DataFrames (2)

Big Data Analysis with Scala and Spark

Heather Miller

DataFrames

So far, we got an intuition of what `DataFrames` are, how to create them, and how to do many important transformations and aggregations on them.

DataFrames

So far, we got an intuition of what `DataFrames` are, how to create them, and how to do many important transformations and aggregations on them.

In this session we'll focus on the DataFrames API. We'll dig into:

- working with missing values
- common actions on DataFrames
- joins on DataFrames
- optimizations on DataFrames

Cleaning Data with DataFrames

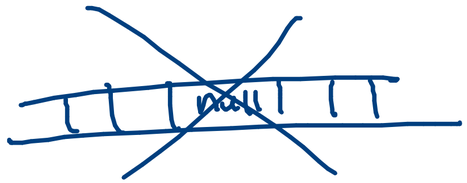
Sometimes you may have a data set with `null` or `NaN` values. In these cases it's often desirable to do one of the following:

- drop rows/records with unwanted values like `null` or "`NaN`"
- replace certain values with a constant

Cleaning Data with DataFrames

Sometimes you may have a data set with `null` or `NaN` values. In these cases it's often desirable to do one of the following:

- drop rows/records with unwanted values like `null` or "NaN"
- replace certain values with a constant



Dropping records with unwanted values:

- **`drop()`** drops rows that contain `null` or `NaN` values in **any** column and returns a new `DataFrame`.
- **`drop("all")`** drops rows that contain `null` or `NaN` values in **all** columns and returns a new `DataFrame`.
- **`drop(Array("id", "name"))`** drops rows that contain `null` or `NaN` values in the **specified** columns and returns a new `DataFrame`.

Cleaning Data with DataFrames

Sometimes you may have a data set with `null` or `NaN` values. In these cases it's often desirable to do one of the following:

- drop rows/records with unwanted values like `null` or "NaN"
- replace certain values with a constant

Replacing unwanted values:

- **`fill(0)`** replaces all occurrences of `null` or `NaN` in **numeric columns** with **specified value** and returns a new `DataFrame`.
- **`fill(Map("minBalance" -> 0))`** replaces all occurrences of `null` or `NaN` in **specified column** with **specified value** and returns a new `DataFrame`.
- **`replace(Array("id"),Map(1234 -> 8923))`** replaces **specified value** (`1234`) in **specified column** (`id`) with **specified replacement value** (`8923`) and returns a new `DataFrame`.

Common Actions on DataFrames

Like RDDs, `DataFrames` also have their own set of **actions**.

We've even used one several times already.

Common Actions on DataFrames

Like RDDs, `DataFrames` also have their own set of **actions**.
We've even used one several times already.

`collect(): Array[Row]`
Returns an array that contains all of Rows in this DataFrame.

`count(): Long`
Returns the number of rows in the DataFrame.

`first(): Row/head(): Row`
Returns the first row in the DataFrame.

`show(): Unit`
Displays the top 20 rows of DataFrame in a tabular form.

`take(n: Int): Array[Row]`
Returns the first n rows in the DataFrame.

Joins on DataFrames

Joins on DataFrames are similar to those on Pair RDDs, with the one major usage difference that, since DataFrames aren't key/value pairs, we have to specify which columns we should join on.

Several types of joins are available:
`inner`, `outer`, `left_outer`, `right_outer`, `leftsemi`.

Joins on DataFrames

Joins on DataFrames are similar to those on Pair RDDs, with the one major usage difference that, since DataFrames aren't key/value pairs, we have to specify which columns we should join on.

Several types of joins are available:

`inner`, `outer`, `left_outer`, `right_outer`, `leftsemi`.

Performing joins:

Given two DataFrames, `df1` and `df2` each with a column/attribute called `id`, we can perform an inner join as follows:

```
df1.join(df2, $"df1.id" === $"df2.id")
```

It's possible to change the join type by passing an additional string parameter to `join` specifying which type of join to perform. E.g.,

```
df1.join(df2, $"df1.id" === $"df2.id", "right_outer")
```

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

```
case class Abo(id: Int, v: (String, String))
case class Loc(id: Int, v: String)

val as = List(Abo(101, ("Ruetli", "AG")), Abo(102, ("Brelaz", "DemiTarif")),
              Abo(103, ("Gress", "DemiTarifVisa")), Abo(104, ("Schatten", "DemiTarif")))
val abosDF = sc.parallelize(as).toDF

val ls = List(Loc(101, "Bern"), Loc(101, "Thun"), Loc(102, "Lausanne"), Loc(102, "Geneve"),
              Loc(102, "Nyon"), Loc(103, "Zurich"), Loc(103, "St-Gallen"), Loc(103, "Chur"))
val locationsDF = sc.parallelize(ls).toDF
```

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

```
// abosDF:                          locationsDF:
// +---+--------------------+       +---+---------+
// | id|                   v|       | id|        v|
// +---+--------------------+       +---+---------+
// |101|         [Ruetli,AG]|       |101|     Bern|
// |102|  [Brelaz,DemiTarif]|       |101|     Thun|
// |103|[Gress,DemiTarifV...|       |102| Lausanne|
// |104|[Schatten,DemiTarif]|       |102|   Geneve|
// +---+--------------------+       |102|     Nyon|
//                                  |103|   Zurich|
//                                  |103|St-Gallen|
//                                  |103|     Chur|
//                                  +---+---------+
```

Joins on DataFrames: A Familiar Example

Example:
Recall our CFF data set from earlier in the course.

How do we combine only customers that have a subscription and where there is location info?

Joins on DataFrames: A Familiar Example

Example:
Recall our CFF data set from earlier in the course.

How do we combine only customers that have a subscription and where there is location info?

We perform an inner join, of course.

```
val abosDF = sc.parallelize(as).toDF
val locationsDF = sc.parallelize(ls).toDF

val trackedCustomersDF =
  abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))
```

Joins on DataFrames: A Familiar Example

Example:

How do we combine only customers that have a subscription and where there is location info?

```
val trackedCustomersDF =
  abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))

  // trackedCustomersDF:
  // +---+--------------------+---+---------+
  // | id|                   v| id|        v|
  // +---+--------------------+---+---------+
  // |101|         [Ruetli,AG]|101|     Bern|
  // |101|         [Ruetli,AG]|101|     Thun|
  // |103|[Gress,DemiTarifV...|103|   Zurich|
  // |103|[Gress,DemiTarifV...|103|St-Gallen|
  // |103|[Gress,DemiTarifV...|103|     Chur|
  // |102|  [Brelaz,DemiTarif]|102| Lausanne|
  // |102|  [Brelaz,DemiTarif]|102|   Geneve|
  // |102|  [Brelaz,DemiTarif]|102|     Nyon|
  // +---+--------------------+---+---------+
```

Joins on DataFrames: A Familiar Example

Example:

How do we combine only customers that have a subscription and where there is location info?

```
val trackedCustomersDF =
  abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))

  // trackedCustomersDF:
  // +---+--------------------+---+---------+
  // | id|                   v| id|        v|
  // +---+--------------------+---+---------+
  // |101|         [Ruetli,AG]|101|     Bern|
  // |101|         [Ruetli,AG]|101|     Thun|
  // |103|[Gress,DemiTarifV...|103|   Zurich|
  // |103|[Gress,DemiTarifV...|103|St-Gallen|
  // |103|[Gress,DemiTarifV...|103|     Chur|
  // |102|  [Brelaz,DemiTarif]|102| Lausanne|
  // |102|  [Brelaz,DemiTarif]|102|   Geneve|
  // |102|  [Brelaz,DemiTarif]|102|     Nyon|
  // +---+--------------------+---+---------+
```

As expected, customer 104 is missing! :-)

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

Which join do we use?

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

```
val abosWithOptionalLocationsDF
  = abosDF.join(locationsDF, abosDF("id") === locationsDF("id"), "left_outer")
//   +---+--------------------+----+---------+
//   | id|                   v|  id|        v|
//   +---+--------------------+----+---------+
//   |101|         [Ruetli,AG]| 101|     Bern|
//   |101|         [Ruetli,AG]| 101|     Thun|
//   |103|[Gress,DemiTarifV...| 103|   Zurich|
//   |103|[Gress,DemiTarifV...| 103|St-Gallen|
//   |103|[Gress,DemiTarifV...| 103|     Chur|
//   |102|  [Brelaz,DemiTarif]| 102| Lausanne|
//   |102|  [Brelaz,DemiTarif]| 102|   Geneve|
//   |102|  [Brelaz,DemiTarif]| 102|     Nyon|
//   |104|[Schatten,DemiTarif]|null|     null|
//   +---+--------------------+----+---------+
```

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

```
val abosWithOptionalLocationsDF
  = abosDF.join(locationsDF, abosDF("id") === locationsDF("id"), "left_outer")
//   +---+------------------+----+---------+
//   | id|                 v|  id|        v|
//   +---+------------------+----+---------+
//   |101|         [Ruetli,AG]| 101|     Bern|
//   |101|         [Ruetli,AG]| 101|     Thun|
//   |103|[Gress,DemiTarifV...| 103|   Zurich|
//   |103|[Gress,DemiTarifV...| 103|St-Gallen|
//   |103|[Gress,DemiTarifV...| 103|     Chur|
//   |102|  [Brelaz,DemiTarif]| 102| Lausanne|
//   |102|  [Brelaz,DemiTarif]| 102|   Geneve|
//   |102|  [Brelaz,DemiTarif]| 102|     Nyon|
//   |104|[Schatten,DemiTarif]|null|     null|
//   +---+------------------+----+---------+
```

As expected, customer 104 has returned! :-)

Revisiting Our Selecting Scholarship Recipients Example

Now that we're familiar with the `DataFrames` API, let's revisit the example that we looked at at a few sessions back.

Revisiting Our Selecting Scholarship Recipients Example

Now that we're familiar with the `DataFrames` API, let's revisit the example that we looked at at a few sessions back.

Recall Let's imagine that we are an organization, CodeAward, offering scholarships to programmers who have overcome adversity. Let's say we have the following two datasets.

```
case class Demographic(id: Int,
                       age: Int,
                       codingBootcamp: Boolean,
                       country: String,
                       gender: String,
                       isEthnicMinority: Boolean,
                       servedInMilitary: Boolean)
val demographicsDF = sc.textfile(...).toDF // DataFrame of Demographic

case class Finances(id: Int,
                    hasDebt: Boolean,
                    hasFinancialDependents: Boolean,
                    hasStudentLoans: Boolean,
                    income: Int)
val financesDF = sc.textfile(...).toDF // DataFrame of Finances
```

Revisiting Our Selecting Scholarship Recipients Example

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

Revisiting Our Selecting Scholarship Recipients Example

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- Swiss students
- who have debt & financial dependents

How might we implement this program with the DataFrame API?

```
// Remember, DataFrames available to us:
val demographicsDF = sc.textfile(...).toDF // DataFrame of Demographic
val financesDF = sc.textfile(...).toDF     // DataFrame of Finances
```

Revisiting Our Selecting Scholarship Recipients Example

With DataFrames:

```
demographicsDF.join(financesDF, demographicsDF("ID") === financesDF("ID"), "inner")
        .filter($"HasDebt" && $"HasFinancialDependents")
        .filter($"CountryLive" === "Switzerland")
        .count
```

Revisiting Our Selecting Scholarship Recipients Example

Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

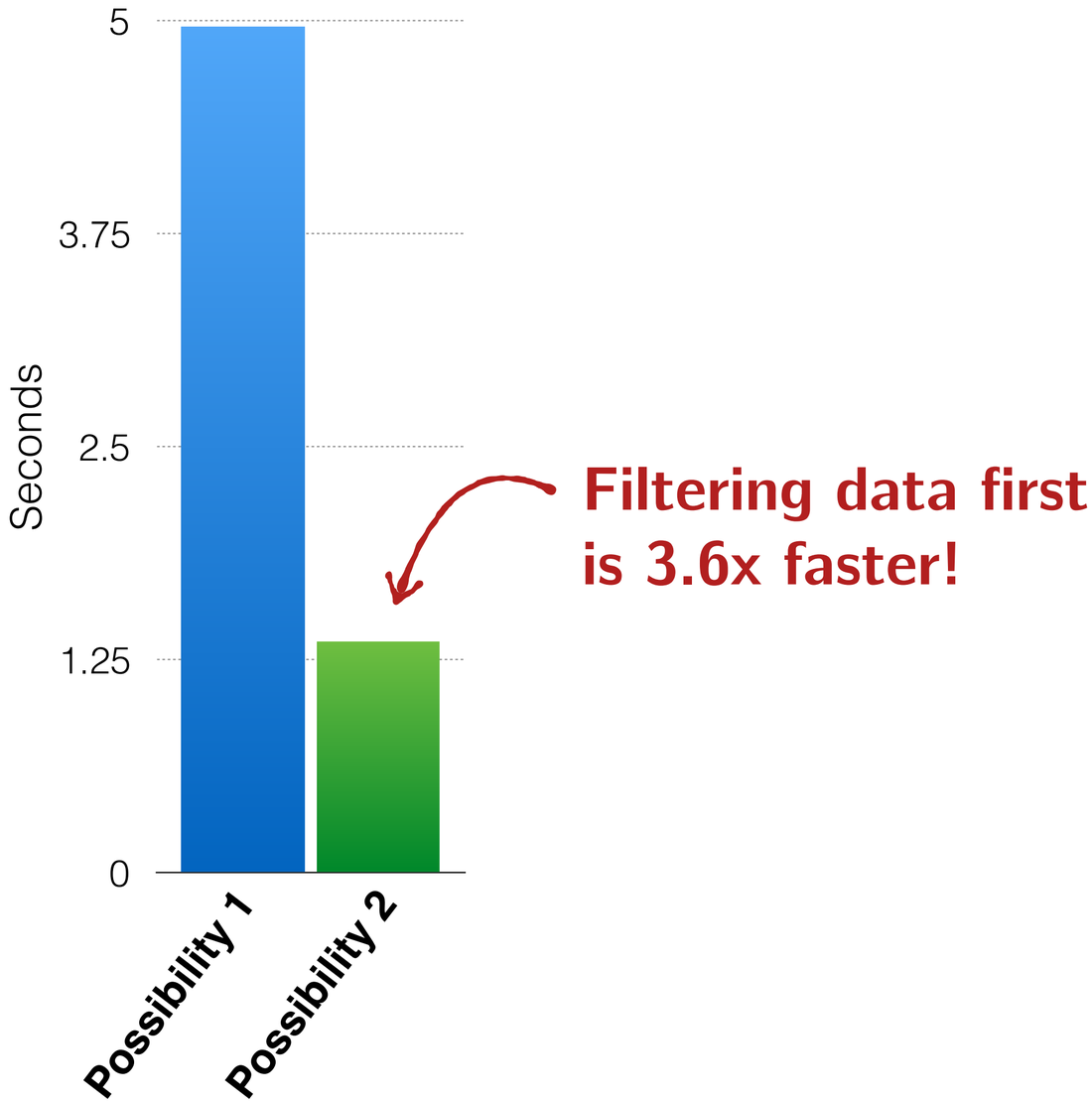
```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

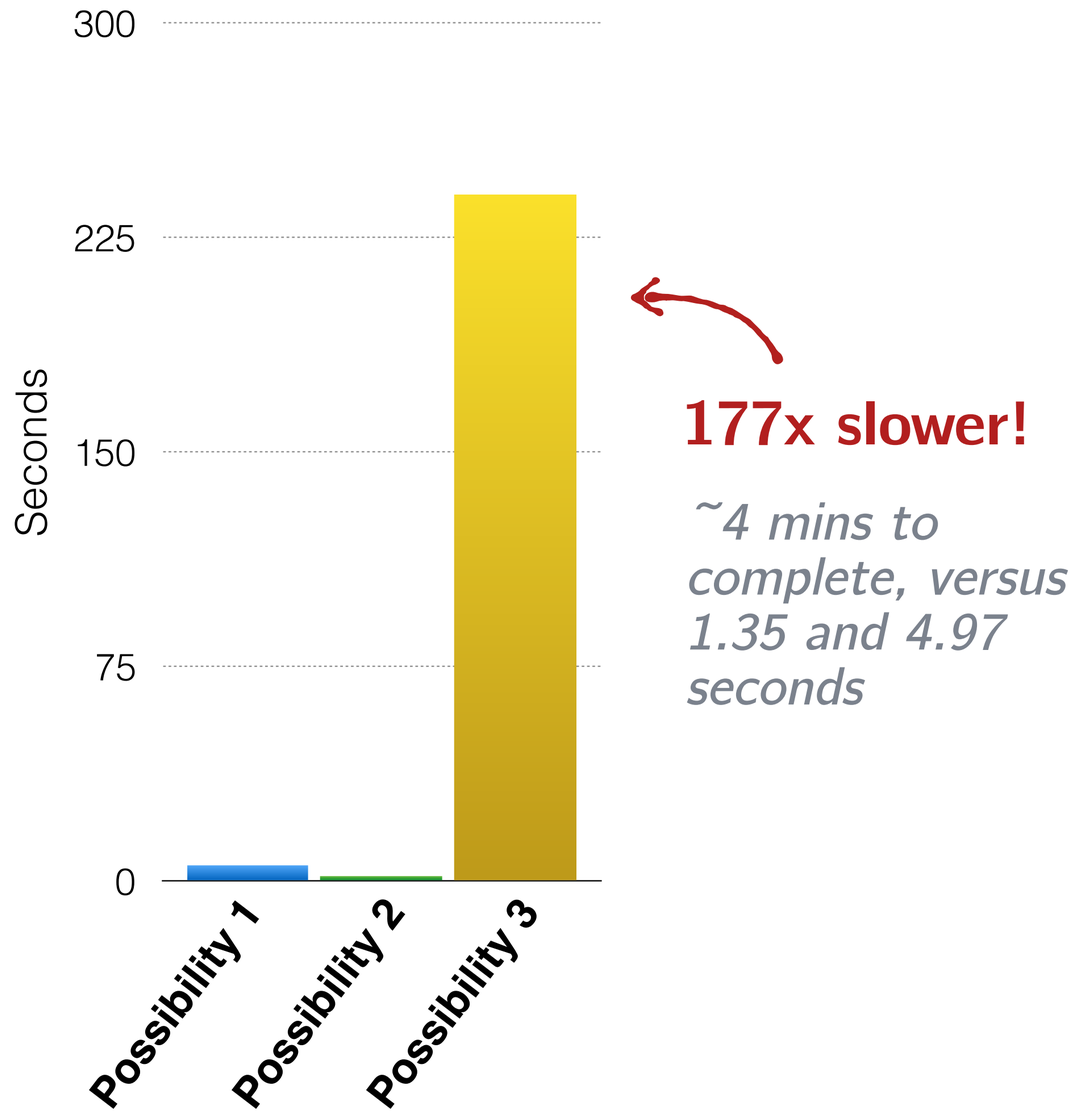
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```



Revisiting Our Selecting Scholarship Recipients Example

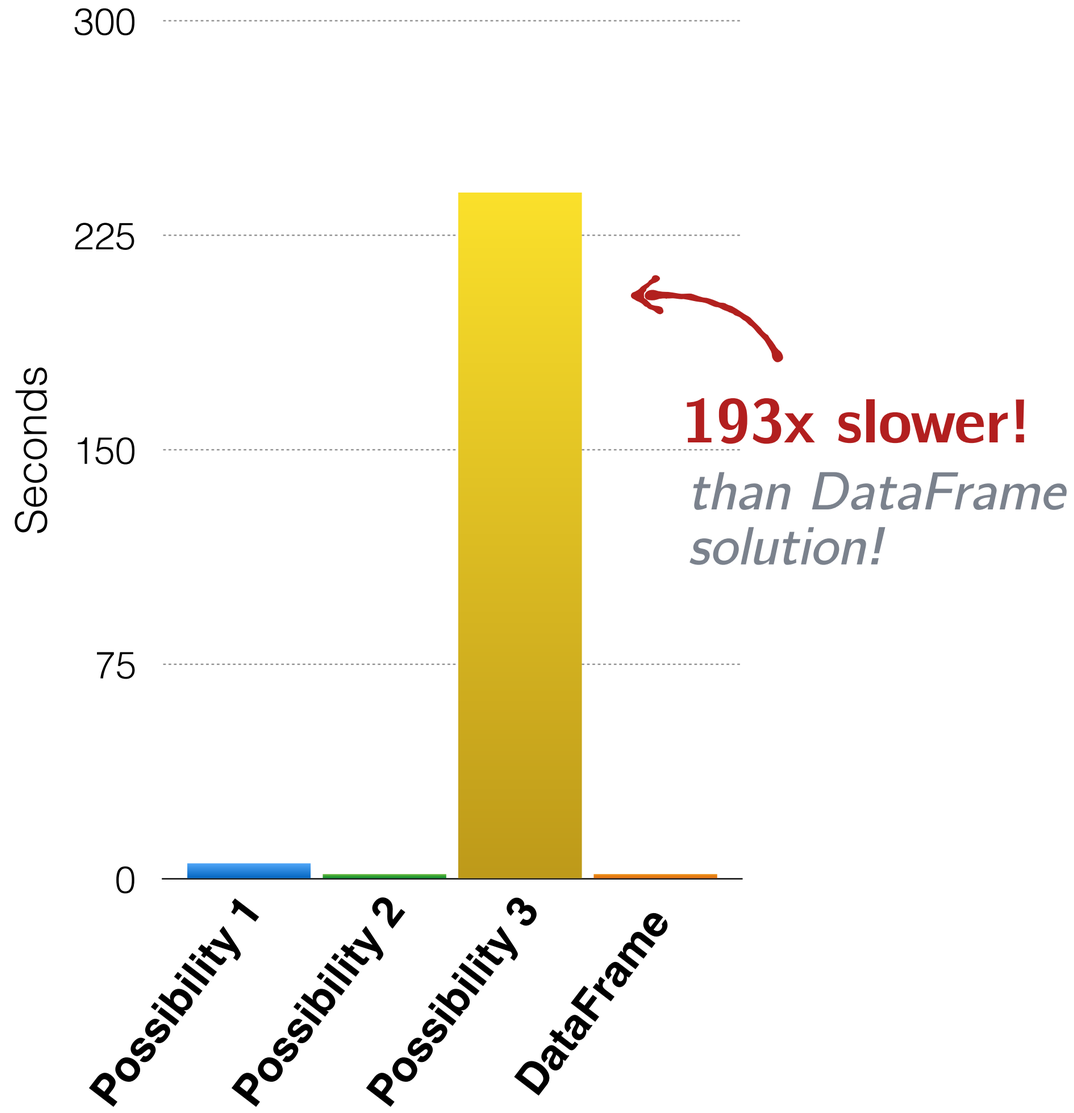
Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

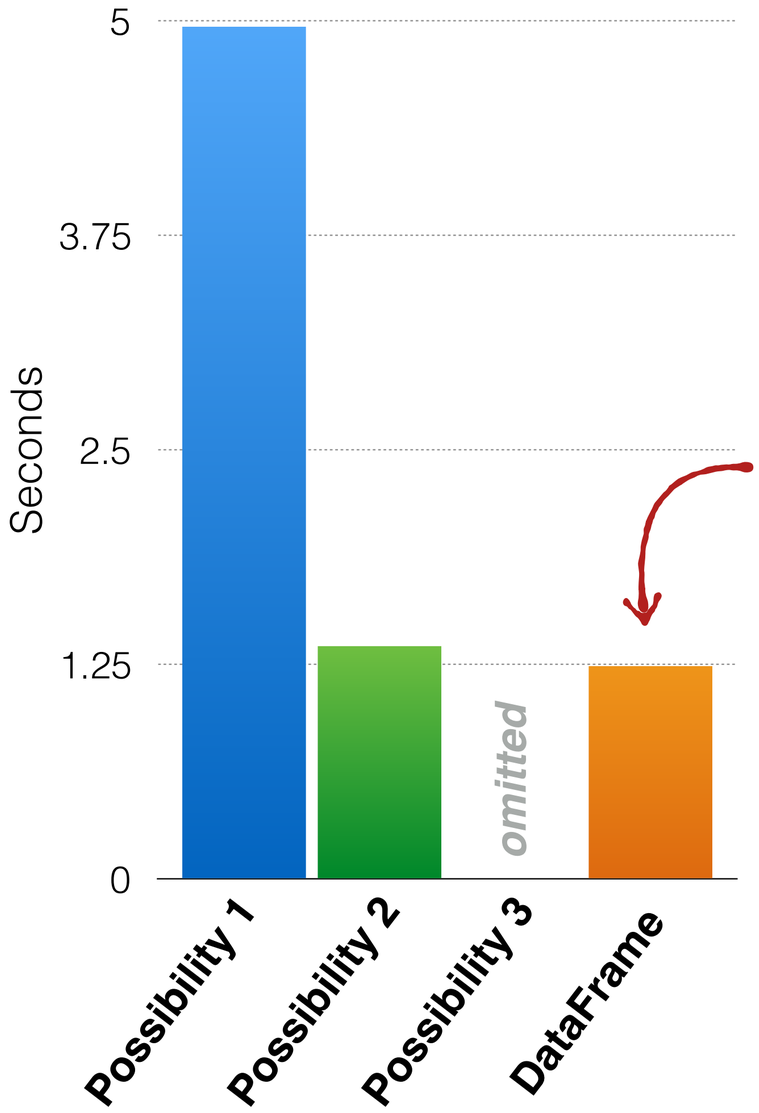
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```

DataFrame

```
> demographics.join(fi
                .filter(
                .filter(
                .count
```

▸ (2) Spark Jobs

```
res24: Long = 10
Command took 1.24 seconds
```



4x faster!
than almost same program written for RDDs

Optimizations

How is this possible?

Optimizations

How is this possible?

Recall that Spark SQL comes with two specialized backend components:

- **Catalyst**, query optimizer.
- **Tungsten**, off-heap serializer.

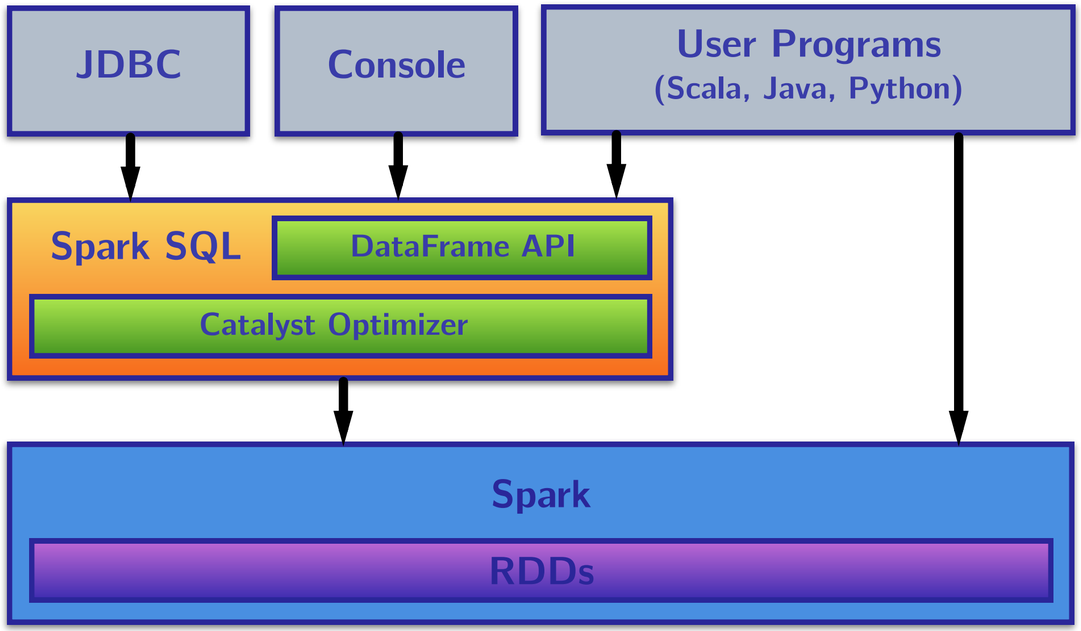
Let's briefly develop some intuition about why structured data and computations enable these two backend components to do so many optimizations for you.

Optimizations

Catalyst

Spark SQL's query optimizer.

Recall our earlier map of how Spark SQL relates to the rest of Spark:

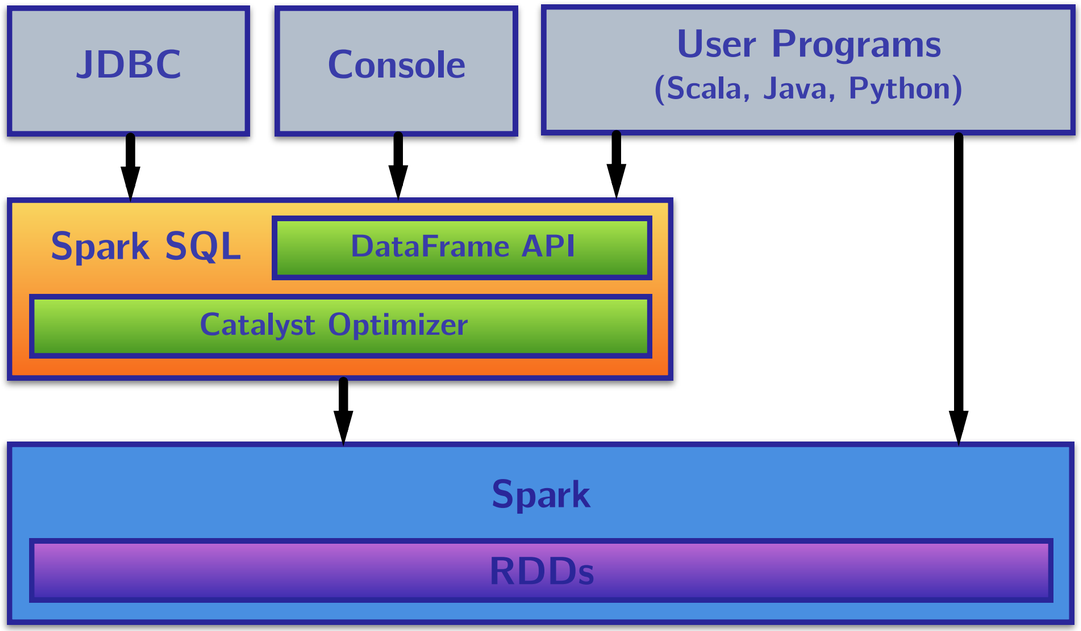


Optimizations

Catalyst

Spark SQL's query optimizer.

Recall our earlier map of how Spark SQL relates to the rest of Spark:



Key thing to remember:

Catalyst compiles Spark SQL programs down to an RDD.

Optimizations: RDDs vs DataFrames

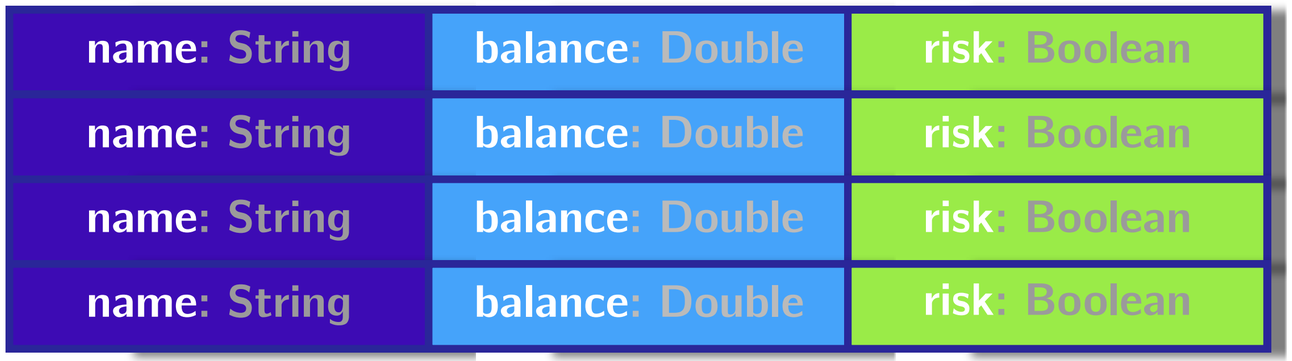
In summary:

Spark RDDs:

Not much structure.

Difficult to aggressively optimize.

DataFrames/Databases/Hive:




```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

Lots of structure.

Lots of optimization opportunities!

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- has full knowledge and understanding of all data types
- knows the exact schema of our data
- has detailed knowledge of the computations we'd like to do

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- has full knowledge and understanding of all data types
- knows the exact schema of our data
- has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- **Reordering operations.**

  Laziness + structure gives us the ability to analyze and rearrange DAG of computation/the logical operations the user would like to do, before they're executed.

E.g., Catalyst can decide to rearrange and fuse together filter operations, pushing all filters early as possible, so expensive operations later are done on less data.

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- has full knowledge and understanding of all data types
- knows the exact schema of our data
- has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- **Reordering operations.**
- **Reduce the amount of data we must read.**

  Skip reading in, serializing, and sending around parts of the data set that aren't needed for our computation.

E.g., Imagine a Scala object containing many fields unnecessary to our computation. Catalyst can narrow down and select, serialize, and send around only relevant columns of our data set.

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- has full knowledge and understanding of all data types
- knows the exact schema of our data
- has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- **Reordering operations.**
- **Reduce the amount of data we must read.**
- **Pruning unneeded partitioning.**
  Analyze DataFrame and filter operations to figure out and skip partitions that are unneeded in our computation.

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- highly-specialized data encoders
- **column-based**
- off-heap (free from garbage collection overhead!)

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- highly-specialized data encoders
- **column-based**
- off-heap (free from garbage collection overhead!)

Highly-specialized data encoders.

Tungsten can take schema information and tightly pack serialized data into memory. This means more data can fit in memory, and faster serialization/deserialization (CPU bound task)

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- highly-specialized data encoders
- **column-based**
- off-heap (free from garbage collection overhead!)

Column-based

Based on the observation that most operations done on tables tend to be focused on specific columns/attributes of the data set. Thus, when storing data, group data by column instead of row for faster lookups of data associated with specific attributes/columns.

Well-known to be more efficient across DBMS.

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- highly-specialized data encoders
- **column-based**
- off-heap (free from garbage collection overhead!)

Off-heap

Regions of memory off the heap, manually managed by Tungsten, so as to avoid garbage collection overhead and pauses.

Optimizations

Taken together, Catalyst and Tungsten offer ways to significantly speed up your code, even if you write it inefficiently initially.

Limitations of DataFrames

Limitations of DataFrames

Untyped!

```
listingsDF.filter($”state” === ”CA”)

// org.apache.spark.sql.AnalysisException:
//cannot resolve ’‘state‘’ given input columns: [street, zip, price];;
```

Your code compiles, but you get runtime exceptions when you attempt to run a query on a column that doesn't exist.

Would be nice if this was caught at compile time like we're used to in Scala!

Limitations of DataFrames

Limited Data Types

If your data can't be expressed by `case classes`/`Products` and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala `class`.

Limitations of DataFrames

Limited Data Types

If your data can't be expressed by `case classes`/`Products` and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala `class`.

Requires Semi-Structured/Structured Data

If your unstructured data cannot be reformulated to adhere to some kind of schema, it would be better to use RDDs.