) Sort [customer_id#127L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(customer_id#127L, 200)
      +- *(3) Filter isnotnull(customer_id#127L)
         +- Scan ExistingRDD[id#126L,customer_id#127L,order_name#128]
```

What Happened:

- We want to find what `order`s each `customer` made, so we will be `join`ing the `customer`s table to the `order`s table.
- When performing the join, we perform a `hashpartitioning` on `customer_id`.
- From our data creation, this means 95% of the data landed onto a single partition.

Results:

- Similar to the `Null Skew` case, this means that single task/partition will take a lot longer than the others, and most likely erroring out.

Option 2: Salt the key, then Join

Helper Function

```
def data_skew_helper(left, right, key, number_of_partitions, how="inner"):
    salt_value = F.lit(F.rand() * number_of_partitions % number_of_partitions).cast('int')
    left = left.withColumn("salt", salt_value)

    salt_col = F.explode(F.array([F.lit(i) for i in range(number_of_partitions)])).alias("
salt")
    right = right.select("*",  salt_col)

    return left.join(right, [key, "salt"], how).drop("salt")
```

Example

```
num_of_partitions = 5
```

```
left = customers

salt_value = F.lit(F.rand() * num_of_partitions % num_of_partitions).cast('int')
left = left.withColumn("salt", salt_value)

left.toPandas().head(5)
```

	customer_id	first_name	salt
0	1	John	4
1	2	Bob	3

```
right = orders

salt_col = F.explode(F.array([F.lit(i) for i in range(num_of_partitions)])).alias("salt")
right = right.select("*",  salt_col)

right.toPandas().head(10)
```

	id	customer_id	order_name	salt
0	0	1	order #0	0
1	0	1	order #0	1
2	0	1	order #0	2
3	0	1	order #0	3
4	0	1	order #0	4
5	1	1	order #1	0
6	1	1	order #1	1
7	1	1	order #1	2
8	1	1	order #1	3
9	1	1	order #1	4

```
df = left.join(right, ["customer_id", "salt"])

df.orderBy('id').toPandas().tail(10)
```

	customer_id	salt	first_name	id	order_name
90	1	4	John	90	order #90
91	1	4	John	91	order #91
92	1	4	John	92	order #92
93	1	4	John	93	order #93
94	1	4	John	94	order #94
95	2	3	Bob	95	order #95
96	2	3	Bob	96	order #96
97	2	3	Bob	97	order #97
98	2	3	Bob	98	order #98
99	2	3	Bob	99	order #99

```
df.explain()
```

```
== Physical Plan ==
*(5) Project [customer_id#122L, salt#136, first_name#123, id#126L, order_name#128]
+- *(5) SortMergeJoin [customer_id#122L, salt#136], [customer_id#127L, salt#141], Inner
   :- *(2) Sort [customer_id#122L ASC NULLS FIRST, salt#136 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(customer_id#122L, salt#136, 200)
   :     +- *(1) Filter (isnotnull(salt#136) && isnotnull(customer_id#122L))
   :        +- *(1) Project [customer_id#122L, first_name#123, cast(((rand(-80401295512237
67613) * 5.0) % 5.0) as int) AS salt#136]
   :           +- Scan ExistingRDD[customer_id#122L,first_name#123]
   +- *(4) Sort [customer_id#127L ASC NULLS FIRST, salt#141 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(customer_id#127L, salt#141, 200)
         +- Generate explode([0,1,2,3,4]), [id#126L, customer_id#127L, order_name#128], fa
lse, [salt#141]
            +- *(3) Filter isnotnull(customer_id#127L)
               +- Scan ExistingRDD[id#126L,customer_id#127L,order_name#128]
```

What Happened:

- We created a new `salt` column for both datasets.
- On one of the dataset we duplicate the data so we have a row for each `salt` value.
- When performing the join, we perform a `hashpartitioning` on `[customer_id, salt]`.

Results:

- When we produce a row per `salt` value, we have essentially duplicated `(num_partitions - 1) * N` rows.
- This created more data, but allowed us to spread the data across more partitions as you can see from `hashpartitioning(customer_id, salt)`.

Summary

All to say:

- By `salt`ing our keys, the `skewed` dataset gets divided into smaller partitions. Thus removing the skew.
- Again we will sacrifice more resources in order to get a performance gain or a successful run.
- We produced more data by creating `(num_partitions - 1) * N` more data for the right side.

A `skewed dataset` is defined by a dataset that has a class imbalance, this leads to poor or failing spark jobs that often get a `OOM` (out of memory) error.

When performing a `join` onto a `skewed dataset` it's usually the case where there is an imbalance on the `key` (s) on which the join is performed on. This results in a majority of the data falls onto a single partition, which will take longer to complete than the other partitions.

Some hints to detect skewness is:

1. The `key` (s) consist mainly of `null` values which fall onto a single partition.
2. There is a subset of values for the `key` (s) that makeup the high percentage of the total keys which fall onto a single partition.

We go through both these cases and see how we can combat it.

Library Imports

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

Template

```
spark = (
    SparkSession.builder
    .master("local")
    .appName("Exploring Joins")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
)

sc = spark.sparkContext
```

Situation 2: High Frequency Keys

Inital Datasets

```
customers = spark.createDataFrame([
    (1, "John"),
    (2, "Bob"),
], ["customer_id", "first_name"])

customers.toPandas()
```

	customer_id	first_name
0	1	John
1	2	Bob

```
orders = spark.createDataFrame([
    (i, 1 if i < 95 else 2, "order #{}".format(i)) for i in range(100)
], ["id", "customer_id", "order_name"])

orders.toPandas().tail(6)
```

	id	customer_id	order_name
94	94	1	order #94
95	95	2	order #95
96	96	2	order #96
97	97	2	order #97
98	98	2	order #98
99	99	2	order #99

Option 1: Inner Join

```
df = customers.join(orders, "customer_id")

df.toPandas().tail(10)
```

	customer_id	first_name	id	order_name
90	1	John	90	order #90
91	1	John	91	order #91
92	1	John	92	order #92
93	1	John	93	order #93
94	1	John	94	order #94
95	2	Bob	95	order #95
96	2	Bob	96	order #96
97	2	Bob	97	order #97
98	2	Bob	98	order #98
99	2	Bob	99	order #99

```
df.explain()
```

```
== Physical Plan ==
*(5) Project [customer_id#122L, first_name#123, id#126L, order_name#128]
+- *(5) SortMergeJoin [customer_id#122L], [customer_id#127L], Inner
   :- *(2) Sort [customer_id#122L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(customer_id#122L, 200)
```

```
:     +- *(1) Filter isnotnull(customer_id#122L)
:        +- Scan ExistingRDD[customer_id#122L,first_name#123]
+- *(4) Sort [customer_id#127L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(customer_id#127L, 200)
      +- *(3) Filter isnotnull(customer_id#127L)
         +- Scan ExistingRDD[id#126L,customer_id#127L,order_name#128]
```

What Happened:

- We want to find what `order`s each `customer` made, so we will be `join`ing the `customer`s table to the `order`s table.
- When performing the join, we perform a `hashpartitioning` on `customer_id`.
- From our data creation, this means 95% of the data landed onto a single partition.

Results:

- Similar to the `Null Skew` case, this means that single task/partition will take a lot longer than the others, and most likely erroring out.

Option 2: Split the DataFrame in 2 Sections, High Frequency and Non-High Frequency values