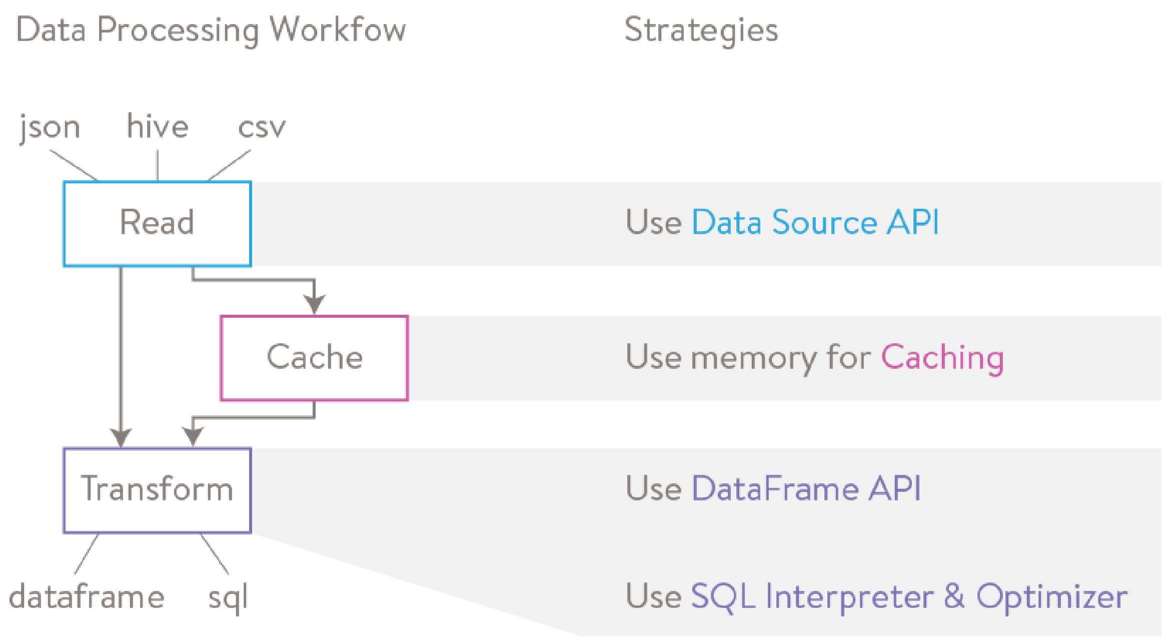
& Performance in Spark For Data Scientists

JAS KHELA

October 06, 2015 - San Francisco

Spark is a cluster computing framework that can significantly increase the efficiency and capabilities of a data scientist's workflow when dealing with distributed data. However, deciding which of its many modules, features and options are appropriate for a given problem can be cumbersome. Our experience at Stitch Fix has shown that these decisions can have a large impact on development time and performance. This post will discuss strategies at each stage of the data processing workflow which data scientists new to Spark should consider employing for high productivity development on big data.

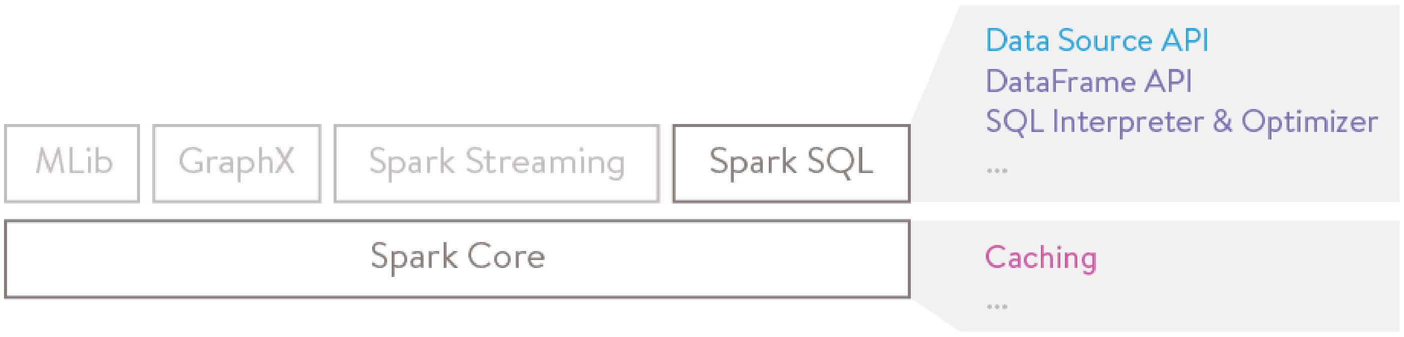


When to Use Spark

A common task for a data scientist is to manipulate, aggregate, or transform raw data into a format that can be passed into machine learning models. As data outgrows traditional single-node databases, it becomes necessary to move to distributed platforms such as Hadoop. However, this is often at great cost in terms of productivity due to increased latency, reduced built-in query optimization, and more complex coding, as in the case of MapReduce. We have found Spark to be an excellent option as a scalable distributed computing system. It mitigates productivity tradeoffs by performing many tasks in-memory and provides a data scientist friendly DataFrame API, as well as integration with both R & Python. The result of this is a smooth, fast, and familiar development experience.

The Spark Ecosystem

The Spark ecosystem includes many components, including Spark SQL for querying, MLlib for machine learning, and GraphX for representing data as a graph. The ecosystem makes Spark a solution for the entire data processing pipeline . This article will focus on Spark SQL and its accompanying APIs.



Reading Data With Data Source API

The starting point for any data processing workflow is to read in data. The Spark data source API provides the ability to read structured data from a large number of input sources, including Hive, JSON, and CSV. Because the API comes equipped with deep integration it can eliminate columns or filter out rows from source data. This is an important feature since it can avoid a lot of unnecessary reading. The examples below show how to read in data and run basic data frame operations using a few different sources in python.

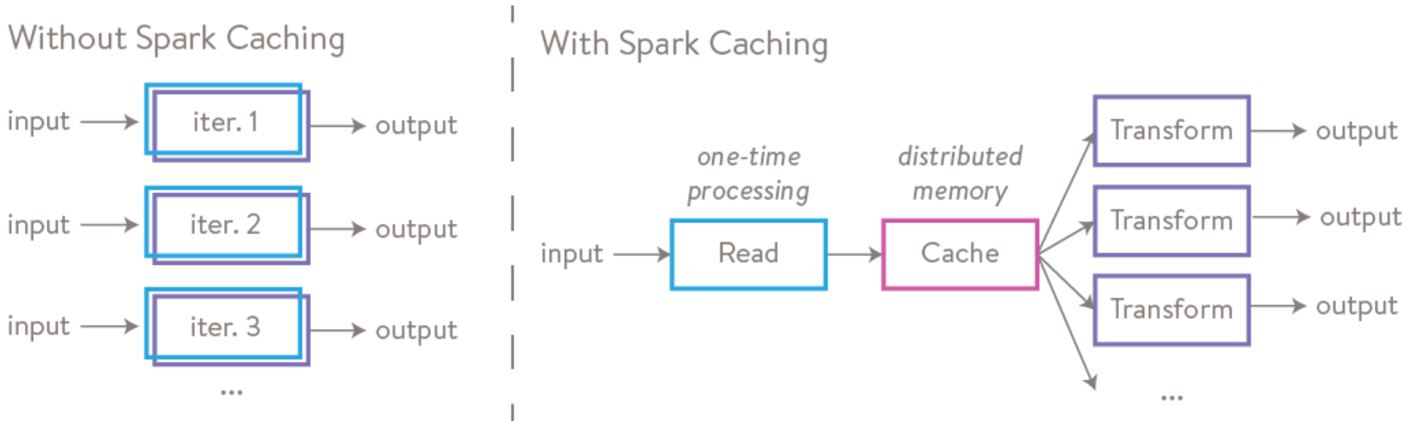
```
# Read from Hive (note: sc is an existing SparkContext)
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)
df_hive = sqlContext.sql("SELECT client, timestamp, items FROM client_table")
df_hive.show() # Show first rows of dataframe
df_hive.printSchema() # Show column names and types

# Read from JSON
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df_json = sqlContext.read.json("path/client_table.json")
df_json.show()
df_json.printSchema()
```

```
# Read from CSV (Note: Requires installation of spark-csv lib)
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df_csv = sqlContext.read.format('com.databricks.spark.csv').options(header='true')
                             .load('client_table.csv').select('client', 'timestamp')
df_csv.show()
df_csv.printSchema()
```

Caching Data

After data is read into Spark, one strategy to improve workload performance is to cache tables in memory. Only the first iteration of a task needs to be read from disk (which is slow) and all subsequent iterations can be read from memory (which is fast).



Although caching data is an excellent way to improve performance, it is also important to be strategic about what data is cached. If you run out of memory, just as on your laptop, data will be expelled from memory back to disk. As a general rule of thumb, one should only cache data when it will be used in an iterative algorithm or multiple downstream processes. The example below shows how caching can be used in an iterative function.

```
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)
df_hive = sqlContext.sql("SELECT client, timestamp, items FROM client_table")
```

```
df_hive.cache() # cache table in memory
N = 100 # number of iterations
value = 0 # starting value for function
for i in range(N):
    new_value = my_iter_func(df_hive, value)
           value = new_value
print 'final function output is: {value}'.format(value = value)
```

Transforming Data using DataFrame API

After reading and caching data in Spark, the next step is to transform raw data. An excellent way to improve development time and speed for this task is to use Spark dataframes. Data frames are a relational representation of data popularized by R, and implemented in Python through the Pandas library. Spark has its own implementation, providing some (though not all) of the programming expressiveness of R or Pandas dataframes in a distributed context. There are two major advantages:

1. Spark dataframes abstract over the processing implementation, allowing data scientists to *perform map/reduce tasks* without explicitly *writing map/reduce commands*. This results in code that is more succinct, more intuitive, and ultimately quicker to develop for data scientists. The example below compares writing a simple count and group by statement using dataframes vs the basic Spark data abstraction called an RDD.

```
# Transformation Using RDDs
data_rdd = sqlContext.sql("SELECT client, timestamp, items from client_table").rdd
data_rdd.map(lambda x: (x[0], 1))
              .reduceByKey(lambda x, y: x + y)
              .map(lambda x: ( x[1], x[0] ))
              .sortByKey(False)
              .collect()

# Transformation Using DataFrames
data = sqlContext.sql("SELECT client, timestamp, items from client_table")
data.groupBy("client")
```

```
        .count()
        .orderBy("count")
        .show()
```

2. Not only are Spark dataframes a familiar way for data scientists to transform data, they are also *significantly* faster than using RDDs. Spark dataframes are highly optimized by a query optimizer. Additionally, because of the way in which execution code is generated, Python users experience the same level of performance as Scala & Java. Here at Stitch Fix, we have converted Python jobs that had previously used RDDs into jobs that use Spark dataframes and have seen 6X performance improvement.

Transforming Data using SQL interpreter and Optimizer

Data frames are useful for analytics in Spark, but Spark can also transform data using sql syntax. To do this in SparkSQL, start by caching the data frame in memory as a table using the 'registerTempTable()' command. Once cached, the table can be queried like a standard table in a relational database. This is a great option for users who feel more comfortable using SQL. It also gives data scientists a second method for computing transformations - a nice flexibility to have, given some transformations lend themselves best to dataframe operations while others are easier to do in SQL.

```
#Using Hive
data = sqlContext.sql("SELECT client, timestamp, items FROM client_table")
data.registerTempTable("data_t")
sqlContext.sql("SELECT client, count(*) FROM data_t").collect()

#Using csv as data source
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
df = sqlContext.read.format('com.databricks.spark.csv').options(header='true')
```

```
                         .load('client_table.csv')
df.registerTempTable("df_t")
sqlContext.sql("SELECT client, count(*) FROM df_t").collect()
```

Conclusion

Spark has evolved into a platform that is well suited to serve the needs of data scientists. With APIs for both R and Python, users will find it easy to integrate Spark into their jobs. The addition of data frames has added the capability to transform data in a familiar way and at high speeds. It is precisely for these reasons that we have employed Spark for many tasks here at Stitch Fix. As we endeavor to understand our clients at a deeper level, the requirement to capture, process, and model large amounts of data will continue to grow. We adopted Spark to get us a step closer to that goal as we are now able to process greater amounts of information and turn that into a more personalized Stitch Fix experience than even before.

About Jas

Jas received his Masters in Financial Engineering from UCLA where he studied the application of computational and mathematical methods to finance. While he enjoyed the challenge of predicting financial markets, the allure of disrupting the fashion space was too intriguing. In his spare time Jas enjoys traveling, fantasy sports, and contemplating how he can put a Spark in each client's fix.