Structured vs Unstructured Data

Structure and Optimization

Big Data Analysis with Scala and Spark

Heather Miller

Before we get into Spark SQL, which is the topic of this week, I first want to talk to you about structure and optimization, to sort of motivate Spark SQL.

Example: Selecting Scholarship Recipients

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 demographics = sc.textfile(...)... // Pair RDD, (id, demographic)

case class Finances(id: Int,
                    hasDebt: Boolean,
                    hasFinancialDependents: Boolean,
                    hasStudentLoans: Boolean,
                    income: Int)
val finances = sc.textfile(...)... // Pair RDD, (id, finances)
```

adversity: 逆境，苦難

So before I say anything further about what I mean by structure and optimization, let's start with an example. So let's imagine that we're an organization called Code Award, offering scholarships to programmers who have overcome some kind of adversity. Let's say we have the following two datasets, where each one of these case classes will represent some kind of information about a person. So the demographic case class, we have information like an ID number, we have a person's age, whether or not, maybe, they attended some kind of coding bootcamp. We have the country that they live in or are from. We have information like whether or not they're some kind of minority, or they have served in a

military capacity somehow. And then we have other financial information, like whether or not this person has any sort of debt, whether or not this person has financial dependents like children or anybody else that they have to financially support, or student loans, and finally, income of this person. So we have these two different datasets full of these two different pieces of information about different people. One is called demographics which is a pair RDD, where the ID is the key and the value is an instance of this demographic type. And on the other hand, the same is true for finances, where we have a pair RDD, where the key is an ID number and again, we have an instance of this finances thing as the value. Okay, so these are our two data sets.

Example: Selecting Scholarship Recipients

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 Spark program?

```
// Remember, RDDs available to us:
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)
val finances = sc.textfile(...)...     // Pair RDD, (id, finances)
```

And let's assume that our data set includes information about people from many countries, with many life and financial backgrounds. So given that, let's imagine that our goal is to count up and select students for a specific scholarship. So, of course, every scholarship has some kind of criteria, so what is our criteria? Let's say that we have, for example, grant money given to us by the Swiss government so that we want to target Swiss students. And we want to target people who are potentially in need financially. Maybe they have some kind of debt, or they have many financial dependents. So, as a first step, we want to have a look at our data set and see how many people might be eligible by this criteria. So first, let's count up the people in our dataset who are both Swiss and who have some kind of debt with also financial dependents. So assuming that we just want to count the number of people in our data set that match this criteria, how could we implement this spark program? So, again, remember that we have two paired RDDs available to us. One called demographics and one called finances. With ID numbers and instances of demographic and finances as the values, okay? So, how do we implement this Spark program? I'll give you a moment to try and come up an example solution.

Example: Selecting Scholarship Recipients

Possibility 1:

(Int, (Demographic, Finances)) — id

```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

Well, the short answer is that there are many ways to implement this program. Since we know we need to combine both the demographic and financial information, we need to do an inter-join. Because we need both demographic and financial information for a given person. The return type of this join will be another pair RDD with key integer, which represents the ID, and then a pair of demographic and finances. Once we've done that join, we can then filter out the people who meet our criteria. So in the first element that we have in the resulting pair of our join, which is of type demographic, we can check whether or not this person is from Switzerland. And then, we can check in the second part of our resulting value pair whether or not this person has financial dependents, and whether or not they have debt. If all of these conditions are met, then we can pass that through the filter, and then, finally, we can count it all up at the end.

Example: Selecting Scholarship Recipients

Possibility 1:

```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

Steps:

1. Inner join first
2. Filter to select people in Switzerland
3. Filter to select people with debt & financial dependents

So, just to summarize, in this solution we do an inner join first, then we filter to select the people in Switzerland, and then we filter to select people with debt and financial dependents. So we do an inner join first, and then we filter afterwards, and then we count it up at the end.

Example: Selecting Scholarship Recipients

Possibility 2:

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == "Switzerland")
            .join(filtered)
            .count
```

Steps:

1. Filter down the dataset first (look at only people with debt & financial dependents)
2. Filter to select people in Switzerland (look at only people in Switzerland)
3. Inner join on smaller, filtered down dataset

So another possibility is to do our filters first. We could take our two data sets, finances and demographics, and then filter them down with the conditions that we already know we have. And once we've done that filtering, we can then join the filtered results and count them up. So in the finances data set, we pass through people who have both financial dependents and debt. And in the demographics data set, we pass people who are from Switzerland. And once we've filtered those two data sets down, then we do our join and then we count it up. So just to summarize, we filter down the dataset first. We look only at people with debt and financial dependents. Then we filter to select the people in Switzerland. And finally, we do an inner join on this smaller, filtered down data set. So the assumption here is that the dataset should now be a little bit smaller when you join on a smaller dataset.

Example: Selecting Scholarship Recipients

Possibility 3:

```
val cartesian                    finances
  = demographics.cartesian(demographics)

cartesian.filter {
  case (p1, p2) => p1._1 == p2._1
}
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
                   (p2._2.hasFinancialDependents) &&
                   (p2._2.hasDebt)
}.count
```

Steps:

1. Cartesian product on both datasets
2. Filter to select resulting of cartesian with same IDs
3. Filter to select people in Switzerland who have debt and financial dependents

And yet another possibility is to start with a Cartesian product between demographics and, oops, this should be finances. This is finances, here. So we start with the Cartesian product and then we pass through our filter pairs that have the same ids. And then we finally, we filter out the people from who are from Switzerland and people who have both financial dependents and some kind of debt as well, and then we count it up at the end. So this is yet another possibility that would give us the same answer, so again, this is finances, there's a typo here. So again, to go through the steps, we start by computing the Cartesian product in both data sets. Then we filter to select the results of the Cartesian that had the same IDs, so this is equivalent to an inner join. And finally, we filter out the people from Switzerland who have debt and financial dependents.

Example: Selecting Scholarship Recipients

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.

150,000 people

Possibility 1

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

▸ (1) Spark Jobs

res0: Long = 10

Command took 4.97 seconds -

Possibility 2

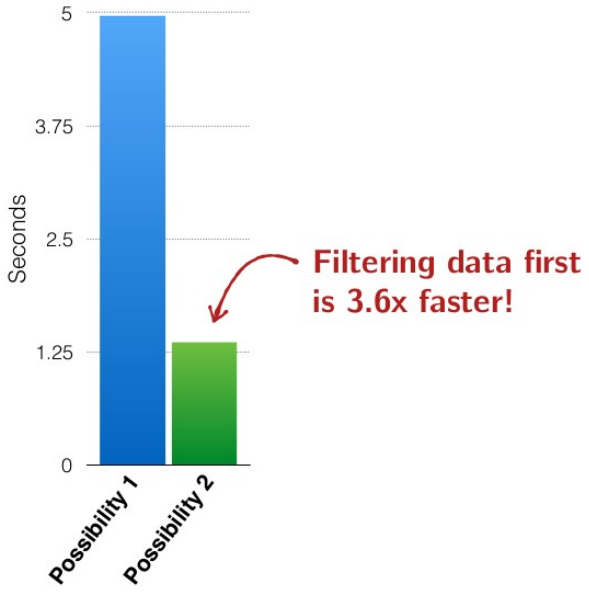
```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

fsi: org.apache.spark.

res4: Long = 10

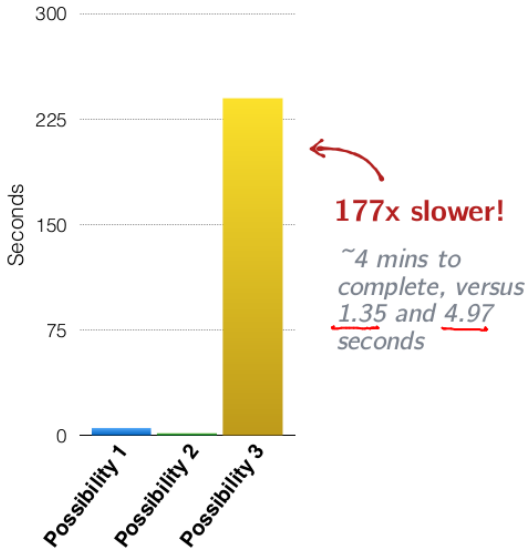
Command took 1.35 seconds



So the bottom line here is that, for all three of these examples, the end result is always going to be the same. However, the big difference is the time that it takes to compute the job. So while the resulting value might be the same, the execution time is vastly different. Here are the running times of the first two possibilities, on a data set of roughly 150,000 people. In the first possible solution, where we did the join first followed by the filters afterwards, this computation took 4.97 seconds, so this blue bar here. Whereas in the second possible solution, we filtered down the data set first before computing the join. It only took 1.35 seconds, so that's here. So the bottom line is that the running times of these two different possible solutions is pretty different. Filtering the data set first results in a run time that's 3.6 times faster than if we did the join first and the filter second.

Example: Selecting Scholarship Recipients

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.



But that doesn't even compare to the possible solution where we computed the Cartesian product first. In this case, it took approximately 4 minutes to complete our computation versus 1.35 seconds, or 4.97 seconds for the other two possible solutions. So this is 177 times slower than if we both did an inner join instead of a Cartesian product. This is kind of common sense, but the bottom line here is to show that you can get the same result while having vastly different running times.

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

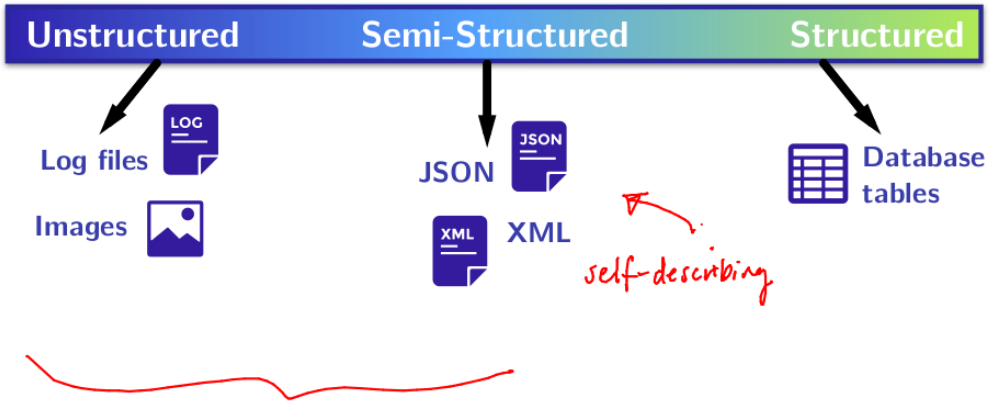
Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

Given a bit of extra structural information, Spark can do many optimizations *for* you!

Since a recurring theme of the course so far has been that we have to carefully think about what our Spark jobs are doing, I would hope that you wouldn't think to use a Cartesian product first. However, it is conceivable that you might have found yourself in a situation where you lost a factor 3 or 4x of performance by choosing the join first before the filter. Many people would think to join first and then filter down second. Whereas, upon retrospect, it does seem more sensible to filter first and then join afterwards. Wouldn't it be nice, though, if Spark automatically knew, if we wrote the code in possibility 3, which was the Cartesian product code, or even if we wrote the code in possibility 1, that it could rewrite our code to be possibility number 2. So, Spark would just automatically know that it would be smarter to do a filter before the join, rather than the filter after the join. It would be great if Spark was smart enough to do that. What I hope to convey over the next few sessions is that given a bit of extra structural information, Spark is actually able to do many of these optimizations for you, and that all comes in the form of Spark SQL. So the key here is that structural information will enable optimizations. And in order to get a little bit of intuition about how that might be the case, we've got to start by looking at what I mean by structure.

Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.



So to start, all data is not created equal in a structural sense. Data falls on some kind of spectrum between unstructured and structured. But what does structured mean? Well, structured is something like a database table, like SQL or something that is very rigid with a fixed schema. On the other far end of the scale is unstructured data. So things like dumps of log files, or just unstructured text data of some kind, or even images. Images are another form of unstructured data. In the middle, we have semi-structured data, so things like JSON or XML. These are data sets that carry along some kind of schema. And they're, so-called, self-describing. So they describe their own structure. However, their schemas are maybe not as rigid as those schemas in a database table. So, as you can see, there's a little bit of a spectrum here, from unstructured to very structured. And so far, with RDDs, we've kind of focus on this side of of the spectrum (紅括號裡的). We've been typically reading in data from logs, or data from JSON. Somehow we've been manually parsing the JSON ourselves into case-class objects. And then we've been doing some kind of large-scale computation on all of that once we've parted into spark somehow.

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the ***schema*** of the data it's dealing with.

Given an arbitrary RDD, Spark knows that the RDD is parameterized with arbitrary types such as,

- `Person`
- `Account`
- `Demographic`

but it doesn't know anything about these types' structure.

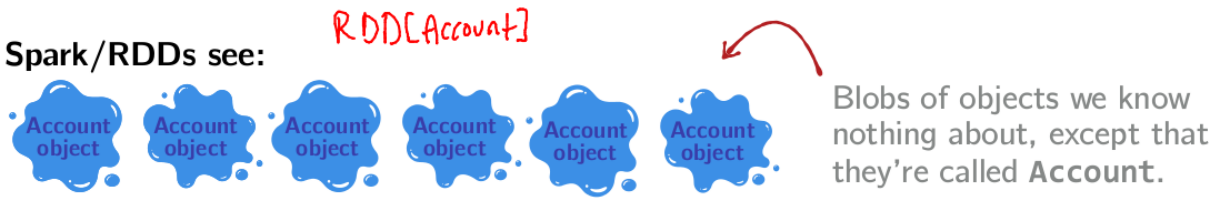
But how does all that relate back to Spark? Well, so far, Spark and the regular RDDs we've been looking at have no concept about the schema of the data that it's dealing with. So has no idea where the structure is of the data that it's operating on. All Spark knows, given some RDD, is that that RDD is parametrized with some kind of arbitrary type, like person, account, or demographic. But it doesn't actually know anything about these type structures. It just has a name. It has a name. It knows it should have a person inside of it. But it doesn't know anything about what that person is made up of.

Structured Data vs RDDs

Assuming we have a dataset of **Account** objects:

```
case class Account(name: String, balance: Double, risk: Boolean)
```

Spark/RDDs see:



Blobs of objects we know nothing about, except that they're called **Account**.

Spark can't see inside this object or analyze how it may be used, and to optimize based on that usage. It's opaque.

Perhaps a little more visually, assuming that we have a case-class account with these fields, name, which is a type string. Balance, a type double, and risk, a type Boolean. With an RDD of type Account, all Spark knows about Account is that it has these blobs of objects called Account. We know nothing about them other than their names. Spark can't look inside of these Account objects to look at their structure and analyze what parts of their structure might be used in the subsequent computation. For example, it's conceivable that this account object might be bigger than these three fields here, it could have hundreds of fields. And perhaps, a certain computation only needs one of those fields. But Spark, as is, will serialize each one of these really big account objects and send them all over the network, even though it's not necessary to use most of the data that it's sending around. Spark can't do optimizations like these, because it can't see inside of these account objects. And it can't optimize based on their structure.

Structured Data vs RDDs

Assuming we have a dataset of **`Account`** objects:

```
case class Account(name: String, balance: Double, risk: Boolean)
```

Spark/RDDs see:

A database/Hive sees:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

Columns of named and typed values.

If Spark could see data this way, it could break up and only select the datatypes it needs to send around the cluster.

On the other hand, in a structured data set, in some kind of database table, for example, computations are done on columns of named and typed values. So absolutely everything about the structure of a data set is known in a structured setting like in a database table or in Hive. So absolutely everything about the structure of a data set is known in a database table. And in fact, databases tend to be heavily optimized. So given all of this structure, optimizations like I had mentioned with this account object can be done, because we know the structure of the data set that we're operating on, and we know which parts of it we are going to use, and we can optimize based on that information.

Structured vs Unstructured Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.

Like the data Spark operates on, function literals too are completely opaque to Spark.

A user can do anything inside of one of these, and all Spark can see is something like: **`$anon$1@604f1a67`**

So far we've been talking about structured versus unstructured data, but the same could also be said for

computation. So there is also structured and unstructured computation. So far, in Spark, what we've been doing is, we've been doing functional transformations on data. So we pass some kind of user-defined function literal to a high order function like map, flatMap, or filter. So we're passing, essentially some kind of lambda function, some sort of function or passing it around. This, too is completely opaque. Just like in the previous example, Spark only knows that this is called some funny name like $anon$1@604f1a67. So, in this case, this is also completely unstructured and opaque. Spark can't look at the operations that we're trying to do and then make some optimizations based on those optimizations. Again, we have some kind of opaque blob that Spark can't optimize on.

Structured vs Unstructured Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.

In a database/Hive:

- We do **declarative transformations** on data.
- Specialized/structured, pre-defined operations.

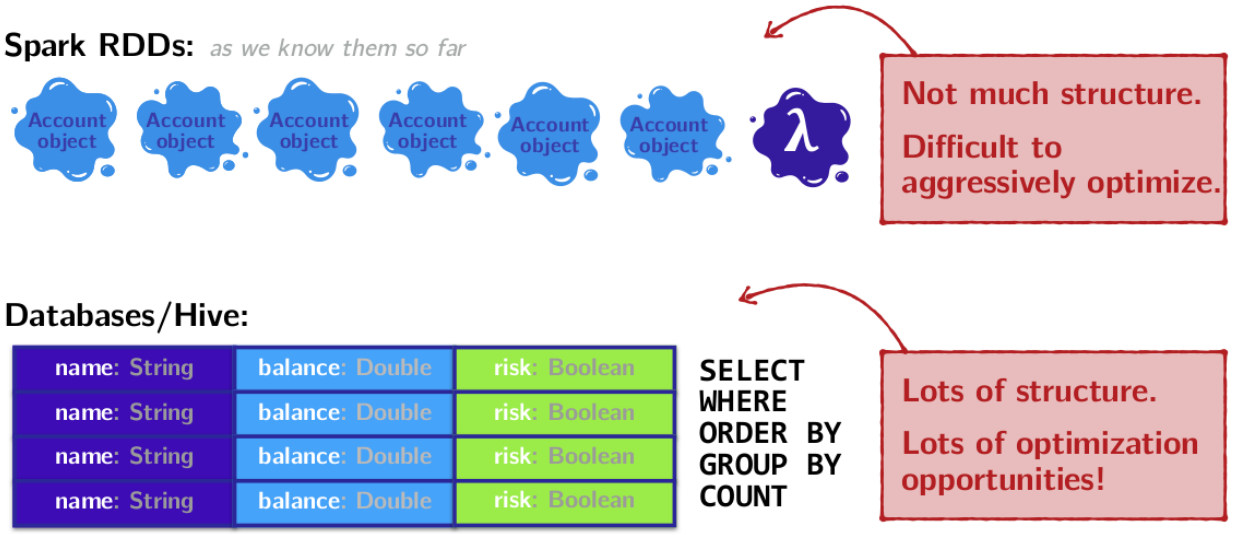
Fixed set of operations, fixed set of types they operate on.

Optimizations the norm!

Whereas in databases, typically we do declarative transformations on the structured data that we have in this data set. And all of these operations tend to be very specialized, very structured, very fixed, very rigid, predefined operations. So we know all of the possible operations that somebody could potentially do. And we can do lots of optimizations based on knowing all of the possibilities for these operations.

Structured vs Unstructured

In summary:



So just to put these two things next to each other, in summary what we have when we look at Spark RDDs, is basically, a bunch of these unstructured objects that we don't know much about, and some kind of functionality that we also don't know anything about, so just some kind of lambda function that we can't look into and make optimizations based off of. And on the other hand, we have databases, where we have this very structured data set; everything is some kind of specified data type. It's organized in rows, in columns, in this very rigid structure. And there are these very fixed set of operations that we can do on this structured data. So, on the one hand we have not much structure here and it's difficult to aggressively optimize with so little structure. And on the other hand, we have lots of structure here, and optimization opportunities abound, which gives us all kinds of opportunities to do optimizations like reordering operations, for example, and putting filters before joins.

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

Spark SQL makes this possible!

We've got to give up some of the freedom, flexibility, and generality of the functional collections API in order to give Spark more opportunities to optimize though.

So that brings me back to Optimizations in Spark. How does all of this talk about structured data and structured computations come back to Spark? Because, as we've seen, RDDs essentially operate on unstructured data, we have to parse it ourselves. And there are a few limits on the computations that we can do. Our computations already find us functions that we pass to high order functions that we've, you know, written ourselves. On our own data types, we defined our data types as well. As we painstakingly learned over the past two weeks, we have to think a lot about what's happening on the cluster and we have to do optimizations ourself. We have to think about how we can optimize our computations. Whereas in the databases world, optimizations are done automatically for you. Wouldn't it be nice if Spark could do some of these optimizations as well for us? Well, that's the whole point of Spark SQL. Spark makes these optimizations possible. The one caveat is we're going to have to give up some of the freedom, flexibility and generality that we've learned to love in this functional collections API in order to give Spark some structure and thus more opportunities to optimize.

Spark SQL

Spark SQL

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we are diving to a very important components of Spark, called spark SQL.

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- if it were possible to seamlessly intermix SQL queries with Scala?
- to get all of the optimizations we're used to in the databases community on Spark jobs?

Spark SQL delivers both!

Despite Spark's rising popularity, SQL has been and still is the the lingua franca for doing analytics. Over despite how widespread SQL is used, it's still a pain in the neck to connect big data processing pipelines like spark or Hadoop to an SQL database. Usually for in application, one has to choose either SQL or something like Spark or Hadoop. Wouldn't it be nice that were possible to seamlessly intermix SQL queries with regular Scala code? And even better than that, wouldn't it be nice to get all off the optimizations that we're used to in the databases community on Spark jobs as well? So it's pretty common in databases to do all kinds of very advanced optimizations. But so far we have to optimize our computer pipelines in Spark by hand. Spark SQL is a component that delivers both of these two nice things. It makes it possible to seamlessly intermix SQL and Scala, and it also optimizes Spark SQL code very aggressively kind of like using many the same techniques from the databases world.

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.
2. High performance, achieved by using techniques from research in databases.
3. Easily support new data sources such as semi-structured data and external databases.

So Spark SQL has three main goals, the first of which is to support relational processing in Spark. So this that goal of intermixing relational code, declarative SQL like syntax with functionally APIs like Sparks RDD API. And this is because it's sometimes more desirable to express a computation in SQL syntax than with functional APIs and vice a versa. So, you could imagine that, for example, selecting an entire column of the database is easier to do with a single line in a SQL statement than it is to do somehow with Mac or a filter operation. The other two main goals of Spark SQL is of course, high performance like previously mentioned and support for getting data more easily into Spark. Wouldn't it be nice if we could just read in semi-structured data like JSON, for example? So Spark SQL seeks to add relational processing to Spark, bring super high performance from optimizations in the databases world, and to support reading in data from semi-structured and structured datasets.

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Three main APIs:

- **SQL literal syntax**
- **DataFrames**
- **Datasets**

Two specialized backend components:

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

That said, what is Spark SQL? Is it like another system that we have to learn how to use? No, it's just a component of the Spark stack. So, you can think of it as a library. It's a Spark module for structured data processing or sort of doing relational queries and it's implemented as a library on top of the Spark. So you can think of it as just adding new APIs to the APIs that you already know. And you don't have to learn a new system or anything. And the three main APIs that it adds is SQL literal syntax, and a thing called DataFrames and another thing called datasets. So these are the APIs, these are the things

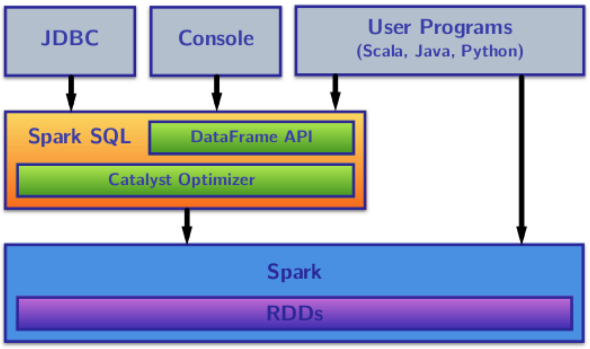
that you see in and have access too, and on the backend it adds two new backend components. It adds something called Catalyst, which is a query optimizer. This is the component that will optimize your code. And another component called tungsten which is an off-heap serializer will set another way to system that encodes Scala objects in a very, very efficient representation off-heap away from the garbage collector. So this is the main things that Sparks SQL adds, APIs and some stuff from the backend that should speed up your code quite a bit.

Spark SQL

Spark SQL is a component of the Spark stack.

- It is a Spark module for structured data processing.
- It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:



Just to give you a visual depiction of hows Spark relates to the rest of the whole Spark ecosystem, so we have this system called Spark and inside of it we have this RDD API, we've been operating this entire course on these things called RDDs. We've been doing functional transformations on them, they're sort of the core data structure of Spark. Spark SQL sits on top of Spark as a library. It has its own APIs, and it has this thing called the Catalyst Optimizer, and while it has it's own DataFrame API, it ultimately runs on RDDs under the covers. And on top of Spark SQL, you have everything else, like the Console, and regular User Programs, so user can write programs both using the DataFrame API and RDDs, and freely intermix them. But the bottom line here is that, we have this API that then does some optimizations and ultimately runs RDDs under the covers.

Relational Queries (SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- There are a set of fixed data types. `Int`, `Long`, `String`, etc.
- There are fixed set of operations. `SELECT`, `WHERE`, `GROUP BY`, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Let's quickly establish a common set of vocabulary and a baseline understanding of SQL.

So Spark SQL is a Spark component that provides a SQL-like API on top of Spark. This is neat because everything about SQL is structured. In fact, SQL stands for Structural Query Language. It has a fixed set of data types, so things like longs, integers, strings, arrays even, and has a fixed set of operations, SELECT, WHERE, GROUP, BY, etc. And for decades, both research and industry surrounding these relational databases have focused on exploiting this rigidness. This known data types, this known operations to get all kinds of performance speedups. Since knowledge of SQL databases wasn't required pre-requisite for this course, I'm just going to quickly spend a slide running through vocabulary.

Relational Queries (SQL)

Data organized into one or more **tables**

- Tables contain ***columns*** and ***rows***.
- Tables typically represent a collection of objects of a certain type, such as **`customers`** or **`products`**

A *relation* is just a table.

Attributes are columns.

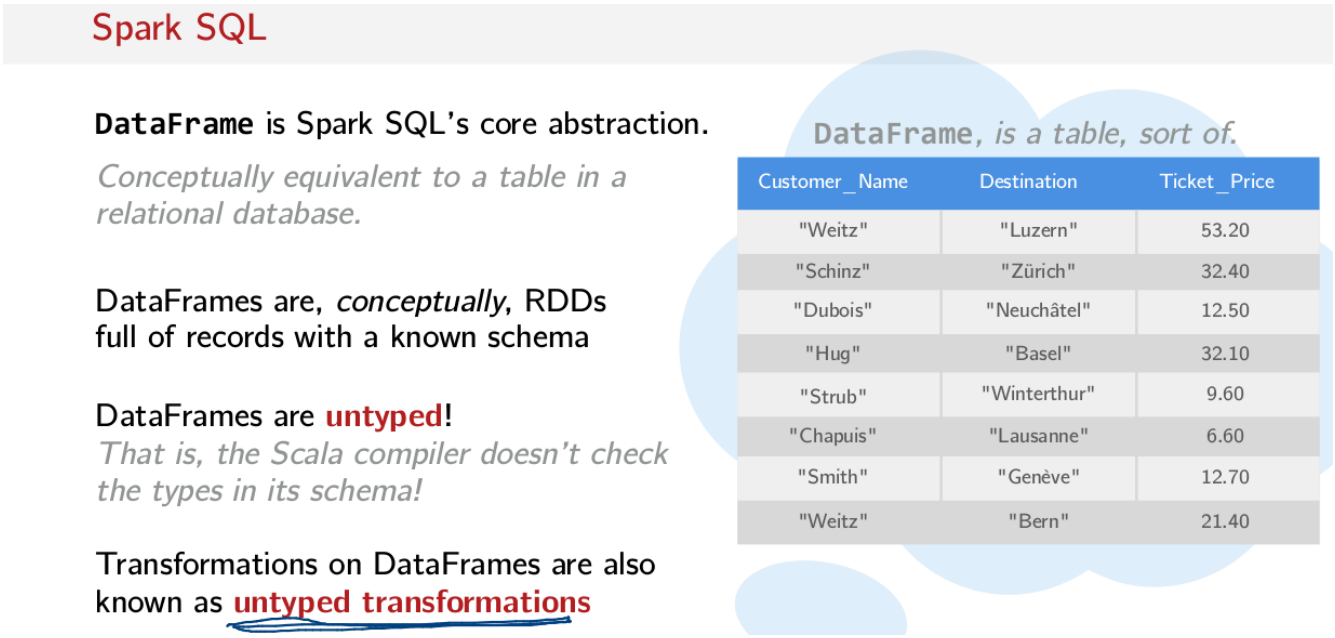
Rows are *records* or *tuples*

record /tuple

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

When thinking in SQL, data's organized into one or more tables. So here's the example of what a table might look like, there's a header, tables of columns which are named, for example, Customer_Name, or Destination, or Ticket_Price. And then there are rows, so records. And a table typically represents some kind of collection or dataset. So, you might have a table and think of it as a table of customers. So we can think of this as the CFF or SBB customer data set. So like the train dataset that we were looking at in an earlier session. This is how you think about a table. It's kind of like a collection of data. Other words that you may here used. A relation is just a table, and attributes are columns. So this

Customer_Name here is an attribute. So this is just a column. Rows are also called either records or tuples. So this row here represents a record or a tuple. This record represents, for example, a ticket purchase, this is the last name of the customer, the destination of the customer and the price that the customer paid for their ticket, this is one record in our dataset representing that purchase. So these is the terminology that we are going to use. We have tables also known as relations, we have columns in our tables which are also known as attributes. And then we have rows which are also known as records or tuples.



Now back to spark SQL. Spark SQL's core abstraction is known as a DataFrame. And every time we use the word DataFrame, you can imagine it as being in conceptually equivalent to a table in a relational database. So DataFrames are conceptually distributed collections of records. And these records, they have a known schema, okay?. And super important is this point here, that unlike RDDs which are also a kind of distributed collection, DataFrames require some kind of schema information about the data that it contains. DataFrames are also untyped, okay? This is an important point that's going to come back. What do I mean by untyped? Well I mean, that it doesn't have some kind of type parameter that the Scala compiler statically checks, compare that for a moment with RDDs. RDDs have a type parameter (RDD[T]). So we can always make sure that the type that's inside an RDD makes sense. However, DataFrames have no type parameter. They can contain rows which can contain any schema which the Scala compiler does not have a lot of ability to type check. And finally, one last terminological point. Transformations on DataFrames are called untyped transformations. This terminology's going to also come back to us later.

SparkSession

To get started using Spark SQL, everything starts with the `SparkSession`

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("My App")
  //.config("spark.some.config.option", "some-value")
  .getOrCreate()
```

Real quick before I go any further, we gotta talk about the SparkSession object. So, if you remember this Spark context from the rest of this course, the SparkSession is basically the Spark context for everything Spark SQL. So you're going to now have to switch to using the SparkSession instead of Spark context. And to create and use one of these SparkSession objects is still pretty straightforward. You just import the SparkSession object and there's this builder pattern here. You can add custom configurations that you'd like, but in general, you have to start with a builder. You have to name your app, and then you have to say .getOrCreate() so you can have any number of configuration options in the middle here. But this is the general shape, this is kind of the minimum that you need to start a new spark session. Another important thing to notice this SparkSession object I believe is meant to supersede SparkContext down the line, so this is something that you should start looking into.

Creating DataFrames

`DataFrames` can be created in two ways:

1. From an existing RDD.
   Either with schema inference, or with an explicit schema.
2. Reading in a specific **data source** from file.
   Common structured or semi-structured formats such as JSON.

So let's get into, how do we create a DataFrame? There are two ways to create a DataFrame. Either from an existing RDD, or by reading in some kind of specific data source from file. So, reading in some kind of common structured or semi-structured format like JSON. And for creating a DataFrame from an existing RDD, we can do it via two ways. Either by inferring the schema, or by explicitly providing the schema that should be contained inside of our DataFrame.

Creating DataFrames

(1a) Create DataFrame from RDD, schema reflectively inferred

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[(Int, String String, String)]
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF *without arguments, Spark will assign numbers as attributes (column names) to your* DataFrame. _1 _2 _3

So the first way to create a DataFrame that I'm going to cover is to create one from an existing RDD by reflectively inferring this schema, okay? This is probably the most convenient way to create DataFrames. So assuming that you have some kind of RDD,let's just say, in this case, it's a tulpeRDD. To turn our tulpeRDD into a DataFrame, all we have to do is call ToDF on it. We can pass arguments to toDF which represent the names of the columns. If we don't pass this list of column names to toDF when we call it then Spark will assign numbers as attributes or column names to your DataFrame will be usually something like this, _1 or _2 or _3 for the number of the column.

If you already have an RDD containing some kind of case class instance, then Spark can infer the attributes from the case class's fields.

```
case class Person(id: Int, name: String, city: String)
val peopleRDD = ... // Assume RDD[Person]
val peopleDF = peopleRDD.toDF
```

PVE 代碼中好像就是按 case class 那樣弄的.

However, if you have an RDD of case class, things are even more convenient. So assuming we have this case class Person, and we have an RDD of Person, all we have to do is call toDF on this RDD of Person objects. And in this case, the column names automatically get inferred to be the names of the fields of these case class. And again, this is done only of every reflection.

Creating DataFrames

(1b) Create `DataFrame` from existing RDD, schema explicitly specified

Sometimes it's not possible to create a `DataFrame` with a pre-determined case class as its schema. For these cases, it's possible to explicitly specify a schema.

It takes three steps:

- Create an RDD of `Rows` from the original RDD.
- Create the schema represented by a `StructType` matching the structure of `Rows` in the RDD created in Step 1.
- Apply the schema to the RDD of `Rows` via `createDataFrame` method provided by `SparkSession`.

Given:

```
case class Person(name: String, age: Int)
val peopleRdd = sc.textFile(...) // Assume RDD[Person]
```

The other way of creating DataFrame from an existing RDD which is quite a bit involved is to explicitly specify a schema. And this is useful for the case where you don't have some kind of predefined case class that you can use as its schema. Like I said, this is a bit involved, so bear with me here. It takes three steps. First you create an RDD of rows from the original RDD. Then you create a schema separately that matches the structure of rows. Then finally, you apply that schema to your RDD of rows using this method called createDataFrame. So, let's step through a quick example. And like I said, this is a little bit involved, so please bear with me. For the sake of this example, I'm just going to stick with a case class even though I know it's possible to use the two DF method. Just stick with me here. So assuming I have an RDD full of these Person objects.

Creating DataFrames

(1b) Create `DataFrame` from existing RDD, schema explicitly specified

```
// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

I first have to do encode the schema in the string, so I have to do this manually. Then I have to build up a schema using these StructType things and StructField things that matches the shape of the data that I'm trying to build a schema for. So this is done by breaking up this string and then mapping on it and creating a StructField for each part of the schema which has the name in the string here as the parameter. It has a type called StringType which we'll see later. And nullable is set to true. Like I said, we'll see some of these things later. And then all of that goes inside of something called StructType and now we have a schema. So this is what a schema looks like. We build up the schema from these things called StructTypes, StructFields. Then we convert our RDD of people to a rowRDD. Then we convert our peopleRDD into a rowRDD by breaking it up and mapping everything to Row objects. And finally, we can use this createDataFrame method on the SparkSession object and pass to it the rowRDD that we just created, as well as the schema that we just created. So that's how you manually, explicitly build up a schema and then create a DataFrame from that schema. It's a bit involved but sometimes necessary.

Creating DataFrames

(2) Create `DataFrame` by reading in a data source from file.

Using the `SparkSession` object, you can read in semi-structured/structured data by using the `read` method. For example, to read in data and infer a schema from a JSON file:

```
// ‘spark‘ is the SparkSession object we created a few slides back
val df = spark.read.json(”examples/src/main/resources/people.json”)
```

Semi-structured/Structured data sources Spark SQL can directly create DataFrames from:

- JSON
- CSV
- Parquet
- JDBC

To see a list of all available methods for directly reading in semi-structured/structured data, see the latest API docs for `DataFrameReader`:
`http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader`

And finally, likely the most convenient way to create a DataFrame is by reading in data from a source file. So using the same SparkSession object, you can read in a number of semi-structured or structured data types using the read method. So, there's this read method on SparkSession object and then you can read in, you can specify what you'd like to read in, for example, JSON, and then specify a path to this JSON file. It's kind of like read text file but it's better. It all partially semi-structured and structured data and it will build up a DataFrame with that schema of that semi-structured or structured datas. And of course, it's important to know that there are a handful of formats that Spark SQL can just automatically read in for you, can't read arbitrarily anything but the most common are probably going to be JSON, CSV, and Parquet. There are more methods available for directly reading in semi-structured and structured data. To see that entire list you can visit the docs which are here in the slides.

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement

// SQL literals can be passed to Spark SQL's sql method
val adultsDF
  = spark.sql("SELECT * FROM people WHERE age > 17")
```

Now that you know how to make DataFrames, once you have a DataFrame, you can freely write familiar SQL syntax to operate on your DataFrame. So how do we use plain old SQL syntax on a DataFrame? Assuming that we have a DataFrame called peopleDF, all we have to do is register our DataFrame as a temporary SQL view. So there's a method called createOrReplaceTemView here, but you can call on your DataFrame. This gives you a name that you can refer to in your SQL FROM statement. So in order to use SQL syntax, you need to register your DataFrame as a table. And then, you can operate on it and regular SQL syntax. So, now that I had this temporary SQL view register called people, I can freely use it inside of a SQL query. So, to do a SQL query in Spark, once you have a DataFrame called the SQL method on the SparkSession object and pass to it a SQL query which refers to a registered temporary SQL view. So in this case, people. So in this crew, we select all rows from this people table where the age field is greater than seventeen, for example. It's quite convenient, isn't it?

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

- SELECT
- FROM
- WHERE
- COUNT
- HAVING
- GROUP BY
- ORDER BY
- SORT BY
- DISTINCT
- JOIN
- (LEFT|RIGHT|FULL) OUTER JOIN
- Subqueries: SELECT col FROM (SELECT a + b AS col from t1) t2

Supported Spark SQL syntax:
https://docs.datastax.com/en/datastax_enterprise/4.6/datastax_enterprise/spark/sparkSqlSupportedSyntax.html

For a HiveQL cheatsheet:
https://hortonworks.com/blog/hive-cheat-sheet-for-sql-users/

For an updated list of supported Hive features in Spark SQL, the official Spark SQL docs enumerate:
https://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features

Now, since you know you're writing SQL queries on top of Spark which is not a SQL database, you might ask what SQL statements you have available to you. Well, the SQL statements that are available to you are largely what's available in the Hive Query Language. So, this includes standard SQL statements such as SELECT, FROM, WHERE, COUNT, HAVING, GROUP BY, ORDER BY, SORT BY, and of course all of the JOINS. And also importantly, you can do subqueries as well in this syntax. Most everything from HiveQL is supported so you can usually just try it out and see if it works. But if you want to collect more information about what syntax may or may not be supported, I've included the number of links for you here in the slides. There's a cheat sheet for HiveQL, a list of supported Sparks SQL syntax and an updated list of supported high features in Spark SQL on this Sparks official documentation.

A More Interesting SQL Query

Let's assume we have a `DataFrame` representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF
```

Let's query this data set to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

What would this SQL query look like?

So let's doing more interesting SQL query because so far we've just created DataFrames and we've done really simple query. Let's now assume we have a DataFrame representing a data set of employees. So, how do we create that data set of employees? Well, we have a case class Employee here with a number of fields, id which has type Integer, first name, last name, both type String, age type Integer and then the city that this person lives in, type String here. And to create a DataFrame, we simply create an RDD and then call toDF on it. So now we have a DataFrame for these employee records here. Now it's time for a little quiz. Let's query this data set to obtain just the IDs and the last names of the employees working in a specific city. Let's say Sydney, Australia and then, once we've obtained that, let's sort our result in order of increasing employee ID. So that means, we want the IDs and the last name's column of this employee DataFrame for only employees that work in a specific city. And then after we're done with that, we want to sort our employees in order of increasing employee ID. So I leave it to you, how would you write this SQL query?

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                   FROM employees
                  WHERE city = "Sydney"
               ORDER BY id""")
```

registered "employees"

Well it's not too difficult. Let's assume that we have a table registered With the name employees. I look at that on the last slide, but assuming we had this table registered, all you have to do is say SELECT id and last name. So we select these two columns from this employees' table. And, we filter out everybody who lives in Sydney and then, we just ORDER BY the id. To make sure we understand this query, let's visualize it.

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF

// employeeDF:
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

"employees"

So assuming that this is our dataset here, so we have this employee DataFrame. So assuming we have this dataset here, this is our employee DataFrame. And remember, we register it as a table called employees.

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                 FROM employees
                WHERE city = "Sydney"
             ORDER BY id""")
```

Result ↓

```
// employeeDF:                          sydneyEmployeesDF:
// +---+-----+-------+---+--------+     +---+-------+
// | id|fname|  lname|age|    city|     | id|  lname|
// +---+-----+-------+---+--------+     +---+-------+
// | 12|  Joe|  Smith| 38|New York|     |221| Walker|
// |563|Sally|  Owens| 48|New York|     |645|Markham|
// |645|Slate|Markham| 28|  Sydney|     +---+-------+
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

Note: it's best to use Spark 2.1+ with Scala 2.11+ for doing SQL queries with Spark SQL.

This is what the result of the query looks like. We've obtained our result in dataset by selecting only these two columns. And then filtering out the records where city equals Sydney. So we get rid of these two people that live in New York here. And then we sort the elements by order of increasing id numbers. So in this dataset, the ids are backwards and here they're now in the right order. So the bottom line, now we know how to create DataFrame and how to do arbitrary SQL queries on top of them, pretty cool.

DataFrames (1)

DataFrames (1)

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we're going to dig deeper into the DataFrame API.

DataFrames

So far, we got an intuition of what `DataFrames` are, and we learned how to create them. We also saw that if we have a `DataFrame`, we use SQL syntax and do SQL queries on them.

DataFrames have their own APIs as well!

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

- available DataFrame data types
- some basic operations on DataFrames
- aggregations on DataFrames

So far, we got a little bit of an intuition about what DataFrames are, how they fit into this whole Spark SQL thing, and we saw how to create them. We saw that if we have a DataFrame, we can use SQL syntax and do SQL queries on them, which is pretty cool. But it doesn't just end there. DataFrames have their own APIs as well in addition to being able to use just arbitrary SQL syntax on a DataFrame. In this session we'll focus on the DataFrames API. In particular, we'll dig into the available DataFrame data types, we'll look at some of the basic operations that are available on DataFrames that you should know about, because it's not like RDDs. And finally we'll look at how to do aggregations on DataFrames, because there's also a little bit of a different way to do that as well.

DataFrames: In a Nutshell

`DataFrames` are...

A relational API over Spark's RDDs
Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized
Spark SQL applies years of research on relational optimizations in the databases community to Spark.

Untyped!
The elements within DataFrames are `Rows`, which are not parameterized by a type. Therefore, the Scala compiler cannot type check Spark SQL schemas in DataFrames.

Okay, DataFrames in a nutshell, let's recap some of the key points to remember when using DataFrames. So the first point is that DataFrames can be thought of as just a relational API over Spark's RDDs. So you're running ultimately on top of RDDs, the same RDDs we learned about in previous sessions, but we have a nice relational API over them now. And the reason for this is because it's just sometimes more convenient to use a declarative relational API, than it is to use functional APIs for

analysis jobs. We saw this in the last example, in the last session, where we saw this nice little query, where we were trying to select some information about some employees, and then filter it and sort it. And it was done in almost no code at all and it was extremely clear to read. It would have been a little bit more involved and a little bit less clear to read if we had done it with a functional API. The next key point to hold onto about DataFrames is that they're able to be aggressively optimized automatically. So Spark SQL takes many years of research in the database communities. When we apply them in Spark, now that we have some computations that are built up in sort of a relational way, we can apply these relational optimizations to get the same optimizations that people in the databases community have been enjoying for many years. And finally, one thing that you're really going to have to remember, maybe it won't bite us right in the beginning of this session, but it definitely will later, is that DataFrames are untyped. So the elements within DataFrames are rows and they're not parameterized by some types. So the Scala compiler can't type check Spark SQL schemas in DataFrames. They're just opaque to the Scala compiler. There's all kinds of analysis done by Spark SQL, but the Scala compiler doesn't do any type checking. So they're effectively untyped.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Basic Spark SQL Data Types:

Scala Type	*SQL Type*	*Details*
`Byte`	`ByteType`	1 byte signed integers (-128,127)
`Short`	`ShortType`	2 byte signed integers (-32768,32767)
`Int`	`IntegerType`	4 byte signed integers (-2147483648,2147483647)
`Long`	`LongType`	8 byte signed integers
`java.math.BigDecimal`	`DecimalType`	Arbitrary precision signed decimals
`Float`	`FloatType`	4 byte floating point number
`Double`	`DoubleType`	8 byte floating point number
`Array[Byte]`	`BinaryType`	Byte sequence values
`Boolean`	`BooleanType`	true/false
`Boolean`	`BooleanType`	true/false
`java.sql.Timestamp`	`TimestampType`	Date containing year, month, day, hour, minute, and second.
`java.sql.Date`	`DateType`	Date containing year, month, day.
`String`	`StringType`	Character string values (stored as UTF8)

So as I've mentioned a few times now, Spark SQL's DataFrames operate on a restricted set of data types. This is in order to enable optimization opportunities, so we always know exactly what the data looks likes. So here is a list of the basic data types in Spark SQL. So in this column we have the Scala types, so all of the types that we're used to already seeing in this course, and these are the types that SQL sees. And this is a description of each of the types. These are all pretty standard Scala types, or pretty standard primitive types. We have things like byte, shorts, ints, longs, we have strings here at the bottom. Interestingly enough, there is a timestamps and dates from Java. We've things like floats and doubles, BigDecimal, these are all pretty uncontroversial data types. But these are the data types that Spark SQL uses. These are the core data types that Spark SQL is built around.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Arrays

Array of only one type of element (`elementType`). `containsNull` is set to true if the elements in `ArrayType` value can have null values.

Example:

```
// Scala type     // SQL type
Array[String]     ArrayType(StringType, true)
```

In addition to these basic data types, Spark SQL has a handful of complex data types available to use, such as arrays, maps, and well, in this case, case classes, but structs. So these are complex data types that can contain basic data types or other complex data types themselves, so you can build up richer data types using these things, these arrays, maps, or in this case, structs. Since each of these data types are represented a little bit differently in Spark SQL, because of their history in regular SQL, I'm going to run through each of them to show you sort of the main differences. And really, the main difference has to do with this containsNull thing here. So, in this case an array, for example an array of string, in SQL is an array type with a string type, which is its parameter with this boolean here. And in this case this boolean is this containsNull thing, which is set to true if the elements in the ArrayType can have no values. So if it's possible for this array to contain nulls, we're going to have this conditional set to true, otherwise it'll be set to false. So there's always this additional piece of data hanging around here called containsNull. So we have not just type information about the type contained in the array and the fact that it's an array at all, but we also have some knowledge about whether or not this thing can contain nulls.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Maps

Map of key/value pairs with two types of elements. `valuecontainsNull` *is set to true if the elements in* `MapType` *value can have null values.*

Example:

```
// Scala type          // SQL type
Map[Int,String]        MapType(IntegerType, StringType, true)
```

The next complex data type that we're going to look at is a map. Just like maps in Scala, maps have keys and values. And the SQL version of this has types that correspond to the keys and types that correspond to the values. So a map type contains a keyType and a valueType, as well as this Boolean here, which represents whether or not the valueType can contain null, okay? So it's similar to what we just saw in arrays, but instead this containsNull thing refers to the value instead. So whatever is in this value, so in this case a string, if this value can be null, then this is set to true. So we know the type, this is nullable basically.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's `DataFrames` operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	*SQL Type*
`Array[T]`	`ArrayType(elementType, containsNull)`
`Map[K, V]`	`MapType(keyType, valueType, valueContainsNull)`
`case class`	`StructType(List[StructFields])`

Structs

Struct type with list of possible fields of different types. `containsNull` *is set to true if the elements in* `StructFields` *can have null values.*

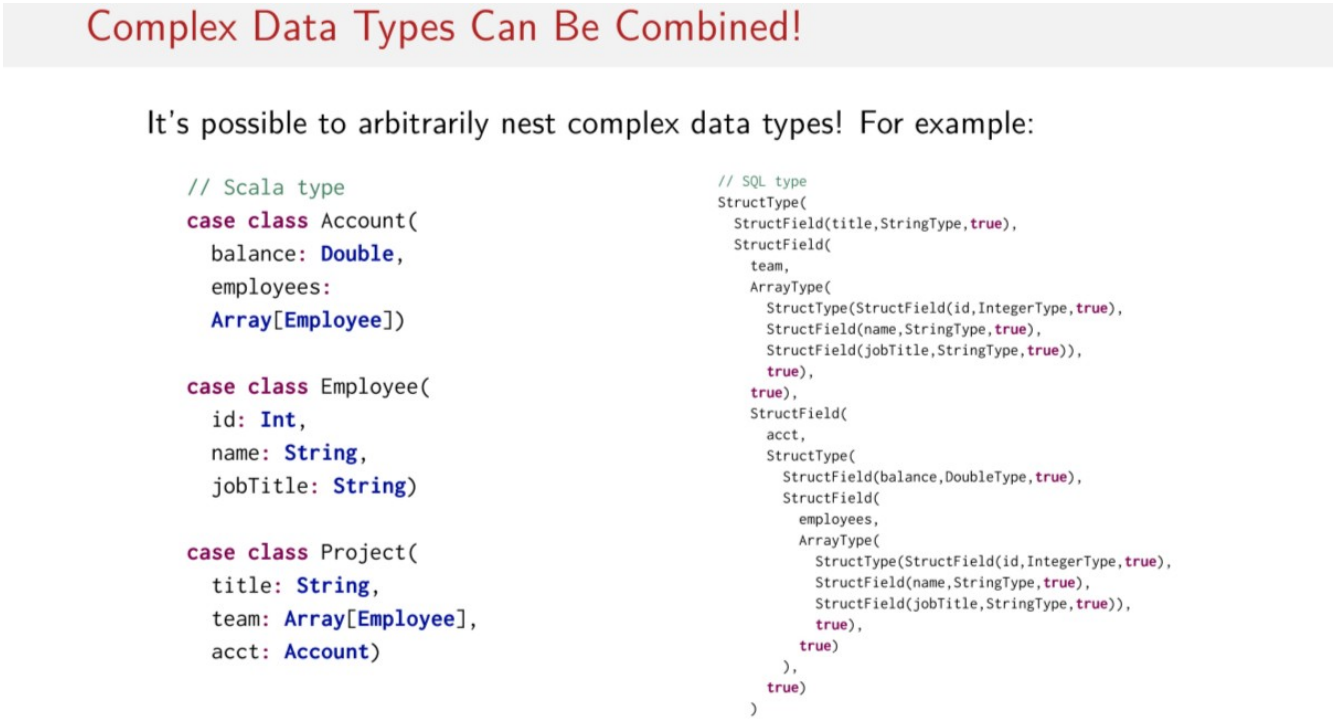
Example:

```
// Scala type
case class Person(name: String, age: Int)

// SQL type
StructType(List(StructField("name", StringType, true),
                StructField("age", IntegerType, true)))
```

Case class 原來就相當於 C 語言等中的 struct 哦.

And finally Struct. So Structs are little bit less rigid than arrays and maps. The Struct type contains a list of possible fields of different types. So unlike arrays and maps, which were one type or two possible types, structs can be any arbitrary number of types. But you have to have something called a struct field, which contains a name for the field, the type of the field, and then again, this nullable value here. So you can build up more interesting data types this way. And this is how you can take, for example, something like a case class and map it to something that SQL can understand. So, in this case we'll be using case classes in Scala but in SQL the same case class would look like this. So if we have a case class person with a field name and a field age, the name is type String and the age is type Int. This can be realized as a StructType in SQL with a list of fields. One field, so one Struct field, has the name name. It has the type String, and it's nullable, it's possible to be nulled. And again, another field which has name age as type integer and also is nullable. So this is how we can represent, for example, a person case class object here. So these are the data types that Spark SQL can operate on. This is pretty important because if for some reason your data type doesn't fit into one of these shapes, it can't be worked on in Spark SQL. So just to repeat, Spark SQL has a restricted set of possible types. Nothing fancy happening in these types, you have basic structs, basic arrays, basic maps. And then you have this list of effectively primitive types that I showed you on the previous slide. And every computation that we will be doing in Spark SQL has to be able to be represented by these sets of types.



Just to show you that it's possible to arbitrarily nest these complex data types, here's an example where I've taken some Scala types here. So there are actually rather complex and nested, so I have a type called Account, which contains a field called balance, and a field called employees, which is an array of Employees. But Employee is defined as another case class with these fields here, with primitive fields, Int, String, String, and finally there's another case class called Project. With a type String as a title, and then another array of these Employee objects and then a value of type Account here. So this is a rather complex data type this Project thing here. So if I wanted to represent this Project type, I could do it just like this within SQL. So it's super involved and difficult to read. But this is actually just a realization of

this Project type here, which contains Accounts and Employees types. So you can basically write this code and then expand it to be this, right? So this here is project, right? And I'm sorry about the wonky, True is here something happened in the PDF readar. But, these are all struct fields and these true things are the nullable Booleans. Things like arrays, you have things like again more Structtypes, within Structtypes. And then you have, of course, primitive type so here is a StructField of type StringType with the name name, for example. So again, these things can be nested and built up to represent reasonably complex data types.

Accessing Spark SQL Types

Important.
In order to access *any* of these data types, either basic or complex, you must first import Spark SQL types!

```
import org.apache.spark.sql.types._
```

One important note. So, this is actually going to come up again and again throughout Spark SQL. Spark SQL really requires a lot of imports. Throughout this and the next few sessions, I've tried to tell you everywhere where we might have to have some special imports to use one of the things that I'm showing you. Though if ever in doubt, have look at the Spark docs. In this case, in order to use any of the data types, you need to import all the Spark SQL's types so that can be done like this. So again, if you're ever going to use Spark SQL data type, you actually need to explicitly import the types that Spark SQL operates on.

DataFrames Operations Are More Structured!

When introduced, the DataFrames API introduced a number of relational operations.
The main difference between the RDD API and the DataFrames API was that DataFrame APIs accept Spark SQL expressions, instead of arbitrary user-defined function literals like we were used to on RDDs. This allows the optimizer to understand what the computation represents, and for example with filter, it can often be used to skip reading unnecessary records.

DataFrames API: Similar-looking to SQL. Example methods include:

- `select`
- `where`
- `limit`
- `orderBy`
- `groupBy`
- `join`

Okay, so what sorts of operations are defined on DataFrames? So far all I told you is that, the DataFrames API is all about relational operations, relational API. So I said simply, the DataFrames API actually is quite similar looking to SQL, example method includes select, where, limit, orderBy,

groupBy, join. You might notice that this is quite different looking from RDDs that you're already used to because we've been operating on things like maps and filter and with reduces and stuff like this, we've been passing functions to higher order functions like map, produce and filter, right? So the main difference between the RDD API and the DataFrames API was that, when introduced the DataFrames API was designed to accept just the Spark SQL expressions instead of arbitrary user-defined function literals, like we used all over the place when we were learning about RDDs. And the reason for this was to give the optimizer a restricted set of operations that it already knew how to optimize based on, so these relational optimizations that needed relational operations do those relational optimizations. Right, so here's a selection of some of the more often we use methods, but we'll go into more detail in subsequent slides.

Getting a look at your data

Before we get into transformations and actions on `DataFrames`, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.show()
// +---+-----+-------+---+--------+
// | id|fname|  lname|age|    city|
// +---+-----+-------+---+--------+
// | 12|  Joe|  Smith| 38|New York|
// |563|Sally|  Owens| 48|New York|
// |645|Slate|Markham| 28|  Sydney|
// |221|David| Walker| 21|  Sydney|
// +---+-----+-------+---+--------+
```

But first before I do that I gotta show you one important trick so you can actually see what you're doing with these relational operations. So, to see what your data looks like. So there are a couple of methods in Spark to show what your data looks like. The first that I'm going to be using all the time is called show. It pretty prints the DataFrame in a tabular form. And it shows only the first 20 elements. So it looks something like this. So if I have this employee DataFrame again, right, which we use toDF on, and if I do show on it, it prints out this nice thing here which shows all of the elements, in this case there's only four. Otherwise, it would show up to 20. But it print these out nicely. So you can visually sort of parse this easier.

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

printSchema() prints the schema of your DataFrame in a tree format.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.printschema()

// root
//  |-- id: integer (nullable = true)
//  |-- fname: string (nullable = true)
//  |-- lname: string (nullable = true)
//  |-- age: integer (nullable = true)
//  |-- city: string (nullable = true)
```

And then another operation that we can use to get a look at our data is called printSchema which prints the schema of the DataFrame in tree format, so here's an example of what this looks like. I'm using the same exact DataFrame here. So this employee DataFrame that we saw earlier and so here's what it looks like when we say printschema. So, this is actually a quite simple schema it basically, this is the root and there's just one level of fields here. I have just five fields, one of type integer string, string, integer string, right, and all of them are nullable. Remember this nullable thing that we saw in previous slides? There's always this nullable Boolean that has to hang around to say whether or not one of these data types can be nulled out for some reason.

Common DataFrame Transformations

Like on RDDs, transformations on DataFrames are (1) operations which return a DataFrame as a result, and (2) are lazily evaluated.

Some common transformations include:

```
def select(col: String, cols: String*): DataFrame
// selects a set of named columns and returns a new DataFrame with these
// columns as a result.

def agg(expr: Column, exprs: Column*): DataFrame
// performs aggregations on a series of columns and returns a new DataFrame
// with the calculated output.

def groupBy(col1: String, cols: String*): DataFrame // simplified
// groups the DataFrame using the specified columns. Intended to be used before an aggregation.

def join(right: DataFrame): DataFrame // simplified
// inner join with another DataFrame
```

Other transformations include: filter, limit, orderBy, where, as, sort, union, drop, amongst others.

由後面知, agg 就是指 sum, max, min, avg 這些東西

Okay, so let's actually look at some of these transformations. Two things to remember, we're talking about transformations right now which means that transformations always return a data as a result and second, don't forget that they're lazily evaluated, okay, just like regular RDDs, these are lazily evaluated too. Some of the most common transformations include, select, which I'm sure you already have kind of a idea of what it might do. It selects a set of name columns, so more than one column potentially and it returns and you DataFrame with these columns as a result. We saw this earlier when we said, we wanted to select two out of the, i don't know, five or several columns in the SQL query that we did in the previous session where we took the employee's last name and employee's ID number out of our DataFrame and returned a new DataFrame with those columns. So just visually, if you have If you have a table with four columns and then you say, I want to select columns A and D then you get back a new DataFrame or new table I guess with just these two columns. And you select by passing the names of the columns to the select method.They are here other old versions of this. You can have a look at this part to see the different choices for selecting columns but this is ultimately the idea here. The next transformation that's going to come up is called aggregate. So, this one is peculiar and we're going to go into more detail in to it later in this session. But, the short story is that aggregate performs aggregations on a series of columns, so on one or more columns and it returns a new DataFrame with the calculated output with the aggregated output. Okay, so we'll get more concrete about what that means a few slides from now, but moving on group by, which you can guess what that might do. So it groups the DataFrame using the specified columns. So, it's important to note that this is a little bit of weird method here. I say that it returns a DataFrame, but it doesn't exactly return a DataFrame. Because this groupBy method is actually intended to be used right before an aggregation. So typically, you would be doing a groupBy and then an aggregation after the groupBy. So we'll go into a little more depth when we talk about aggregations about what this groupBy exactly does but you can think about it right now as grouping the DataFrame by some specified columns, okay? And then, we'll do aggregation on it in the second step. And finally, one of the more commonly used operations is, of course, a join. There are many ways to interact with joins. We'll go into detail in the next session about all of the ways to use joins. Just remember that, all of these things are transformations. Either they explicitly return a DataFrame or they indirectly return a DataFrame as a result. Of course, there are several more transformations such filter, limit, orderBy, where, as, sort, union, and drop. These are all basically the same operations that we know from SQL. All of these along with their overloads can be found in the Spark SQL docs.

Specifying Columns

As you might have observed from the previous slide, most methods take a parameter of type `Column` or `String`, always referring to some attribute/column in the data set.

Most methods on `DataFrames` tend to some well-understood, pre-defined operation on a column of the data set

You can select and work with columns in three ways:

1. **Using $-notation**

```
// $-notation requires: import spark.implicits._
df.filter($"age" > 18)
```

2. **Referring to the Dataframe**

```
df.filter(df("age") > 18))
```

sql(" ... ")

3. **Using SQL query string**

```
df.filter("age > 18")
```

So I'd like to show you an example of how to use some of these transformations. But before I do, I have to talk a little bit about columns. So if we go back to the previous slide, we'll note that sometimes you pass Strings to this operations and sometimes you pass a Column. And, usually the Strings are meant to refer to a Column but what's a Column? So how do I actually talk about this columns in the context of this Spark SQL programs? So there's a few different ways to specify columns and it could sometimes be confusing if you just jump in to a code so that anybody telling you what this things mean. So the main three ways that you'll see people referring to specific columns in the DataFrame are as follows. Most commonly you'll see this $-notation which is just a short-hand for specifying a name of a specific columns. So, if your calling for example filter on some DataFrames, so I'll go DataFrame .filter, if I want to filter on the age column, I can do $ and then in quotations put the name of the column and then I can do whatever operation, whatever SQL operation I want to do. So, this basically means the age column of this df, dataframe here. So, another way to refer to a specific column is to actually explicitly refer to the DataFrames, so we can, just like above, we can explicitly say, okay, the df DataFrame here and the the age column we pass a string to it. So, we pass age as a string and then that's a signal to Spark SQL to look in this DataFrame for that column. So and finally, the last way to do it is using a SQL query string. So just like what we saw when we were actually writing out these SQL queries in this syntax like this, where the SQL query was a string here, Spark SQL actually parses that with a SQL parser and then figures out what are columns and what are operators from this string that you pass to it. So you can do the same thing here where you can pass sort of a SQL query kind of and hope that Spark SQL can parse it correctly. There's sometimes some errors with this. So I would, if I were you I would probably go with one of these two methods (前兩個) over this method (第三個), because these are a little bit less error prone.

DataFrame Transformations: Example

Example:
Recall the example SQL query that we did in the previous session on a data set of employees. Rather than using SQL syntax, let's convert our example to use the DataFrame API.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
```

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

How could we solve this with the DataFrame API?

Okay, so now that we know how to refer to columns inside of these DataFrame operations, let's look at an example. Recall the example SQL query that we did in the previous session on this data set of employees. So we've been looking at this employee DataFrame now for a little while. Rather than using SQL syntax like we did in the last session, let's convert our example to use the DataFrame API. So if you remember what this example asks of us, it asked us to obtain just the IDs and the usernames of the employees working in a specific city. Say in this case, you want to select just the people living in Sydney, okay? And then let's sort the result in order of increasing employee ID afterwards. So once we've grabbed these columns and then filtered out the people who live in Sydney, then we sort them by the employee ID. So how do we solve this with the DataFrame API with some of the methods that you've seen so far? I'll let you try it first.

DataFrame Transformations: Example

Example:
We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
                                  .where("city == 'Sydney'")
                                  .orderBy("id")
// employeeDF:
// +---+-----+-------+---+--------+     sydneyEmployeesDF:
// | id|fname|  lname|age|    city|     +---+-------+
// +---+-----+-------+---+--------+     | id|  lname|
// | 12|  Joe|  Smith| 38|New York|     +---+-------+
// |563|Sally|  Owens| 48|New York|     |221| Walker|
// |645|Slate|Markham| 28|  Sydney|     |645|Markham|
// |221|David| Walker| 21|  Sydney|     +---+-------+
// +---+-----+-------+---+--------+
```

The DataFrame solution actually looks a lot like the SQL query that we did in the previous session. In the previous session, we used this, a select statement followed by a where statement, followed by an orderBy statement. We can do essentially the same thing here. So given the employee DataFrame, all we have to do is select the ID of the employee and the last name of the employee. So now we've selected those two columns, and then we filter out everybody who does not live in Sydney. And finally, we order the records or the rows by the ID. And as always, it's nice to try this out in a real dataset. So this is the input dataset, so this what employee DF looks like. So, in this case we have these four records. And after running the Spark SQL expression here, we get a new DataFrame called sydneyEmployeesDF, like this, with only two columns. And then the ID numbers and last names of the people who live in Sydney, sorted by the ID Numbers. So David Walker is now first because he's 221. He lived in Sydney, so he's the first element, and then the next one is Slate Markham, who is now second because his ID is 645, and he also lives in Sydney. And then New York people are nowhere to be found. So here's an example of a few transformations on a DataFrame.

Filtering in Spark SQL

The DataFrame API makes two methods available for filtering:
filter and **where** (from SQL). *They are equivalent!*

```
val over30 = employeeDF.filter("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

```
val over30 = employeeDF.where("age > 30").show()
// +---+-----+-----+---+--------+
// | id|fname|lname|age|    city|
// +---+-----+-----+---+--------+
// | 12|  Joe|Smith| 38|New York|
// |563|Sally|Owens| 48|New York|
// +---+-----+-----+---+--------+
```

Filters can be more complex too:
We can compare results between attributes/columns. Though can be more difficult to optimize.

```
employeeDF.filter(($"age" > 25) && ($"city" === "Sydney")).show()
// +---+-----+-------+---+------+
// | id|fname|  lname|age|  city|
// +---+-----+-------+---+------+
// |645|Slate|Markham| 28|Sydney|
// +---+-----+-------+---+------+
```

Filter can be more complex too: 意思不是說 filter 可以比 where 複雜, 而是說 filter(或 where)可以寫得比你以為的 更複雜.

Also it's important to note that there are two methods available for filtering in Spark SQL. One is called filter and one is called where. Both of them are the same. One is just an alias of the other, so you can always think of these things as completely equivalent. When filter looks better write filter, when where looks better write where. And also note that it's possible to make filters a little bit more complex too. We can have Boolean expressions and, id or or. And we can also use several different columns in these expressions. So we can compare between columns as well. The only thing to notice is that if you do a lot of this, it can be a little more difficult for this Spark SQL query optimizer catalyst to optimize your queries.

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

For grouping & aggregating, Spark SQL provides:

- a `groupBy` function which returns a `RelationalGroupedDataset`
- which has several standard aggregation functions defined on it like `count`, `sum`, `max`, `min`, and `avg`.

How to group and aggregate?

- Just call `groupBy` on specific attribute/column(s) of a `DataFrame`,
- followed by a call to a method on `RelationalGroupedDataset` like `count`, `max`, or `agg` (for `agg`, also specify which attribute/column(s) subsequent `spark.sql.functions` like `count`, `sum`, `max`, etc, should be called upon.)

```
df.groupBy($"attribute1")
  .agg(sum($"attribute2"))
```

```
df.groupBy($"attribute1")
  .count($"attribute2")
```

The last topic I'm going to cover in this session is grouping and aggregating on DataFrames. I mentioned kind of awkwardly earlier on that the aggregate method is meant to be used after a call to the groupBy method. Let me show you a little bit more of what I mean here. So there are some special operations for grouping and aggregating, and this is because one of the most common things that you want to do with a table is to group data by some column, by some attribute, and then to do some kind of aggregation on that column, that attribute, like a count. So these things are a pretty common pattern. For grouping and aggregating, Spark SQL provides this groupBy function that I mentioned earlier, which returns this strange thing called a RelationalGroupedDataSet. This is when I said the type was simplified. We'll talk about that in a second. And then it has several standard aggregation functions defined on it. So these are defined on this RelationalGroupedDataSet, some of which include count, sum, max, min and average, amongst many others. So you have kind of this two step thing going on here, you've gotta do groupBy. You get back this RelationalGroupedDataset thing, and then you get some of these aggregation functions that you could potentially use. This can get a little hairy with the types, so the API docs are really your best friend here. But the basic pattern you have to remember is that you have some DataFrame, you do a groupBy on it, then you call some other method like agg here, which I'll talk about in more detail in a moment, or count. Things like this. So, this count operation is defined on this RelationalGroupedDataset thing, and it returns another DataFrame. So then you get back to DataFrames again. So this API basically requires you to go through two steps. First call groupBy and then then call some other operation to get you back to a DataFrame. This can look a little complicated so let's look at an example to clear it up.

Grouping and Aggregating on DataFrames: Example

Example:
Let's assume that we have a dataset of homes currently for sale in an entire US state. Let's calculate the most expensive, and least expensive homes for sale per zip code.

```
case class Listing(street: String, zip: Int, price: Int)

val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._

val mostExpensiveDF = listingsDF.groupBy($"zip")
                                .max("price")

val leastExpensiveDF = listingsDF.groupBy($"zip")
                                 .min("price")
```

So, let's assume that we have a dataset of homes currently for sale in one entire US state. Let's try to use these grouping and aggregating operations to calculate the most expensive and the least expensive homes for sale per zip code. So we want to do this per zip code. We want to find the most expensive and the least expensive per zip code in this entire state. So assuming that we have a DataFrame full of these listing things here, how could we compute these most expensive and least expensive homes for sale per zip code? I'm going to let you try it. Feel free to go back to the last slide if it helps. But see if you can use groupBy and some kind of aggregate function to do these calculations. Here's how we can do it. First things first, we're going to have to import another SQL import. So this one's called functions and it gives you these things here, like the max and the count and all of that. So you're going to have to import that if you're going to want to use any of these grouping and aggregation functions. Otherwise, you just do groupBy zip code, which is pretty straightforward, right? We're grouping by the zip code, and then per each zip code, we want to know what the maximum price is of the listing. So the price is of type Int. We can easily calculate the max. And we're grouping by the zip code, which is also an Int. So now we have the maximum price per zip code, and likewise we can do the same thing with minimum, so we can just groupBy the zip codes. So now we have groups of zip codes and then we can choose the minimum price you need to zip code. This is extremely readable, extremely straightforward. If you want to see how this looks on real data, I encourage you go to the data [INAUDIBLE] community edition notebooks and to try this out on a real dataset. You can almost exactly copy the code here. Okay, that was kind of easy, wasn't it?

Grouping and Aggregating on DataFrames: Harder Example

Example:
Let's assume we have the following data set representing all of the posts in a busy open source community's Discourse forum.

```
case class Post(authorID: Int, subforum: String, likes: Int, date: String)

val postsDF = ... // DataFrame of Posts
```

Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforum.

```
import org.apache.spark.sql.functions._

val rankedDF =
  postsDF.groupBy($"authorID",$"subforum")
         .agg(count($"authorID")) // new DF with columns authorID, subforum, count(authorID)
         .orderBy($"subforum",$"count(authorID)".desc)
```

Let's try something a little harder. Let's assume that we have the following dataset representing all of the posts in a busy open source community's discourse forums. So just think forums with subforums, okay? And we have this case class Post here which represents some activity or some post in this forum. Each post has an author ID, it has a location where this post was made, in this case, it was made in a specific subforum, and a number of likes for that post, as well as the date that it was posted. Now, assuming that we have a DataFrame pull up those posts, let's say we'd like to tally up each author's posts per subforum, and then rank the authors with the most posts per subforum, okay? So, basically what we want to do is we want to count up all of the posts that people are making in each subforum, then we want to basically figure out who was the person with the most posts in each subforum. So maybe we want to give this person a community award or something so we want to know who was the most busy in each one of these subforums, okay? How can we do this using these grouping and aggregating functions on DataFrames? I'll give you a few minutes to try it yourself. Okay, so first things first, we've gotta import our SQL functions here. And after that, we just gotta do groupBy and aggregate and an orderBy. So let's step through this. So we want to group author IDs and subforums together, because we're trying to figure out who's posted the most in each subforum. So we need both the author and also the subforums. And then we do an aggregation on these. In this case, I'd use this interesting aggregate method which itself take another call to a function. I actually didn't need to use this here, I could have just used count, but I wanted to show you what it looks like to use it. And then I specify that I want to do a count on the author IDs. This gives me a new DataFrame, with the following columns. I get an authorID column, a subforum column, and then a count column. So the number of counts for per that authorID in that subforum. Okay? And finally, I can just rearrange the order, and I use orderBy. So I have two parameters here because I want to first order by the subforums, and then I want to order by the counts. But notice I do this desc here, which says, okay, I don't want it to be in increasing order, I want it do be in descending order. So, the count column will be in descending order, okay?

Grouping and Aggregating on DataFrames: Harder Example

Example: Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforums.

```
val rankedDF = postsDF.groupBy($"authorID",$"subforum")
                      .agg(count($"authorID"))
                      .orderBy($"subforum",$"count(authorID)".desc)
// postsDF:
// +--------+--------+-----+----+
// |authorID|subforum|likes|date|
// +--------+--------+-----+----+
// |       1|  design|    2|    |
// |       1|  debate|    0|    |
// |       2|  debate|    0|    |
// |       3|  debate|   23|    |
// |       1|  design|    1|    |
// |       1|  design|    0|    |
// |       2|  design|    0|    |
// |       2|  debate|    0|    |
// +--------+--------+-----+----+
```

```
rankedDF:
+--------+--------+---------------+
|authorID|subforum|count(authorID)|
+--------+--------+---------------+
|       2|  debate|              2|
|       1|  debate|              1|
|       3|  debate|              1|
|       1|  design|              3|
|       2|  design|              1|
+--------+--------+---------------+
```

So what does this look like in real data? This is maybe not the most beautiful dataset that you're going to have a look at, but note that there are two subforums. They have actually similar looking names if you glance really quickly. One is called design, another is called debate. And let's just say I have three authors, authorID 1, 2, 3. Author ID 1 posts a lot in the design forum, for example. And this is what the result of the groupBy aggregate followed by the orderBy looks like here. So as you can see, I've ordered by the subforums now. So I've got the debate subforum first, followed by the design subforum. And then I have the counts in decreasing order. So the author with the most posts in the debate forum is author number 2, and the author with the most posts in the design forum is author number 1.

Grouping and Aggregating on DataFrames

API

After calling `groupBy`, methods on `RelationalGroupedDataset`:

To see a list of all operations you can call following a `groupBy`, see the API docs for `RelationalGroupedDataset`.
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset

API

Methods within `agg`:

Examples include: `min`, `max`, `sum`, `mean`, `stddev`, `count`, `avg`, `first`, `last`. To see a list of all operations you can call within an `agg`, see the API docs for `org.apache.spark.sql.functions`.
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

As you might have noticed, there's kind of two APIs here. There's this API that's available inside of this RelationalGroupedDataset thing here, and there's also an API available for use with this aggregate

method here. And each of these have individuals scalar doc pages that you can look at to see what operations are available to use. So there is a set of operations available within this aggregate thing here, and you can figure out what all those operations are by going to the spark.sql.functions page of the API docs, you can see a full list. And if you want to see a full list of the operations that you can call following a groupBy, have a look at the API docs for this RelationalGroupedDataset thing here.

DataFrames (2)

DataFrames (2)

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we're going to continue diving into the DataFrames API.

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

So far we've developed a little bit of an intuition about what DataFrames are, how we can create them, we learned how to do transformations and importantly also aggregations on DataFrames. So we have a little handful of operations that we know how to do in DataFrames those in particular relational operations. And in this session, we'll continue to focus on the DataFrames API. In this session, we'll cover how to work with missing values or how to deal with things like null. We'll look at common actions on DataFrames, so we saw transformations and aggregations, these were both transformations of some sort. But we didn't look at any actions, yet, or so we think. Then we'll dive into joints on DataFrames, and finally we'll talk a little bit about 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

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`.

So let's get started with cleaning data with DataFrames. Anybody who works with real data sets knows that data is never pretty when you start with it. Often it's full of unwanted values like null or NaN which stands for not a number. In these cases, it's often desirable to do something about these unwanted values in particular often we like to drop rows or records with unwanted values like null or NaN. Or sometimes we like to actually replace those values with something that is less problematic than null. For example, maybe if null is in a numeric column, we'd like to replace it with zero. Spark SQL offers a handful of methods to help you clean your data. For example on the dropping front, there are a number of methods or overloads of this drop method that we can use to clean up some of these unwanted values. The most commonly used is the method called drop without any parameters. And if you call this method on a data frame, what this does is it drops rows that contain either null or NaN in any column and returns a new data frame without those problematic values. Which means if you have some row with many values in it and most of them are good but one is null, then that entire row gets dropped. That behavior might not be desirable sometimes. Perhaps we know we can deal with that null value and replace it with something else in some other way. In that case you might want to use the drop method while passing all to drop. And in this case, what we do is we just drop rows that contain null or NaN in all columns. So in other words, that just means that if we have a row with all null values then we drop the entire row but only if all of the columns are null or NaN. And finally, there's another variant of drop that we can pass an array too which contains column names and this means that we only dropped more records that contain null values or NaN values in these columns. So that means we only drop columns for example if the id and the name has a null value but we don't care about other columns with null or NaN values for example.

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`.

The other class of operations that we can do to clean up our datasets has to do with replacing unwanted values with something else. There are few different variance of the fill method. This variant here that I have chosen to show if you pass a parameter to it, in this case, a number, 0, and then what we can do is replace all the occurrences of null or NaN in numeric columns. If we pass something numeric to fill with the specified values on this case. If there is a null or a NaN found in any column that has a numeric type, we replace the value with 0 instead. And of course we return a new DataFrame. This variant to fill gets a little bit more specific. In this variant we can pass a Map to fill. Where the keys are column names and the values are what will replace null or NaN with. So perhaps we want to handle different columns differently. So say we've got one column called minimum balance. And another column called name. Perhaps we want to replace minBalance if there's a null. We want to replace it with a 0. But if there was another column called name, we would maybe instead replace the null that we find in that column with some other string that could be more useful. And finally there's the replace method here which is good for replacing specific values. So let's say I messed up my data set accidentally perhaps there's a row or something isn't what it should be, I can make corrections with this replace method. In this case, what we are doing is we pass an array to replace and a map as well. So the array contains column ids and map contains values that should be replaced. So in this case, what happens is in the column called id, we replace the specified value which is 1,2,3,4 with 8,9,2,3 if come across 1,2,3,4 in the id column. So the idea is I can one off replace erroneous values for example using this replace method. So these are just a gist of some of the methods you can use to clean your data. These are the main variants of the three methods that we can use on DataFrames to clean up datasets, so drop, fill, and replace are the main methods. They each have a few variants, but we've covered most of them so far.

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.

You might have noticed that we covered transformations in the last session but we didn't say anything about actions. Just like RDDs, DataFrames had their own set of action and we've actually even used one of these actions several times already. In fact the actions that are found in DataFrames aren't that all interesting, they're actually very similar to the actions found on RDDs. So I'm not going to show you examples of each one of these things being used since we've already seen them many times already in RDDs sections but there is one that is particularly interesting and that's this one here called show. And this is the one we've actually already seen several times already. This one is rather special for DataFrames. So as we saw, this one displays the top 20 rows of the Dataframe in a tabular form so pretty prints it in a very nice way so it's easier to actually look at and visually parse table. The other common actions are actions that we already know such collect(), which returns an array that contains all of the rows in this case, of the DataFrame back to the master node. There's also the count operation, which returns the number of rows. So this is an operation that returns the number of rows specifically and then there's method first or head. These are both the same method which returns the first row, the entire first row in the DataFrame, so if you ever want to just take a peek at what's in your DataFrame, you can just grab the first row on top, using this first or head method. And finally, there's the take method which we're already also very well familiar with, the idea is return the first N rows back to the master node. So it returns an array of rows and once we get back this array of rows, we can then introspect it. And that's actually about it for actions. There aren't so many special actions on data frames. And in fact we're actually often using aggregations more than we're using these actions.

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")
```

Okay, so let's get into joins. What's pretty neat is that joins on dataframes are actually pretty similar to those on pair RDDs with the one major difference, that because we don't have key value pairs anymore, we have to specify which columns we should join on. Because with Pair RDDs, we were always implicitly joining on keys. Now, we have used a table, we have to say which of these columns in the table should behave like the key that we should focus on in our join. So just like in Pair RDDs, there are several joins to choose from. And we have to tell Spark which one we want to use. And to do a join, the syntax is just a little bit different. This is for example how we do an innerjoin. So assuming we have two DataFrames, one called df1 and the other called df2. We can do an interjoin by saying df1.join, and then passes a parameter df2. Now this here is how we tell Spark SQL which column should behave as keys. So in this case, each of these DataFrames, oops, this should be 2. Both of these should be 2. So in this case, each of these DataFrames DataFrames have a column called ID here, and we want to join on these ID columns. So what we do is we select the column via this dollar sign notation here. That we want to make sure that both of these IDs are equal when they're joined and that's it. We just pass the DataFrame that we want on the right side of our join, as well as which operators should be the keys, essentially. Unlike joins on RDDs, joins on DataFrames always use the same method named join h ere. In RDDs we were using different methods, left_outer join, right_outer join, to specify the different kinds of joins that we wanted to do. To specify that we want to do a different kind of join than an inner join all you have to do is pass another parameter. In this case we say right outer for the right outer joins. So we're saying that we want to do a right outer join with DataFrame one on the left and DataFrame two on the right. We want the ids again to be our keys essentially in our join. And then we say that we want we do a rewrite outer join, this is the third parameter here we passed the join. And that's all there is to it. Now and order to change this join type to anything else we can pass one of these strings instead of right outer. So if we wanted to change this to a left outer join we pass this string here. Or an outer join would just say outer for example.

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
```

So let's look at an example. Recall this CFF data set that we saw from earlier in the course. So this was the data set with the train passengers and the train tickets. And let's try to adapt that to the data frames API. So this is the code for the data that we use in our example. I've adapted it a little bit to put it in to a DataFrame format. So I've made everything case classes. So now we have case classes representing the subscriptions, they're called Abo. The ID is the key which is subtype int. And the value is a pair of strings which represent the customer's last name and the type of subscription that they have. So that's the case class Abo. In the case class Loc, which stands for location, the key is an integer and the value is a string. And this string represents the cities that this customer most often travels to. So assuming that I have a sample dataset here, I can create two different DataFrames, abosDF. By calling toDF on the RDDs that I created in the previous example, in the RDD example. So I just used this toDF method that we saw earlier. And now I have a couple of dataframes. I have dataframes for the subscriptions called abosDF. And I have a dataframe for the locations called locationsDF.

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|
//                                +---+---------+
```

注意只有左邊的表有 id=104 的.

And if I call the show method on each of these two DataFrames, this is what it looks like. So in the case of abosDF, I have three records, customer 101, 102, 103, 104. And here's their information. So this is the last name and this is the type of subscription they have. So this person has an [FOREIGN] which is a free train pass all over Switzerland. And these people had what's called a DemiTarif which is a half price fare card, okay? So they get a 50% discount on their train tickets. And in this data set I have customer numbers and the cities that these customers most frequently travel to. So for example customer 101 frequently travels to Bern and Thun for example. And customer 103 travels quite a bit, they go to Chur, St. Gallen and Zurich. So these are our two data sets, this is what they look like.

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"))
```

Now say we'd like to combine these two data sets. Such that we have a new data set that contains only customers that both have a subscription and where there exists location info. So how would you do it? I just showed you how to perform joins on DataFrames, so all you have to do is figure out the right join to use. And then you just have to put it into the shape that I showed you on the previous slide. So what would this program look like? Well we perform an inner join of course. Because, as the problem statement says. We want to combine these two data sets. Such that we have a resulting data set that contains only customers that have both a subscription and location in full force. So we want an inner join. That said assuming that we have abosDF and locationsDF, all we have to do is call abosDF. Then we say join locationsDF. We say, okay. Our key is going to be ID here. And that's it, that's all we do. This is by default an inner join if we don't pass a parameter to it specifying the join. So by default, join is a inner join unless we say otherwise.

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! :-)

And this is what the resulting data set looks like. So you'll wind up with both id columns and their resulting data set. But this is correct, notice that we have all customers, 101, 102, and 103 in here, and not customer 104. Because customer 104, if you recall, didn't exist in the locations data set. So it's correct that customer 104 is missing in this case, in the inner joined case.

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! :-)

So let's do another example. Let's assume the CFF wants to know for which subscribers, so this is the keyword here, subscribers. People who have subscriptions like the half fare card. We want to know whether or not we've collected location information for each of these subscribers. So it's possible to have the scenario where maybe somebody has some kind of subscription, like a DemiTarif. But they don't use the CFF app. So it's impossible for us to collect location information about them. It would be good to have a dataset where we could see situations like these. In case we'd like to suggest for example that this user should download the mobile app. So, if we wanted to join these two datasets together into that shows which location information is available, which join do we use? What program will be right with DataFrames to accomplish this task? I'll let you try it for yourself. Well, the answer is that we would choose a left_outer join. So we choose a left_outer join because we'd like to keep all of the keys in the abosDF data set. So in this case, this is the one where we make sure we have an entry for each one of these elements. Which means that our resulting data set looks like this. So notice that there exists a record on the left side here for every single person in the subscriptions data set, in the abos data set. And everybody who has a subscription seems to use the mobile app except for customer number 104. This person has no values, so in this case this is a customer for which we have no location data. But we do have this person in our dataset, which is what we wanted. So now we could do a filter on this dataset. And we could then know to suggest to user number 104, hey, would you like to download the mobile app? So, as expected, customer number 104 has returned because we chose a left outer join.

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
```

Okay, so now that we're familiar with the DataFrames API, we're familiar with all of the major operations. We've seen all of the untyped transformations that we care about using. We've seen aggregation and grouping operations. We've seen joins. We've seen actions. Let's revisit that example that we looked at in the first session on Spark SQL. So remember this example about selecting scholarship recipients with a certain dataset. In this example, we imagine that we're an organization called CodeAward. That offers scholarships to programmers who have overcome some kind of adversity. And we have two dataset in this example. We have one full of records represented by this

demographic case class. Which contains demographic fields about a certain person. So each person has an ID number, they have an age. They have other information associated with them like the country that they're from. Or whether or not they served in the military, for example. And the other data set that we had Was one containing financial information. So we had many instances of this finance data type here which also contained an ID and then had financial information such as whether or not this person had debt, whether or not this person had financial dependence or student loans and also income information. So if we rearrange that data set into DataFrames. So imagine then we have two data frames full of these two pieces of information. So we have two data frames. One representing this demographic information and another one representing this financial information. One is called demographicsDF and the other one is called financesDF.

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
```

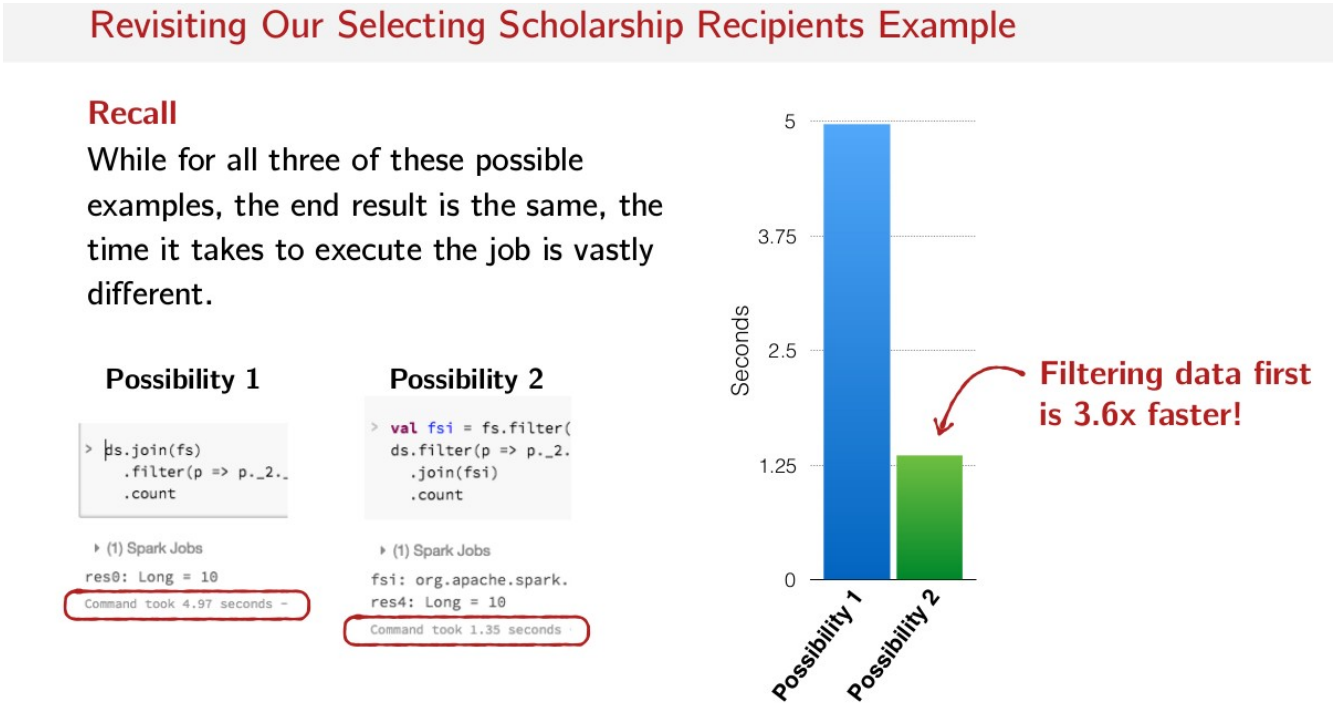
So let's imagine now that our goal is to tally up and select students for a specific scholarship based on some kind of criteria. And if you recall the criteria that we chose was looking for students that are from Switzerland and who have debt and some financial dependent. This is the criteria that we want to search for in order to figure out how many people are eligible for a certain scholarship. So again we want to count these up, we want to figure out how many people are eligible for a scholarship of this criteria. How might we implement this program with the DataFrame API? If you recall, I showed you three different possible solutions to this problem. And I showed you also that the performance was vastly different in each case. So think about what you can do to achieve good performance when solving this problem.

Revisiting Our Selecting Scholarship Recipients Example

With DataFrames:

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

Well, that was sort of a trick question. Because if you use DataFrames, the sales point is that all the optimization should happen automatically for you thanks to the Catalyst query optimizer under the hood. That means that you can just write this query however is most comfortable, and Spark can usually optimize the unnecessary operations, and make it as fast as possible. So in this case, I would just do the most natural thing, like what we did in solution number one when I first showed this example to you. If you recall in solution number 1, the most natural thing to do is to first do a join and then to filter out the people who didn't meet the criteria that we had established on the previous slide, so. So our steps were join first, and then filter down our dataset so that it meets the criteria as a second step. And finally to sum it all up. So looking a little bit more carefully at our code, we take the demographics dataFrame. We called join, we passed to it the finances DataFrame and then we say that we want to chose as keys the IDs from both of these DataFrames. And finally we are just explicit here about the fact that we want to do an inner join, we want to keep all keys that exist in both datasets. And then we filter out, and then we pass through the people in the filter that have debt and they have financial dependents. And we pass through the people who live in Switzerland. And finally, we count the rows in our resulting data frame.

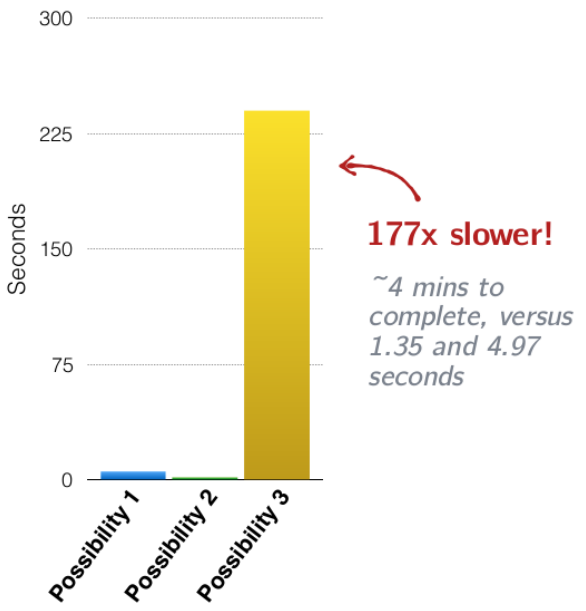


So if you recall the performance numbers from the first session in this sequence of sessions about Spark SQL, you'll remember that the first solution, the one we did join first, filters later, took 5 seconds to complete. Whereas the second possibility which did the filters first and then join second, took only 1.3 seconds to complete. So, the second possibility where we re-ordered things and put the filters first and then joined second, so that we were joining less data had a 3.6 times performance boost.

Revisiting Our Selecting Scholarship Recipients Example

Recall

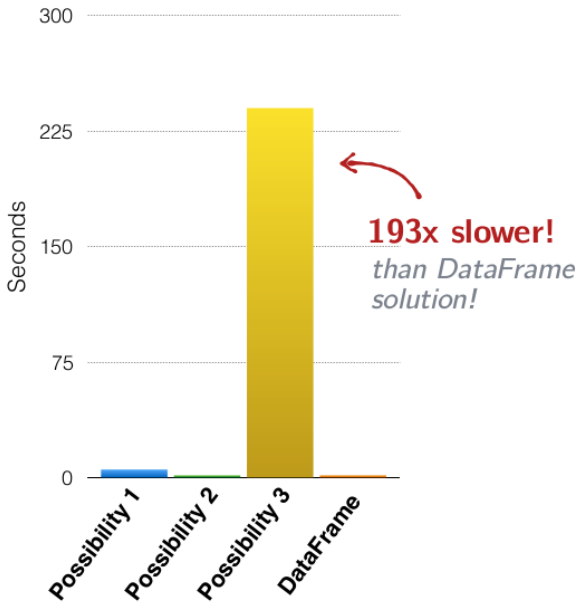
While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



There's also a cartesian product that we could do that was 177 times slower.

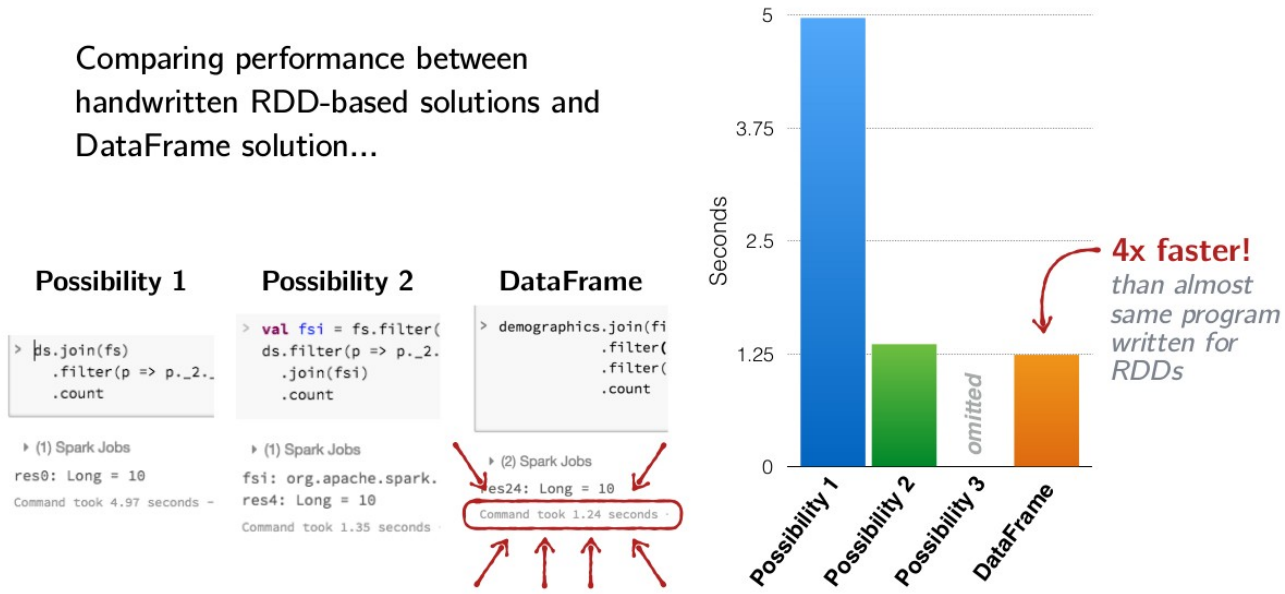
Revisiting Our Selecting Scholarship Recipients Example

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



So how does the DataFrame solution that we just look at compared to these other three possibilities? Well, this is what it looks like. The DataFrame example's performance numbers are here. And actually it's even faster than these other two possibilities here because the cartesian product version is a 193x slower than this DataFrame version here.

Revisiting Our Selecting Scholarship Recipients Example



So the DataFrame version actually took only 1.24 seconds to complete as compared to 1.35 seconds for the filter filter join version. And 4.97 seconds for the join filter version, right? And notice I do join filter filter, it doesn't matter because, so what Spark does under the hood is it reorders these operations to try and get the best performance. And in this case, this is where we're at. It's 4x faster than this first naive solution here that we did with the join first on a regular RDD. So that's a pretty big difference, isn't it? Imagine that if instead of seconds or hours.

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.

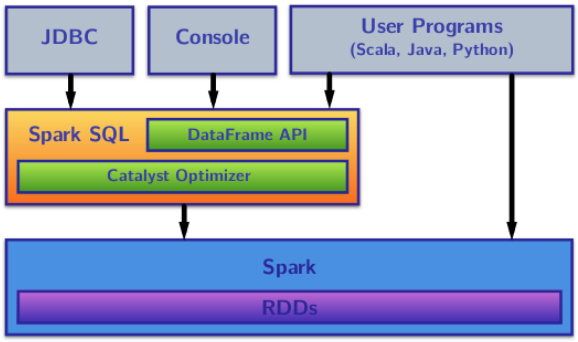
Okay, so let's briefly cover this sort of optimizations that Spark does. I'm not going to dive deep into the query optimizer or which is a data encoding framework. I'm just going to give you a general gist of what these things do. To recall that Spark comes with two specialized backing components. On the one hand it comes with Catalyst which is this query optimizer that I mentioned and on the other hand it comes with this thing called Tungsten which is an off-heap data encoder or a serializer.

Optimizations

Catalyst

Spark SQL's query optimizer.

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



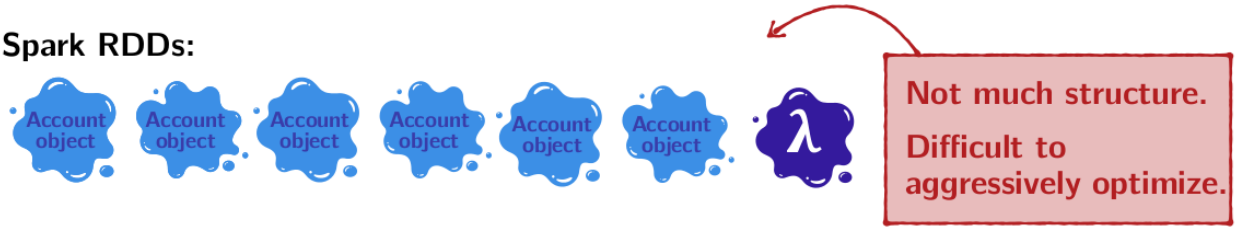
Key thing to remember:
Catalyst compiles Spark SQL programs down to an RDD.

So focusing on catalyst, remember this map that we saw earlier on about how Spark SQL fits into the whole Spark ecosystem. Spark SQL sits on top of regular Spark in RDDs. It takes in user programs with this relational dataframe API. So users are writing this relational code. And ultimately it spits out highly optimized RDDs that are run on regular Spark. And these highly optimized RDDs are arrived at by this tabulus optimizer here. So on the one hand you put in relational operations. Catalyst runs, and then we get out on the other end RDDs. So what's important to remember is that catalyst compiles Spark SQL programs down to an RDD.

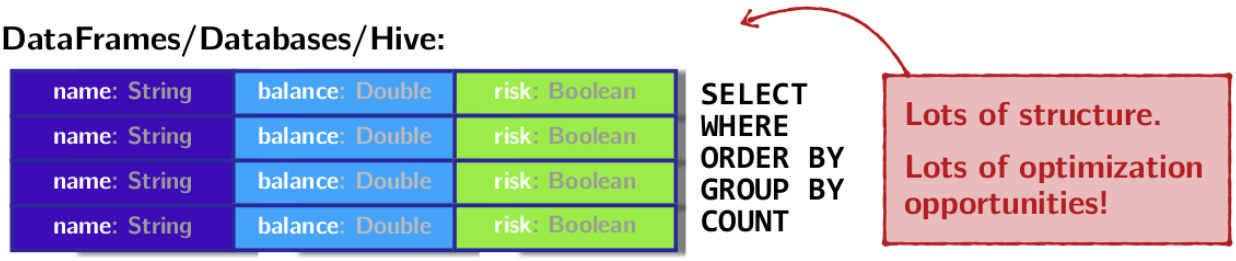
Optimizations: RDDs vs DataFrames

In summary:

Spark RDDs:



DataFrames/Databases/Hive:



So if you recall this picture from an earlier session, on this Spark RDD side of things, there's not a lot

of structure. Things are just blobs of account objects for example or some kind of objects. We don't have very much visibility into the structure of this data. And then we have this old peg operation, so peg function operation that does some kind of operation that we don't know what it is because we can't see inside of it. So there's a lot of ambiguity here. Not a lot of structure that we can easily see, which makes it really difficult to aggressively optimize. And on the other side we have these tables in dataframes that are very, very structured, very, very rigid. And we have this set of operations which is also very, very structured and very, very restricted. So we know more about what a user intends to do when they write in this sort of syntax than if they write in this sort of syntax here.

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.

So assuming that Catalyst has this information, assuming that it has a full knowledge and understanding of all the data types in the program, assuming it knows the exact schema of our data, so that it knows which fields that we have in our data set. And assuming that it has detailed knowledge of the computations that we would like to do, so assuming that it has these three things, there are number of optimizations that it can do. So one example is reordering operations. So the fact that all of our operations are still lazy and that we now have this structure gives us the ability to analyze and rearrange this gag of computations or logical operations that the user would like to do before they're actually executed and run in the cluster. So before this gag of operations, runs as an RDD. We can rearrange this stuff real quick. That makes it possible for Catalyst to rearrange and fuse together filter operations and push all filters as early as possible or push operations down to the data layer. Pushing filters up enables us to filter data out so we can do expensive operations on less data, like we saw with the join examples.

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.

Another sort of optimization that Catalyst can do is that it can reduce the amount of data that we must read. So, if we have a full knowledge of all data types that are possible to be used in the program. And we know the exact scheme of our data and we know the exact computation that will be done, I know exactly which field might not be used in a computation. This means I can move less data run the network because if we recall a few slides back. So if there are numerous fields that we don't need in our computation, in one of these blob of objects that exists inside of an RDD. In this case, we don't know that, because these are blobs of objects that we don't know much about inside of an RDD. There's no way for us to somehow focus on only a small piece of these objects, because, like I said earlier, we have no idea what's inside of these things. So that means if we have to do some kind of operation that involves Shuffling data over the network or moving some of these objects around. That means we have to serialize this entire thing and send it to another machine even though we know we don't need most of that. Not knowing the structure of the data could indeed be more expensive later. Also, it wastes memory if we don't need most of this sitting in memory. So, Catalyst can narrow down, select, and serialize, and send around only relevant columns of our data set. So, we don't have to send the full data set all over the network if we don't need to.

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.

Another kind of optimization that Catalyst does is pruning unneeded partitions. So, it analyzes the DataFrame and filter operations to figure out and skip partitions that are unneeded in some computation. So it can know in advance that we don't have to run a certain computation on certain partitions, which, of course, also saves some time. Of course, there are many more optimizations that Catalyst does, so I hope this gives you a little bit of an intuition about what some of these optimizations actually do.

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)

The other backend component that I mentioned earlier is called Tungsten, which is Spark's off-heap data encoder. So since we have a very restricted set of data types that we know literally everything about, that gives Tungsten the ability to provide highly specialized encoders to encode that data. So, we

know how to represent that and we know how to serialize these things super efficiently, which is column-based. And I'll talk about why that's great in a moment, but its also off-heap, so that means its free from garbage collection overhead. So what I mean by highly specialized data encoders is that Tungsten can take schema information and knowing all of the possible data types that a schema can be built out of. It can super optimally and tightly pack the serialized data into memory. So this means more data can fit into memory and we could have faster serialization and deserialization, which is a CPU bound task. So this is a highly optimized, highly specialized, super performant data decoder. The idea is that we keep data serialized in a special format in memory, so that we can keep more data in memory and we can access it more quickly.

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.

The next feature of Tungsten is that it stores everything in a column-based format. So storing tabular data in a columnar format actually comes from the observation that most operations that are actually done on tables tend to be focused on doing operations on specific columns or attributes of the data set. So, when storing data, it's actually more efficient to store data in these columns, grouped in these columns rather than storing them grouped by these rows. The idea is that it would take more time to look up each individual element, some more specific in a row, than it would to just grab an entire column of data knowing that we have the entire thing already logically grouped together. And this is actually well known to be most efficient across all kinds of different database systems. So these highly specialized data encoders and decoders, they store everything in a column-based format.

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.

And finally, everything is off-heap. So Tungsten knows how to tightly pack all of this data in this columnar format off-heap in memory that it manages itself. So that means that we can avoid garbage collection overhead and garbage collection pauses and things like that. So to summarize, Tungsten lets us to put more data in memory and gives us a lot faster access to that data that's stored in memory.

Optimizations

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

So taken together these two systems, Catalyst, the query optimizer on one hand, and Tungsten, the data encoder that's super performant on the other, these two systems offer ways to significantly speed up your jobs, even if you initially write your jobs very efficiently. So it knows how to rearrange these computations, it can store these computations in an efficent format. It can reduce the amount of data that it needs to send over the network because it knows what sorts of patterns of computation you want to do. So, Catalyst and Tungsten together are actually what provide these enormous speed ups.

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!

Well, that said, you might be wondering well, man, these data frames, they seem to good to be true. Of course they come with some limitations, so let's talk about some of those now. The first major limitation that you're going to come across, which maybe we haven't shown you in these slides, is that they're untyped. I mentioned that they're untyped but we really weren't bitten by it. So what happens a lot of the time with these untyped APIs is that code like this will compile even though there doesn't exist a column called state. So this code happily compiles and it runs but an exception is thrown at some point saying oops, sorry, nope. I can't find state, I only have columns street, zip, and price. I don't have a column called state. So this can be pretty annoying because errors aren't caught at compile time, instead they're caught as exceptions later on. It would be really nice if this error here was caught at compile time like we're already used to. We're used to the compiler catching us when we make errors like this, so this is a little bit uncomfortable if you come from Scala. It also means that you'll find yourself having to cast elements in rows to certain types that you know exist in the schema because rows aren't typed. So, this can be not just annoying because you have exceptions every now and then, but it can also be a little bit verbose and hard to read because you'll find yourself writing row.AsInstanceOfInteger, for example, which nobody wants to really do. We want to just talk about specific fields, for example. So things being untyped in DataFrames can be a little bit annoying sometimes, especially when you come from Scala and you're really used to this nice compiler that helps you out a lot of the time.

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.

Another limitation is that we can only use a limited set of data types. So if your data can't be expressed by case classes or products or these standard Spark SQL data types, it might be difficult to ensure that a Tungsten encoder exists for your data type. So one example of a situation where using DataFrames might be problematic is if you have an application which already has some kind of regular Scala class that's super complicated, and you want to create a DataFrame or a data set of this super complicated thing that you already have. In this case it might be difficult to break it up and represent it as case classes, products, and these SQL data types. So having these limited data types can sometimes be difficult. A big reason that DataFrames can be difficult to use is that perhaps your data set isn't semi-structured or structured, perhaps you have an unstructured data set. So in this case, DataFrames wouldn't be appropriate because you don't know the structure of your data, because you don't have any real structure in your data. You don't have some kind of schema. It's not self describing like JSON is or it's not some existing table that already has an existing schema. There is no structure to it, so these cases it might be better to just use regular RDDs.

Datasets

Datasets

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we're going to talk about the last and the newest major Spark API called Datasets.

Example

Let's say we've just done the following computation on a `DataFrame` representing a data set of `Listings` of homes for sale; we've computed the average price of for sale per zipcode.

```
case class Listing(street: String, zip: Int, price: Int)
val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._
val averagePricesDF = listingsDF.groupBy($"zip")
                                .avg("price")
```

Great. Now let's call `collect()` on `averagePricesDF` to bring it back to the master node...

To motivate Datasets, let's start with an example. Let's say we've just done the following computation on a DataFrame. And let's just say that our DataFrame is full of these things called listings here, and each one of these listings represents a home for sale on the market. What we've done with this data is we've calculated the average price of house for sale per zip code. So we've gone through all these listings and for each zip code we calculated the average price of the homes for sale in that zip code, okay? So that's what the program looks like here. So all we do is we take our listings DataFrame, we say groupBy the zip code, so now we have everything collected in groups per zip code. And then finally, for each group what we do is we call the average function and we pass to it the name of the price column because we want to calculate the average of that column. So that's the computation that we've done. Great, wonderful, seems just fine. Now let's do something completely normal, let's call collect() on it to kick off this computation, and let's try to bring back averagePricesDF in the form of an array back to the master node.

Example

```
val averagePrices = averagePricesDF.collect()
// averagePrices: Array[org.apache.spark.sql.Row]
```

Oh no. What is this? What's in this `Row` thing again?

Oh right, I have to cast things because `Rows` don't have type information associated with them. How many columns were my result again? And what were their types?

```
val averagePricesAgain = averagePrices.map {
  row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int])
}
```

Nope.

```
// java.lang.ClassCastException
```

Oops or no, or what was this thing again? I don't remember, I thought I was getting back an array of doubles representing the average prices. What is this row thing here? We've seen them in the last few sessions, but we didn't really look at them very closely. What's in this row thing again? Okay, that's right, because a row is untyped, we have to cast the stuff in the row, but that means that I have to remember the order of the columns in that row so I know which things to cast. So you might ask yourself, okay, well, then how many columns was my result again? And what were the type of those things, because now I've gotta count them and then indexed into them and then cast them to the correct types, so here's one attempt. So I have a value called averagePricesAgain. And what I do is I take the array of row objects, and for each row I say, okay, the first thing in my row I'll just cast it to a string because that's something, I don't remember what that was, but I think the second one was the price, let's just cast that to an integer and let's see how it works. Nope, got a ClassCastException, so that was wrong.

Example

Let's try to see what's in this Row thing. (Consults Row API docs.)

```
averagePrices.head.schema.printTreeString()
// root
//  |-- zip: integer (nullable = true)
//  |-- avg(price): double (nullable = true)
```

Trying again…

.price

```
val averagePricesAgain = averagePrices.map {
  row => (row(0).asInstanceOf[Int], row(1).asInstanceOf[Double]) // Ew...
}
// mostExpensiveAgain: Array[(Int, Double)]
```

yay! 🎉

Wouldn't it be nice if we could have both Spark SQL optimizations and typesafety?

Let's try again. Instead of randomly trying different casts, let's look up the API docs for Row. Okay, so I can call head on my array of rows. And then once I have a row, apparently there's a method called schema, which gives me the schema, and then I can pretty print it nicely in the form of a tree. Okay, wonderful, so this is my schema. The first element in the row is an integer, and the second element in the row is a double, that's great. Okay, so my average price is this one here. Good to know. Okay, so let's try that again. Hooray, so now I've casted this correctly. So this is an integer, it matches with the first element in the row, and then the second element is double. Wonderful, it all works. It's just really ugly, it's really hard to read, I have no idea what's happening here. And as you could imagine, I could be randomly trying different casts until something works, and I could still be wrong, so this is pretty error prone. But hey, at least it worked. Though wouldn't it be nice if we could have both these wonderful Spark SQL optimizations and actual typesafety, so I didn't have to do this. I would really much rather select the price in a way like this, where I just call .price on something and it turns out to be all the right type, and everything works. This is really inconvenient. So wouldn't it be cool if I could have these Spark SQL optimizations as well as that typesafety, because on DataFrames we don't have that typesafety, everything is untyped, remember?

DataFrames are Datasets!

DataFrames are actually Datasets.

```
type DataFrame = Dataset[Row]
```

What the heck is a Dataset?

- Datasets can be thought of as **typed** distributed collections of data.
- Dataset API unifies the DataFrame and RDD APIs. Mix and match!
- Datasets require strucutred/semi-structured data. Schemas and Encoders core part of Datasets.

Think of Datasets as a compromise between RDDs & DataFrames.

You get more type information on Datasets than on DataFrames, and you get more optimizations on Datasets than you get on RDDs.

Well, that's where Datasets come in. They're kind of the Holy Grail, because they combine these nice optimizations with typesafety. Okay, maybe that's too extreme of a metaphor, but you get what I mean. Now before I go any further, I have to admit that I've been holding some information from you that's rather important information, and that is that DataFrames are actually Datasets. So this whole time we've been talking about this thing called the DataFrame, it's actually really a Dataset with type Row. So DataFrames are actually Datasets, they're actually the same thing. But okay, what the heck is a Dataset then? Well, a Dataset can be thought of as a typed distributed collection of data. So kind of in the same way that we think about an RDD. It's a typed distributed collection of data. However, Datasets have a little bit more than RDDs. So, the Dataset API actually unifies this DataFrame API with the RDD API. So let's think back to the last few sessions where we're looking at these very relational operations that looked very different from the operations that we learned in RDDs. Now, we can call both sets of these operations on a same Spark-distributed collection. What's even cooler is that we can mix and match these things, so we can have both functional operators and relational operators all intermingling in the same code. And finally, one other major difference between Datasets and RDDs is that, like DataFrames, Datasets require structured or semi-structured data, so your data has to have some kind of schema associated with it in order for it to be a Dataset. Another way to think of Datasets is to think of them as a compromise between RDDs and DataFrames. So, on the one hand, you get more type information with Datasets than you did on DataFrames, but on the other hand you get more optimizations on Datasets than you got on RDDs. So, Spark was able to do a few simple optimizations on RDDs like some pipelining optimizations but it couldn't do all of the cool stuff that the Catalyst optimizer could do. So, Datasets gets some of those optimizations, while having that nice flexible functional API as well.

DataFrames are Datasets!

Example:

Let's calculate the average home price per zipcode with Datasets.
Assuming listingsDS is of type Dataset[Listing]:

```
listingsDS.groupByKey(l => l.zip)      // looks like groupByKey on RDDs!
      .agg(avg($"price").as[Double])  // looks like our DataFrame operators!
```

We can freely mix APIs!

Let me tell you what I mean by mixing and matching these two APIs. So here's an example of that same computation that we're looking at a few slides back while we were trying to calculate the average price of all of the homes for sale in different zip codes. So let's assume that we have a Dataset called listingsDS, it's the same shape as it was before, it's a case class with some fields in it, but the most important field is, of course, the price in the zip code field. And the first thing we do is we can do groupByKey on it. Remember groupByKey from RDDs? And then look at this, this is super cool, we're passing a lambda to the groupByKey function. In this case, what we're doing is we're telling our DataFrame which one of our columns is going to be our key, so we pass a function to do that. So here we are, passing functions, it looks a lot like RDDs again. And the next operation is one of these aggregation operations that we became familiar with on DataFrames. So here we are, calculating the average per key grouping. And then we're telling Spark that the type of that calculation is a double. The point here is to show you that in the same line of code I can have both of these different APIs or both of these different ways of thinking about writing code. I can have this nice functional thing here, and then on the next line it can have a very relational expression. So, I can mix and match these things however I want. And the types match up because everything is a Dataset, so it all works out.

Datasets

Datasets are a something in the middle between DataFrames and RDDs

- You can still use relational DataFrame operations as we learned in previous sessions on Datasets.
- Datasets add more *typed* operations that can be used as well.
- Datasets let you use higher-order functions like map, flatMap, filter again!

Datasets can be used when you want a mix of functional and relational transformations while benefiting from some of the optimizations on DataFrames.

And we've *almost* got a type safe API as well.

So just a few more observations to give you a little bit of an intuition about what Datasets are all about. So Datasets are kind of something in the middle between DataFrames and RDDs. Importantly, a DataFrame is just a Dataset, so they're the same type, which means I have the same operations on both of them. So you can still use all the relational DataFrames operations that we learned in the previous

two sessions now on Datasets, so we can just effortlessly carry over the knowledge of that API to this session now and use everything like we've used it before. In addition, Datasets add more typed operations that can also be used. So these typed operations include, actually relational operations, so we can have both typed and untyped relational operations. And then of course we have these functional transformations which are all typed as well. So now we have this higher-order functions like map, flatMap, and filter again, which we didn't really have on plain data frames before, they were mixed together with Datasets. 甚麼時候用 Dataset: Datasets are a good choice of abstraction when you want to mix and match functional relational transformations while still benefitting from some of the optimizations that we saw in DataFrames. And of course, we can't forget that it's got almost a type safe API as well, so it's not completely, perfectly type safe but it's pretty type safe, let's just say.

Creating Datasets

From a DataFrame.
Just use the `toDS` convenience method.

```
myDF.toDS  // requires import spark.implicits._
```

Note that often it's desirable to read in data from JSON from a file, which can be done with the `read` method on the `SparkSession` object like we saw in previous sessions, and then converted to a `Dataset`:

```
val myDS = spark.read.json("people.json").as[Person]
```

From an RDD.
Just use the `toDS` convenience method.

```
myRDD.toDS  // requires import spark.implicits._
```

From common Scala types.
Just use the `toDS` convenience method.

```
List("yay", "ohnoes", "hooray!").toDS  // requires import spark.implicits._
```

Okay, so let's get right into how to create these things, so since Spark 2.0, Creating Datasets is super easy. You can create datasets from data frame, all you've gotta do is call toDS on your dataframe, which is a convenience method that creates a new dataset from a data frame. Though note in order to use these toDS convenience methods you need to import this thing called spark implicits. Another way to create a Dataset is to do it using these operations for reading in structure, to semistructure data from file. So if you remember from the last session when we were using these to create DataFrames, now if you provide some type information. So in this case we say, okay, this json file here is full of Person instances. And if we define a case class whose structure and names and types on that with the structure and scheme on this people.json file. Then this Dataset will be read into memory from file and will be perfectly typed. Until now we've always had to do some kind of parsing or some kind of casting, types in order to get the Datasets into memory or with the right types. So this suddenly makes things a lot easier, we can also create a dataset from an RDD by using the same convenience method. And the same even goes from common Scala types, we can just call toDS on this list, for example. And there you go we have a new dataset of type string.

Typed Columns

Recall the `Column` type from `DataFrames`. On `Datasets`, *typed* operations tend to act on `TypedColumn` instead.

```
<console>:58: error: type mismatch;
 found   : org.apache.spark.sql.Column
 required: org.apache.spark.sql.TypedColumn[...]
                .agg(avg($"price")).show
                     ^
```

To create a `TypedColumn`, all you have to do is call `as[...]` on your (untyped) `Column`.

```
$"price".as[Double]  // this now represents a TypedColumn.
```

Before I can dig in to some of the operations on DataFrames, I've gotta remind you about these Column things here. So recall Columns from the DataFrame sessions, we're always using these Column things, and there was this syntax for using them. In Datasets, because everything is now typed, we now have the notion of typed Columns. So if you try to use the same old syntax (即上圖的.agg(...).show) that we've been using in the previous sessions, you'll get a type error saying nope, this is a column and I was expecting a typed column and you might ask. Well okay, how do I create a typed column then? Well it's pretty easy you just have to use this as method. With the type that you're column contains and then this creates a new typed column. This is going to be important throughout this session because many methods will expect typed columns.

Transformations on Datasets

Remember *untyped transformations* from `DataFrames`?

The `Dataset` API includes both untyped and typed transformations.

- **untyped transformations** the transformations we learned on `DataFrames`.
- **typed transformations** typed variants of many `DataFrame` transformations + additional transformations such as RDD-like higher-order functions `map`, `flatMap`, etc.

These APIs are integrated. You can call a `map` on a `DataFrame` and get back a `Dataset`, for example.

Caveat: not every operation you know from RDDs are available on Datasets, and not all operations look 100% the same on Datasets as they did on RDDs.

But remember, you may have to explicitly provide type information when going from a `DataFrame` to a `Dataset` via typed transformations.

```
val keyValuesDF = List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(3,"-)"),(2,"cre"),(2,"t")).toDF
val res = keyValuesDF.map(row => row(0).asInstanceOf[Int] + 1) // Ew...
```

Okay, so let's get right into the transformation on datasets. So remember how I kept on reminding you that transformation in DataFrames are called untyped transformations? Well that's because they didn't send API introduces a whole hast of typed transformations as well. So you have also typed variance of some of the untyped transformations as well. So, on the one hand we have these untyped transformation which are the same ones that we learned in the data frame session, and on the hand we have typed transformation. So like I said, these are typed variants of many the DataFrame transformations, and then we also have additional transformations like these high order functions on RDDs that we like so much. So when you look at the data sets API you're going to find transformations grouped into these two categories and it's going to be very important. Because if you accidentally use it on untyped transformation, then your nice dataset can lose all of its type transformations. You're going to have to keep these two things straight in your mind Because everything is a Dataset, because both frames and data sets are data sets, these APIs are seamlessly integrated. That means you can call a map actually on a DataFrame even though I didn't show that to you in a previous session. But when you do call a map on a DataFrame you end up getting back a DataSet for example. And of course this can be problematic because if you have no type information, how do you suddenly create type information? So here's an example down here below. Let's see if we have a data frame of pairs here, and now, I want to do a math bonnet, and I want to increment all of the digits in the keys here in these key value pairs. But again, I had this ugly thing going on here where I have rose And I have to cast each element to them if I want to do something with them. So in this case, in order to add 1 to the first digit here in the pairs, I've got to do this ugly cast and then add 1. It's really not clear what I'm doing. So while these two APIs are seamlessly integrated, you can accidentally do things like this (上圖最後一行) and end up with no type information at all when at one point you did have type information. So again, you got to be careful about that. And also, it's important to note that not every operation that you know from RDDs are available on datasets. And even operations that are available on datasets that you've seen on RDDs, they don't always look 100% the same on datasets as they did in RDDs. So you can't just, without looking at the API docs start programming RDD's on top of dataset. We're going to get lots of compiler, so just keep that in mind.

Common (Typed) Transformations on Datasets

`map`	`map[U](f: T => U): Dataset[U]` Apply function to each element in the Dataset and return a Dataset of the result.
`flatMap`	`flatMap[U](f: T => TraversableOnce[U]): Dataset[U]` Apply a function to each element in the Dataset and return a Dataset of the contents of the iterators returned.
`filter`	`filter(pred: T => Boolean): Dataset[T]` Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.
`distinct`	`distinct(): Dataset[T]` Return Dataset with duplicates removed.

So let's go through some of the common typed transformations on datasets. All of these look pretty familiar, don't they? So we have a map operation which takes the function from TDU and returns a new dataset. Just like all the other maps that we're used to the same is true for flatmap, the same is also true

for filters. A filter takes a predicate and it returns a dataset out, distinct is also the same, distinct returns a new dataset with all the duplicates removed. So these are all transformations because they return datasets, and they are all typed because the key just type the information around. So I hope these familiar operations make you comfortable.

Common (Typed) Transformations on Datasets

groupByKey	**groupByKey[K](f: T => K): KeyValueGroupedDataset[K, T]** Apply function to each element in the Dataset and return a Dataset of the result.
coalesce	**coalesce(numPartitions: Int): Dataset[T]** Apply a function to each element in the Dataset and return a Dataset of the contents of the iterators returned.
repartition	**repartition(numPartitions: Int): Dataset[T]** Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.

Because already with groupByKey, things start getting a little weird again. So coalesce and repartition look the same as what we're used to. But groupByKey gets a little weird, because it has this return type called KeyValueGroupedDataset, which is a little strange. How is this a transformation if it doesn't return a dataset, right? Well we'll see in moment they're ultimately grouped by key in two steps will return a dataset. But we're going to see right now that ultimately groupByKey does return a new dataset It just does so with an extra step in the middle.

Grouped Operations on Datasets

Like on DataFrames, Datasets have a special set of aggregation operations meant to be used after a call to groupByKey on a Dataset.

- calling groupByKey on a Dataset returns a KeyValueGroupedDataset
- KeyValueGroupedDatasetcontains a number of aggregation operations which return Datasets

How to group & aggregate on Datasets?

1. Call groupByKey on a Dataset, get back a KeyValueGroupedDataset.
2. Use an aggregation operation on KeyValueGroupedDataset (return Datasets)

Note: using groupBy on a Dataset, you will get back a RelationalGroupedDataset whose aggregation operators will return a DataFrame. Therefore, be careful to avoid groupBy if you would like to stay in the Dataset API.

So let's look at grouping operations on datasets now, so if you remember in the last session on DataFrames we also had this weird two step process. We had this special set of aggregation operations that were meant to be used after we called, in the case of DataFrames, after we called group by, just regular groupBy. But on Datasets we call groupByKey Again we have a special type called aggregation operations that can be used on the result of the groupByKey that was called on Dataset. So it's the same two step thing that we saw on the last session, in the case of groupByKey we return a different type called KeyValueGroupedDataset. You're going to have to remember the name of this if you want to look up some of the aggregation operations that you can use. And in this key value group dataset thing there are a number of aggregation operations defines which return datasets. Okay, so how do we actually group and aggregate on datasets? Well, you call groupByKey on the dataset you get back one of these things, and then you just call a method on their result of groupByKey that you've gotten out of this KeyValueGroupDataset thing here, and it returns a dataset. And that's how you do a group and then aggregate on the dataset. So we'll come back to a by key a little bit later, but let's first talk about some more.

Some KeyValueGroupedDataset Aggregation Operations

reduceGroups	**reduceGroups(f: (V, V) => V): Dataset[(K, V)]** Reduces the elements of each group of data using the specified binary function. The given function must be commutative and associative or the result may be non-deterministic.
agg	**agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]** Computes the given aggregation, returning a Dataset of tuples for each unique key and the result of computing this aggregation over all elements in the group.

Stopa:
15:22
https://www.coursera.org/learn/scala-spark-big-data/lecture/yrfPh/datasets?authMode=login

So let's talk about this KeyValueGroupDataset thing here. Let's talk about the aggregation operations that are available to us on that thing after we do a group by key, there are several operations, I'm going to list just a couple. One is called reduceGroups. So what this thing does here, it's basically a reduce function, but it does a reduce on the elements of each group. So after the group by key was done, we could look at the individual elements that represent the collection of values that have been grouped, and then we do a reduce on that. So that's reduced groups, and that returns a Dataset when it's done. The next thing is called Aggregate, which is this kind of mysterious general aggregation function, that we saw in the previous session on data frames. And this can be pretty general, this aggregation can be pretty general. Also note that it takes a parameter, a typed column rather than a regular column. So this is basically a typed variant of this general aggregation method that we saw on previous session on data frames. But if you remember, there was also an API on this thing too. Let's have a look at that more closely. So just like in DataFrames, there was this general aggregation function. And it has this mysterious typed column argument, which I don't know what to do with. If I pass the typed column to it, what does it aggregate? Usually, we want to calculate an average, or a mean, or a center deviation of something. Well, typically if we go to this function object that's defined in this Spark SQL package and we select one of dozens of different aggregation optimizations, there are many numerical and statistical

aggregation operations, there's things like max and min. There's really dozens of different choices for aggregation operations that you can choose out of this function things here. And then all you do, in this case, let's say we've chosen one called average. So we pass a column name to average and then we pass this average of the column to the aggregate function. And that's how we get this aggregate where we compute the average in a certain column. Oops, the same thing happened that happened earlier in this session. We forgot that this was supposed to be a typed column, so we can't just pass the column name to the average function here. We have to use this as method to convert our untyped regular column into a TypedColumn. So okay we just say that this is double, the parentheses are in the wrong location here. Okay, we do this. And everything is all better now. This average function here which returns a regular untyped column now has been converted into a typed column. And so the column that we're looking at is now a typed column, and we can compute the typed average on that column. So again we keep the type information around. So let's go back to this KeyValueGroupedDataset thing again. Well normally, the operations on these group data sets are aggregations. There are a couple of not really aggregation methods that are also very useful. So one is called mapGroups, which is very similar to the reduce groups that we saw earlier. And what this method does is that it applies the map function that you pass to it to each group of data. So you could imagine going to the collection of the values. And here what map groups does is it basically there's a map on that collection of values, and it returns a new dataset as a result. And we have the same thing for flat maps, so exactly the same idea, just a flattened version. But okay

I just went through a list of transformations and there was one super ridiculously important transformation that we learned about on RDDs, and it was nowhere to be found. So what happened to reduceByKey? Well the short answer is that datasets don't have a reduceByKey method. But we can do a little challenge right now. I just with a number of operations on datasets, I also showed you these grouping operations where you first do things as a group by key and then you have this aggregation operations that you do on the result of the group by key, so we went through all these things. And I argue that with some of that information, it should be pretty easy to emulate the semantics of a reduceByKey transformation on a specific dataset. Maybe this isn't a complete general reduceByKey, but let's say, given this dataset here which is a number of pairs, so we have pairs of type int string, so we have these weird little string snippets and a bunch of integers here is the keys. And your challenge is to try emulate this call here. So, you pretended that these pairs were in the pair RDD, and then you call reduceByKey on them, and you passed this function here, my challenge to you is to compute the same large query result using some of the operations that I've just showed you, but to do it instead on a dataset. I really want you to try this on your own so I'm going to give you a moment to try it yourself. Of course there are several possible solutions but this is the one I've chosen right now. So remember that now, this keyValues thing has type, Dataset [(Int, String)]. And now we call groupByKey on it. And this function that we pass to groupByKey basically tells groupByKey which column or attribute in the dataset should be viewed as the key. So since the elements of this dataset are a pair, and we know that we want to use the number here which appears to be some kind of ID as the key. What we do is, we select the first element of the pair here. And so what this does is it does groupByKey on the integers here. And remember, now we have back one of these strange types, these KeyValue grouped dataset things, right? So it's not a dataset what I got back, and I've go to do something else on it to make it a dataset. So I'm going to use the mapGroup function here, and if you're a call, what this does is, it takes a function from a key and a value to some other type. So this k is an integer, and this vs here is a collection of strings that represent the groups of the values. So what I'm going to do because on an RDD, reduceByKey, when having elements like this, it first groups by the indices and then it reduces on the values. And if you remember, our reduced function and reduceByKey was a simple concatenation, so we were concatenating all these pieces of strings together. So I'd like to do the same thing in this mapGroups function. So I again return a key value pair where I keep the key as it is. And then I just take this collection of values and I do a fold on it to concatenate all the strings together. So

that's how conceptually, with this specific dataset, I can basically emulate what a reduceByKey operation does. And just like on data frames, if I call show on the results, I get now a visualized table here. So here it is, these are my keys and these are my values, and look, there's something interesting happening here. Suddenly, is jumbled stuff makes a little bit of sense. Let's sort that actually another time to make this pit into a better order. So if we sort the records now by ID number where we put 1, 2, and 3 in the right order, then we have a cute little, this is a secret message actually from all of these pieces. So that's how we know we did it right. That's the intended result. So while that might have worked, there's actually a problem with this approach. The only issue is that the API doc gives us a big hint of the danger of using this mapGroups function. So mapGroups does not support partial aggregation, and as a result requires shuffling all of the data in the dataset which is of course extremely expensive and we don't want to do that, right? So if an application intends to perform an aggregation over each key, which is exactly what we're doing, it's actually best to use something else. It's best to use the reduce function or to use this thing called an aggregator. Okay, so mental note. Don't use these mapGroup things unless you really have to. But okay, let's follow the advice of the docs. Let's try doing it with the reduce function. How would you do that? Remember there was a reduced function that we saw in the key value grouped dataset class. In fact, it's called reduced groups. So conceptually, it's a little bit similar to mapGroups because it's focusing on the group to values and instead this is doing a reduce on the group of values. So all we gotta do with the result of our groupByKey is called mapValues. Because reduceGroups has to work on individual strings and not pairs. So mapValues basically goes and takes out all of the strings here and passes them down the line to reduceGroups. Which then concatenates all of those strings. And voila, that works. I could print out the Dataset and it would be exactly the same thing. But the docs also suggested this weird aggregator thing here, what is that thing? So, an aggregator is a class that helps you generically aggregate data. So, you'll notice that some of the aggregation functions that you'll be able to chose from are very specialized. And you don't have the ability to create your own aggregation function. So this aggregation class here is supposed to let you do that. And actually if you think about it, it's kind of like the aggregate method that we saw in RDDs. Because it will give you a couple of functions that will look just like the two functions and the zero that we would pass to the aggregate method on RDDs. First things first, you've gotta know where this thing is. It's inside of spark.sql.expressions, so you're going to have to import this aggregator then to use it. Okay, but what is this aggregator class? The first thing you might notice is that it has three types. A type IN, a type BUF and a type OUT. So if you think about this aggregate method that we saw before in, on RDDs it allowed us to change the types. And to do things in parallel. So we broke up our computation into a couple of functions so it was possible to compute some pieces in parallel and to combine them all later. So this is exactly what this class aggregator is going to do. And in this case, the IN type is actually the input type of the aggregator. So this is would be the type that we got out of the group by key function that we saw earlier. BUF is the intermediate type during aggregation. So if we have to change the type to something else in the middle of this computation, we have this BUF type that we can change it to. And finally, OUT is the type that comes out of the aggregation if we want to change the type. And all of these types can be different. And this is what an aggregator looks like. So what we gotta do is create a new aggregator here. We've gotta define what these types are and then we have to implement these peculiar methods here. So this should be pretty clear what it does. This is the initial value of the aggregation. So this is like the zero that we pass to fold left for example. So this is the initial value. Now reduce and merge, these are like the two functions that we passed to the aggregate method on RDD. Where the reduced method adds an element to the running total and the merge method is what merges these independently computed aggregations together. And finally, this finish method basically gives a chance to change the type one more time before we return the value of the aggregation. So despite how scary it looks, it's actually not so hard once you get the hang of it. Once you get the hang of this structure and you look and you look at things in terms of input, buffer and output types. And you just keep this schema in your mind. So let's try to emulate that reduceByKey

on an earlier slide with one of these aggregators. Let's create our own aggregator. So again we assume the same keyValues data set here and we start by looking at the types in these things. We can break this into steps. The first step is determining what these three parameter types should be to the aggregator. Remember, input, buffer, and output. So, what's the input type? Well, remember, the input type is what came out of the group ByKey method. So, in this case, it would be a pair of int string. So We know that this is going to be a pair of int string. And we that the output type is going to be a string. Since these are pretty simple data types, the buffer value can also be a string. Because we're just going to be concatenating strings together inside of this aggregator. Okay, so first step is complete. Our input type is pair Int, String and then our buffer type is string and output type is string, great. Now we have to feel in the rest of the types. That's completely easy to do, if you go back two slides. And here we have a template of where we can plug in these different types. So, actually everything is going to be a string because BUF and OUT are all strings. And the only thing that's not a string is this input type which is a pair. So we just mechanically plug these types in. Now we have to figure out what these methods should actually each do. The zero or the initial value is pretty straight forward. We're going to be concatenating and building up a strings, so this is definitely going to be an empty string. Okay, cool. So the next method that we have to figure out what to do with is this reduce method here. And actually these types that we've already plugged in give it away. So this is a function where we have to somehow take the string out of this pair and concatenate it with this other string

here. So that's pretty straightforward. We just take B, we pull out the string out of A and we concatenate them together. Now this is a string. And this merge function, since we already have a string we just want to merge together two strings. So, we just do another set of string concatenations. Again, we need to return a string. We just return r without doing anything to it and that's it, there's our aggregator. The last thing that we do is we convert it to a column. So there is this toColumn method here. Because remember, if we're going to parse it to an aggregation method it needs to be of type column. So here we go we parse our aggregator which we named strConcat. We parse it to this aggregation function. And of course we make it a typed column because this is just a regular column, we do as string because the result is of type string. And that's it, voila, we have our aggregator. Oops, expect we don't, we got a compile error. So there was something missing. It might be hard to read but what it says here is that there are two unimplemented members. One called bufferEncoder and one called outputEncoder and they're of type encoder. Okay, what's an encoder? Well, believe it or not we heard about these in passing during the data frame session. Encoders are the things that convert your data between these JVM objects that you're used to operating on and this highly specialized internal tabular representation that Spark SQL operates on. And I mentioned this at the very beginning of the session but these encoder things, these things are required by datasets. So every data set has encoders to go along with it. And these encoder things are extremely specialized, optimized code generators that generate custom bytecode for serialization and deserialization of your data. And they do it all in the special internal tabular representation that we heard about. And this special representation is Sparks internal Tungsten's binary format. Which allows these operations to happen on already serialized data. Which actually greatly improves the memory utilization. So we can put more stuff in memory and we can not have to unpack so much of it basically. Well, that's the context for these things. But since these encoder things are all about serialization you might ask well why isn't Java or Kryo good enough? Well, for a couple of reasons. It starts with the fact that we have a very limited set of data types, which we have encoders define for us. So all the primitives and things that are case classes, all of those Spark SQL core data types. These things that we already understand super, super well we can make sure that there's always a super optimal encoder available for these Core Data types. So that gives us an optimization opportunity. But perhaps most importantly, is the fact that these encoder things actually contain the schema information. So they're not just serializers and deserializers. There's serializers and deserializers that make all of their decisions based on this schema information. So they know in advance, the structure of the data that they are serializing and deserializing. This thing here basically is

what enables Tungsten to do all the optimization that it does. Since Spark only understands the structure of the data that's stored in it's data sets. It could actually create a more optimal layered memory when caching these data sets. So this is kind of the magic that Tungsten does. So this is where we get this great memory utilization from. And that goes without saying that it uses significantly less memory than Kryo or Java. Because we have this extremely compact special format that we're keeping this stuff in in memory, which is a lot smaller than Kryo or Java. And it's actually even ten times faster than Kryo, which is one of the fastest industrial serializers that are around. So these encoder things all tie back to this tungsten component that does all of these optimizations and memory. And with the representation of the data that we're operating on. And I told you that there's an encoder for every dataset. So every Dataset has encoders for its types, but we've been creating Datasets all over the place. And we've never seen these encoder things before. Well that's because the most often used way to introduce an encoder is to do it automatically via implicits from this SparkSession object. But as we saw, well sometimes we run into the situation where we're required to explicitly pass an encoder to something. And for that Spark SQL actually has an object called encoders in the SQL package. Which contains a big selection of methods for creating encoders from Scala primitive types and products and things like that. So here are three kinds of encoders, so there's encoders for INT, LONG, STRING, etc, for primitives that are knowable,right? Which isn't the case for all of Scala's primitives. And then there are methods that create encoders for the Scala primitives. So, scalaInt, sclaLong, scalaByte, etc, the list goes on. And finally, there are methods that create product or tuple encoders, because sometimes we're building up complex types of A case classes. So, here are some examples of how one might go about creating an encoder. So you just call this the name of this package object. Then you invoke this method, for example, scalaInt, and you get back an encoder of type integer, the scala integer. And the same goes for string or for product, for example, as well, if we have a case class type like person here. So that was a quick introduction to encoders. So let's finish up the implementation of this aggregator. All we gotta do is choose the types for these encoders and then look up the right methods to use in the encoder's object. Well, that's pretty simple, because BUF and OUT are both strings, as we saw before. So all we need is to get the string encoders out of this encoder's object and voila, that's it. So I've implemented my encoders. So now all I gotta do is take this custom strConcat Aggregator column thing here, I've gotta make it a typed column. And then I gotta pass it to this aggregator method that we saw before. And that's it, here's the result, just as we expected. So that was a whirlwind tour of actually some of the more advanced ways one could go about grouping and aggregating data. So now you know how to make your own custom aggregators, in case you don't find an aggregator that you need in the Spark API. So I hope this slide is refreshing, so we're changing directions a little bit. We saw transformations and then grouping and aggregating on Datasets. Now let's look at actions on Datasets. And you'll be very happy to notice that these actions are all exactly the same as the actions that we've seen on RDDs and on DataFrames. There's nothing different about them, in fact. Like DataFrames, we have a show method, and all of the other methods like for each or count or collect, they're all the same. So I hope this is reassuring, something super familiar after digging into those aggregator things a few slides back. Everything is as expected with actions, so that's an introduction to Datasets. So the general gist is that they have the same operations available to them that DataFrames have available to them in addition to more of these higher-order functions like math, flat math, and that we liked from RDDs. And, of course, they're also mostly typesafe which is really nice if you've ever had to wrangle with a bunch of exceptions that were thrown by the annualizer when having to deal with DataFrames being untyped. So, at this point I've given you three APIs, three big main different APIs. One is called Datasets, another one is called DataFrames, and then we spent a bunch of time on these things called RDDs. And so, your head must be spinning right now. And you must be like, well, when on Earth would I use one or the other, or the other one of these things? So, I've prepared a simple criteria for when you might be interested in choosing one abstraction over another. So the first thing to remember is that RDDs are really the core abstraction that everything else is built on top of. So this is becoming almost like byte

code to Spark, because they're building all of these richer abstractions on top of these RDD things, and they're generating RDDs. So Datasets and DataFrames, we go through these optimizers, and in the end, we have RDDs that we're actually running. So, we write code in Datasets, and then again, what Spark is running is an RDD, right? So you can think of RDDs as a little bit more low level and totally free form. It's up to you to do your own optimizations on them. And we learned that the hard way in several sessions earlier in the course. That it's super hard, sometimes, to know exactly what these things are doing under the covers. So RDDs are a good choice in the situation that you have data that won't fit for some reason into a Dataset or a DataFrame. Or you have to super fine tune and manage low level details of your RDD computations. You can't just trust the optimizers to do it for you. Or finally, if you have some kind of complex data type that cannot be represented or serialized with these encoder things, which is a real possibility. Encoders can't yet nicely deal with user defined data types. So these are situations when an RDD might be necessary. Otherwise, you may often want to reach for a Dataset or a DataFrames, especially, if you're going to read in some kind of structured data format like JSON or XML. You can just use that wonderful read operation to read in a JSON file and map it to a case class. Datasets do that really well for you. So this is a situation where kind of getting data in suddenly becomes really easy with things like Datasets and even DataFrames. So when you have structured and semi-structured data, it's often really nice to go with a DataFrame or Dataset. DataFrames are generally a good choice when you want to rely as much as possible on these optimization components that Spark provides. If you really want to just write code, not think about what it does, and have something else optimize it for you, and generally trust that it's going to be okay. DataFrames are a good choice, if you can deal with the fact that they're not typed. But, like I said in the very beginning of this session, Datasets are a really nice compromise. Because maybe they're not as performant as these DataFrames here. But they still have good performance, they still have some optimization done on them. So Datasets are a good choice when you want type safety, when you want to work with functional APIs. You have structured / semi-structured data, and good performance is good enough, it doesn't have to be the best. So it's up to you to decide which one. If you prefer based on the data that you have and the sort of computation that you want to do. Perhaps it easier to do it with sort of functional APIs and types. Or maybe it's better to do it totally with relational operations, it really depends on the problem that you're trying to solve. But this is a good set of criteria to help you make that decision. So, before I wrap up the session on Datasets, I wanted to just run through some of the limitations of Datasets that might not have been completely obvious. So, one thing that I eluded to but didn't quite explain is that Datasets don't get all of the optimization the DataFrames get. And they gave you some hints throughout the course about why that's the case. I talked many times about structure and how giving more structure helps optimization. To be a little bit more concrete, relational operations tend to be more performant and able to be easily optimized by the Catalyst core optimizer Because with information about the structure of the data, and the structure of the computations, then the catalyst optimizer knows it can access only the fields that are involved and say, for example, a filter. So if we focus on a filter that look like this, because we have the structure of the data, we have the structure of the computation, we can see into everything. We can see the data. We can see the computation. Then Spark's optimizer can know that it only has to access certain fields involved in that filter and skip over lots of other ones without actually having to instantiate the entire data type. So that's why Catalyst is good at optimizing in this case. Catalyst actually cannot optimize functional filter operations or functional operations that include passing around function literals or lambdas. And the reason why is because when you pass a function to Spark, all Spark can know when you passed it this function BLOB is that you need an entire record serialized, so an entire object serialized, and you need to apply this function to it you don't what's in it. So you need to create an object and then apply this function to this thing and you don't know what's inside either of these two things which requires Spark to do potentially more work in it than it has to in order to do the baseline minimum that it should have to do, because Spark just can't see into lambda functions. It has no idea what they're doing. So Catalyst can't do very much for operations that involve

passing around these function literals. And of course you don't miss out on every possible optimization. You just miss out on some important ones. So the key take away here is that, when you use datasets with high order functions like map, you end up missing out on many optimizations that Catalyst does for data frames. So that's something to keep in mind. This is why sometimes datasets are slower than data frames. However, it doesn't mean that every operation on a data frame doesn't get these wonderful optimizations from Catalyst. Or if you choose relational operations like select, or the relational filters, or any of these operations then you get all of Catalyst's optimizations on datasets. So it really depends on the operations that you're doing on datasets, whether or not you get the full sort of suite of optimizations that are available. Finally, that's not to say that datasets that use things like map operations don't get any optimizations. They still have Tungsten always running under the hood, storing and organizing data in highly efficient and optimized ways, which of course can still result in large speedups over regular RDDs, so that's not nothing. So you basically just lose out a little bit out on Catalyst if you use high-order functions, but you still get all the benefits of Tungsten. So I hope that gives you just a little bit of intuition about the capabilities that Spark has to optimize operations on datasets. And the two last limitations which are also shared by data frames is the fact that the number of data types are limited, so you end up having to express all of your data with case classes, and these standard Spark SQL data types, these primitive data types. If you can't do that, then it might be difficult to ensure that there is a Tungsten encoder that exists for you. So, this could be also be a problem. You can't just put any data type ever into Spark. And finally there is that point about semi-structured and structured data. Sometimes there are just datasets that are completely unstructured, they do have no schema, they're not self describing, there is no way to sort of programmatically guess the structure of these datasets. And in those cases datasets might not be a good thing because it might be really difficult to shuffle that data around and put it into some format that's comfortable for datasets to deal with. It might be easier to just parse the bits and pieces that you need in with regular Spark RDDs.

https://www.coursera.org/learn/scala-spark-big-data#syllabushttps://www.coursera.org/learn/scala-spark-big-data#syllabus