RDDS ARE THE NEW BYTECODE OF APACHE SPARK

29/05/2015 · *par* ogirardot · *in Apache Spark, BigData, Data* · *3 commentaires*

With the Apache Spark 1.3 release the Dataframe API for Spark SQL got introduced, for those of you who missed the big announcements, I'd recommend to read the article : Introducing Dataframes in Spark for Large Scale Data Science from the Databricks blog. Dataframes are very popular among data scientists, personally I've mainly been using them with the great Python library Pandas but there are many examples in R (originally) and Julia.

Of course if you're using only Spark's core features, nothing seems to have changed with Spark 1.3 : Spark's main abstraction remains the RDD (Resilient Distributed Dataset), its API is very stable now and everyone used it to handle any kind of data since now.

But the introduction of Dataframe is actually a big deal, because when RDDs were the only option to load data, it was obvious that you needed to parse your « maybe » un-structured data using RDDs, transform them using case-classes or tuples and then do the special work that you actually needed. Spark SQL is not a new project and you were, of course, able to load your structured-data (like Parquet files) directly from a SQLContext before 1.3 – but the advantages were not that clear at the time – except if you wanted to run SQL queries or expose a JDBC-compatible server for other BI tools.

Now the advantages are quite clear and I'll try to explain them as simply as possible :

1) Dataframes are a higher level of abstraction than RDDs

If you're familiar with Pandas syntax, you will feel at home using Spark's Dataframe and even if you're not, you'll learn and – I'd even add – learn to love it. Why ? Because it's a higher level of programming than the RDD, you can do more, faster (old joke now 😉). Here's an example from Patrick Wendell's Strata London 2015 presentation « What's coming in Spark » of RDDs in Python vs Dataframe :

Of course the second way of writing it is obviously more concise and more understandable, but I'd like to add something else, the *tried-and-tested* Spark programmers have surely noticed the **reduceByKey** transformation used here. It is a very common mistake in Spark for common aggregation tasks to use the **groupBy** then **mapValues** or **map** transformation which can be dangerous in a production environment and produce **OutOfMemory** errors on workers.

Do you notice that such a mistake **cannot** happen using the Dataframe API below for you will be expressing your aggregations using, for example, the **agg(...)** method (or even directly the **avg(...)** method like up there). This will even allow you to define multiple aggregations at the same time, something that is usually tricky using RDDs :

```
case class Person(id: Int, first_name: String, last_name: String, age: Double)

// get simple stats on age repartitions by first_name(min, max, avg, count)
val rdd: RDD[Person] = ...
// first you need to only handle the data you really need, and cache it because you'll - sadly
val persons = rdd.map(person => (person.first_name, person.age)).cache()

val minAgeByFirstName = persons.reduceByKey( scala.math.min(_, _) )
val maxAgeByFirstName = persons.reduceByKey( scala.math.max(_, _) )
val avgAgeByFirstName = persons.mapValues(x => (x, 1))
    .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2)) // simple right ?
val countByFirstName =  persons.mapValues(x => 1).reduceByKey(_ + _)
```

Without even trying to consider the complexity of all I had to write to get all my answers – answers that I would need to join back if I want a consistent RDD with all the informations I need – the most painful point is that I had to duplicate all these aggregations and therefore **cache** my dataset to mitigate the damages.

Now using the dataframe API, I get to leverage out-of-the-box functions and I can even reuse my computations afterward without having to join-back anything :

```
case class Person(id: Int, first_name: String, last_name: String, age: Double)

// get simple stats on age repartitions by first_name(min, max, avg, count)
val df: Dataframe = ...
persons = df.groupBy("first_name").agg(
  min("age").alias("min_age"),
  max("age").alias("max_age"),
  avg("age").alias("average_age"),
  count("*").alias("number_of_persons")
)
// let's add a new column to our schema re-using the two last-computed aggregations :
val finalDf = persons.withColumn("age_delta", persons("max_age") - persons("min_age"))
```

This is a higher level of programming than RDDs, so some things might be more difficult to express with Dataframe than they were using RDDs when you could **groupBy(...)** anything and get the *List[...]* of result as values... But this was a bad practice anyway 🙂.

2) Spark SQL/Catalyst is more intelligent than you

When you're using Dataframe, you're not defining directly a DAG (Directed Acyclic Graph) anymore, you're actually creating an AST (Abstract Syntax Tree) that the Catalyst engine will parse, check and improve using both Rules-Based Optimisation and Cost-Based Optimisation. This is an excerpt from the Spark SQL paper submitted for SIGMOD 2015 :

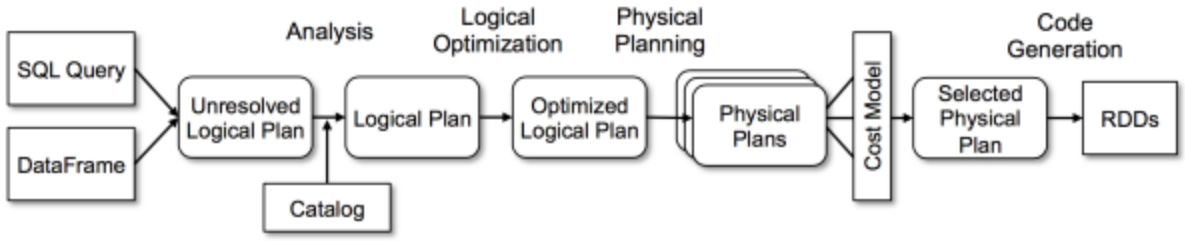


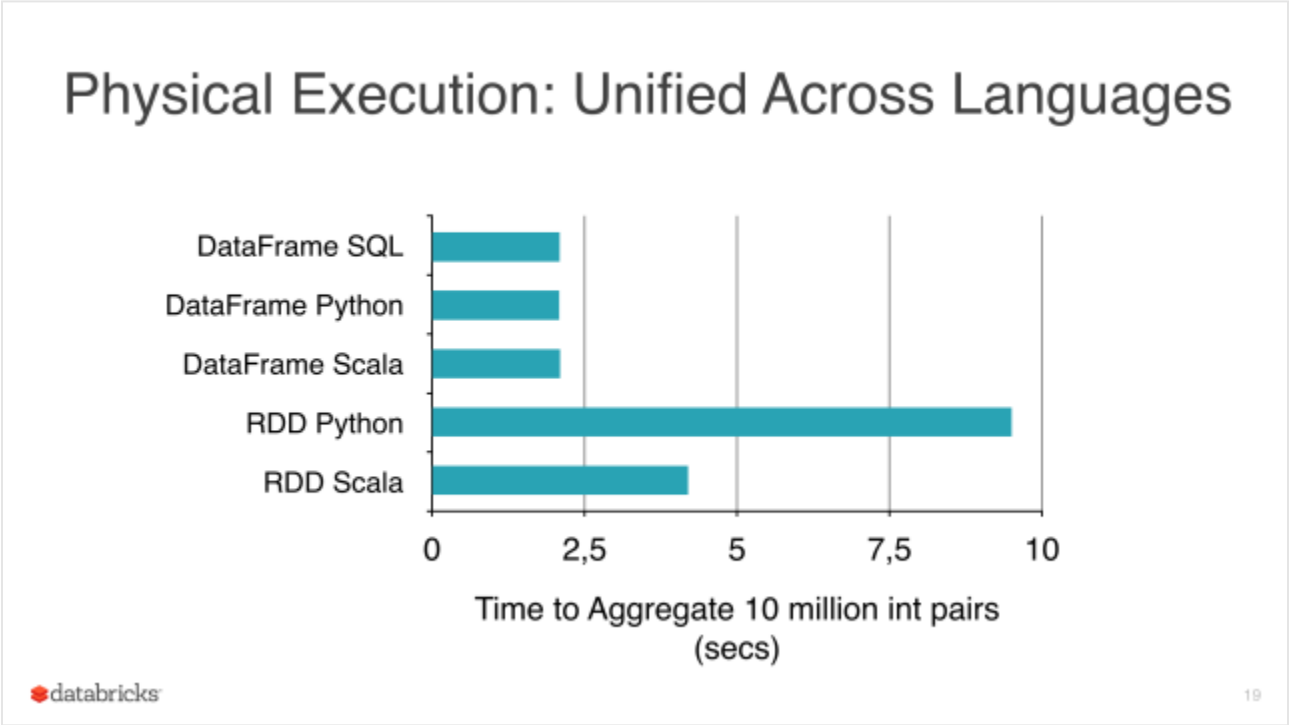
Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

I won't get into the depth of this here, because that would even need more than one full article about it, but if you want to understand more this article Deep dive into Spark SQL's Catalyst optimizer from the Databricks blog (once again) will give you insights into how this works. A simple rule if thumb to get is that a lot of « pretty logical » generic tree-based rules will be used to check and simplify your parsed-Logical Plan and then a few Physical Plans representing different executions strategies will be computed and one will be selected according to their « computation cost ».

The funny thing is that in the end – nothing changes – after all these transformations your Dataframe will get *compiled* down to RDDs and executed on your Spark Cluster.

3) Python & Scala are now even in terms of performance

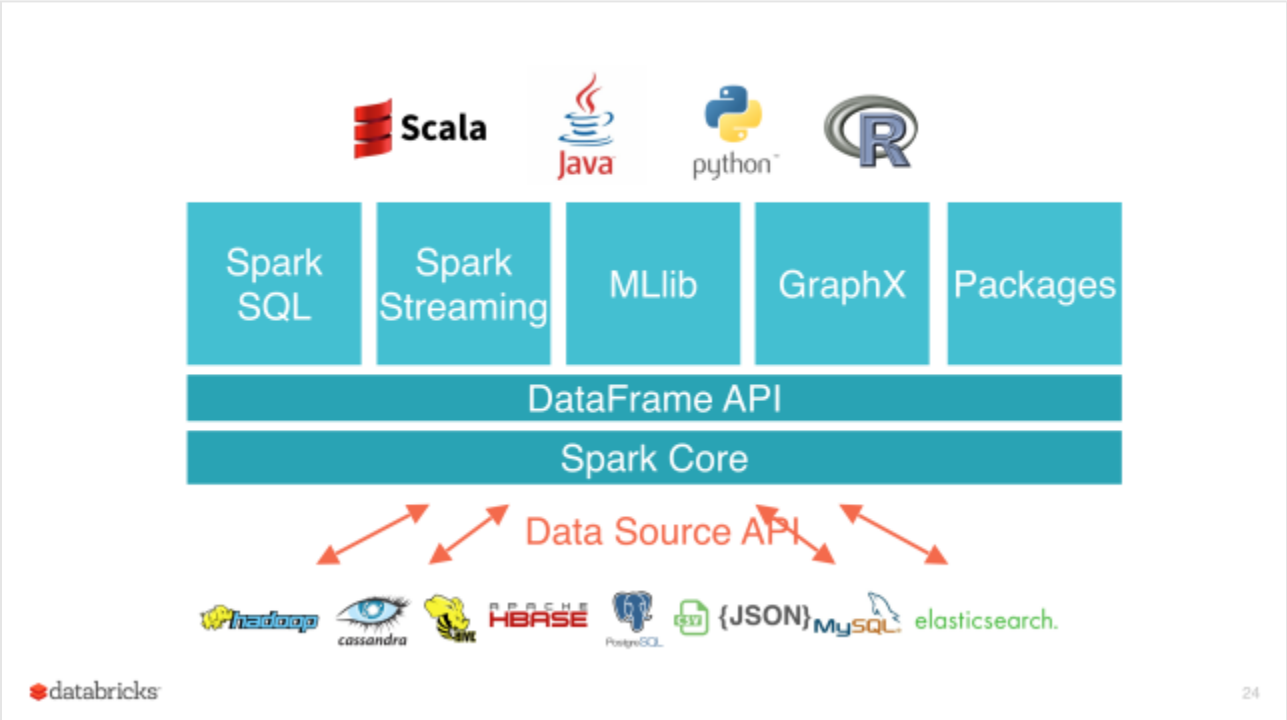
Using the Dataframe API, you're using a DSL that leverages Spark's Scala bytecode – when using RDDs, Python lambdas will run in a Python VM, Java/Scala lambdas will run in the JVM, this is great because inside RDDs you can use your usual Python libraries (Numpy, Scipy, etc...) and not some Jython code, but it comes at a performance cost :



This is still true if you want to use Dataframe's User Defined Functions, you can write them in Java/Scala or Python and this will impact your computation performance – but if you manage to stay in a pure Dataframe computation – then nothing will get between you and the best computation performance you can possibly get.

4) Dataframes are the future for Spark & You

Spark ML is already a pretty obvious example of this, the Pipeline API is designed entirely around Dataframes as their sole data structure for parallel computations, model training and predictions. And even if you don't believe me, here's once again Patrick Wendell's presentation of « What the future of Spark is » :



Anyway, I think I made my point regarding the whole goal of this article : RDDs are the new bytecode of Apache Spark. You might be sad or pissed because you spent a lot of time learning how to harness Spark's RDDs and now you think Dataframes are a completely new paradigm to learn...

You're partially right because if you don't already know Pandas or R API, Dataframes are a new thing and you'll need some work to harness it – but remember that in the end, everything comes down as RDDs – so all that you learned before is still relevant, this is just another skill to add to your resume.

Vale

Tags: Apache Spark, Dataframe, Pandas, RDD