ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Datasets

Big Data Analysis with Scala and Spark

Heather Miller

# 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")
```

# 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...

# Example

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

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

# Example

```scala
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?

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

# Example

```scala
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?

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

Nope.

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

# 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)
```

# 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...

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

**yay!** 🎉

# 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?**

**Enter Datasets.**

# Confession

I've been keeping something from you...

# DataFrames are Datasets!

**`DataFrames` are actually `Datasets`.**

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

# DataFrames are Datasets!

**`DataFrames` are actually `Datasets`.**

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

😳 **What the heck is a `Dataset`?**

# 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 structured/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.

# DataFrames are Datasets!

**Example:**

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

$\lambda$

```
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!

# 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

**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.

# 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]
```

# 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._
```

# 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._
```

# 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
                         ^
```

# 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.
```

# Transformations on Datasets

Remember *untyped transformations* from `DataFrame`s?

# Transformations on Datasets

Remember *untyped transformations* from `DataFrame`s?

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

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

# Transformations on Datasets

Remember *untyped transformations* from `DataFrame`s?

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

- **untyped transformations** the transformations we learned on `DataFrame`s.
- **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.

```scala
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...
```

# Common (Typed) Transformations on Datasets

| | |
|---|---|
| **`map`** | **`map[U](f: T => U): Dataset[U]`**<br>Apply function to each element in the Dataset and return a Dataset of the result. |
| **`flatMap`** | **`flatMap[U](f: T => TraversableOnce[U]): Dataset[U]`**<br>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]`**<br>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]`**<br>Return Dataset with duplicates removed. |

# Common (Typed) Transformations on Datasets

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

# 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`
- `KeyValueGroupedDataset`contains a number of aggregation operations which return `Datasets`

# 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`
- `KeyValueGroupedDataset`contains 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.*

# Some `KeyValueGroupedDataset` Aggregation Operations

| | |
|---|---|
| **reduceGroups** | **`reduceGroups(f: (V, V) => V): Dataset[(K, V)]`**<br>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)]`**<br>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. |

# Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation agg defined on KeyValueGroupedDataset.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

**The only thing a bit peculiar about this operation is its argument. What do we pass to it?**

# Using the General agg Operation

Just like on `DataFrames`, there exists a general aggregation operation `agg` defined on `KeyValueGroupedDataset`.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

**The only thing a bit peculiar about this operation is its argument. What do we pass to it?**

Typically, we simply select one of these operations from `function`, such as `avg`, choose a column for `avg` to be computed on, and we pass it to `agg`.

```
someDS.agg(avg($"column"))
```

# Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation agg defined on KeyValueGroupedDataset.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

Typically, we simply select one of these operations from function, such as avg, choose a column for avg to be computed on, and we pass it to agg.

```
someDS.agg(avg($"column"))
// [error]  found   : org.apache.spark.sql.Column
// [error]  required: org.apache.spark.sql.TypedColumn[Listing,?]
// [error]                        .agg(avg($"column"))
// [error]                                ^
// [error] one error found
```

**Oops. TypedColumn! Remember that we have to use as[...] to convert our untyped regular Column into a TypedColumn.**

# Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation agg defined on KeyValueGroupedDataset.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

Typically, we simply select one of these operations from function, such as avg, choose a column for avg to be computed on, and we pass it to agg.

```
someDS.agg(avg($"column").as[Double])
```

All better now.

# Some KeyValueGroupedDataset (Aggregation) Operations

**mapGroups** `mapGroups[U](f: (K, Iterator[V]) => U): Dataset[U]`
Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an element of arbitrary type which will be returned as a new Dataset.

**flatMapGroups** `flatMapGroups[U](f: (K, Iterator[V]) => TraversableOnce[U]): Dataset[U]`
Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an iterator containing elements of an arbitrary type which will be returned as a new Dataset.

*Note: at the time of writing,* KeyValueGroupedDataset *is marked as @Experimental and @Evolving. Therefore, expect this API to fluctuate–it's likely that new aggregation operations will be added and others could be changed.*

# reduceByKey?

If you glance around the `Dataset` API docs, you might notice that `Datasets` are missing an important transformation that we often used on `RDDs`: **`reduceByKey`**.

# reduceByKey?

If you glance around the Dataset API docs, you might notice that Datasets are missing an important transformation that we often used on RDDs: **reduceByKey**.

**Challenge:**
Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))
```

Find a way to use Datasets to achieve the same result that you would get if you put this data into an RDD and called:

```
keyValuesRDD.reduceByKey(_+_)
```

**Try it on your own now!**

*Note: the objective is just to use the APIs presented so far, don't worry about performance for now.*

# reduceByKey?

**Challenge:**
Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

Dataset[(Int, String)]

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))
```

_ + _

# reduceByKey?

**Challenge:**

Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2))).show()
```

```
+---+----------+
| _1|        _2|
+---+----------+
|  1|   ThisIsA|
|  3|Message:-)|
|  2|    Secret|
+---+----------+
```

# reduceByKey?

**Challenge:**

Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2))).show()
```

```
+---+----------+
| _1|        _2|
+---+----------+
|  1|   ThisIsA|
|  3|Message:-)|
|  2|    Secret|
+---+----------+
```

**Let's sort the records by id number! :-)**

# reduceByKey?

**Challenge:**

Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))
           .sort($"_1").show()

   +---+----------+
   | _1|        _2|
   +---+----------+
   |  1|   ThisIsA|
   |  2|    Secret|
   |  3|Message:-)|
   +---+----------+
```

# reduceByKey?

**Challenge:**

Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))
```

**The only issue with this approach is this disclaimer in the API docs for `mapGroups`:**

> *This function does not support partial aggregation, and as a result requires shuffling all the data in the Dataset. If an application intends to perform an aggregation over each key, it is best to use the reduce function or an org.apache.spark.sql.expressions#Aggregator.*

# reduceByKey?

**Challenge:**

Emulate the semantics of `reduceByKey` on a `Dataset` using `Dataset` operations presented so far. Assume we'd have the following data set:

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

keyValuesDS.groupByKey(p => p._1)
           .mapValues(p => p._2)
           .reduceGroups((acc, str) => acc + str)
```

**That works! But the docs also suggested an `Aggregator`?**

# Aggregators

A class that helps you generically aggregate data. Kind of like the `aggregate` method we saw on RDDs.

```
class Aggregator[-IN, BUF, OUT]
```

org.apache.spark.sql.expressions.Aggregator

- **IN** is the input type to the aggregator. When using an aggregator after `groupByKey`, this is the type that represents the value in the key/value pair.
- **BUF** is the intermediate type during aggregation.
- **OUT** is the type of the output of the aggregation.

# Aggregators

A class that helps you generically aggregate data. Kind of like the `aggregate` method we saw on RDDs.

```
class Aggregator[-IN, BUF, OUT]
```

- `IN` is the input type to the aggregator. When using an aggregator after `groupByKey`, this is the type that represents the value in the key/value pair.
- `BUF` is the intermediate type during aggregation.
- `OUT` is the type of the output of the aggregation.

**This is how implement our own `Aggregator`:**

```
val myAgg = new Aggregator[IN, BUF, OUT] {
  def zero: BUF = ...                    // The initial value.
  def reduce(b: BUF, a: IN): BUF = ...   // Add an element to the running total
  def merge(b1: BUF, b2: BUF): BUF = ... // Merge intermediate values.
  def finish(b: BUF): OUT = ...          // Return the final result.
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate `reduceByKey` on a specific data set, and let's see if we can implement the aggregation part of our `reduceByKey` operation with an `Aggregator`.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[?, ?, ?]{  // Step 1: what should Aggregator's
  def zero: ? = ???                       // type parameters be?
  def reduce(b: ?, a: ?): ? = ???
  def merge(b1: ?, b2: ?): ? = ???
  def finish(r: ?): ? = ???
}.toColumn
```

(Int, String)

String

String

# Emulating `reduceByKey` with an `Aggregator`

Let's return to our example of trying to emulate `reduceByKey` on a specific data set, and let's see if we can implement the aggregation part of our `reduceByKey` operation with an `Aggregator`.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: ? = ???
  def reduce(b: ?, a: ?): ? = ???      // Step 2: what should the rest of
  def merge(b1: ?, b2: ?): ? = ???     // types be?
  def finish(r: ?): ? = ???
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ???
  def reduce(b: String, a: (Int, String)): String = ???        // Step 3: implement the
  def merge(b1: String, b2: String): String = ???              // methods!
  def finish(r: String): String = ???
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = ???        // Step 3: implement the
  def merge(b1: String, b2: String): String = ???              // methods!
  def finish(r: String): String = ???
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = b + a._2   // Step 3: implement the
  def merge(b1: String, b2: String): String = ???              // methods!
  def finish(r: String): String = ???
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
}.toColumn
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```scala
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
}.toColumn                                         // Step 4: pass it to your aggregator!

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String])
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String = ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
}.toColumn

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String])
```

```
[error] object creation impossible, since: it has 2 unimplemented members.
[error] the missing signatures are as follows.
[error]   def bufferEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   def outputEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   val strConcat = new Aggregator[(Int, String), String, String]{
[error]                         ^
[error] one error found
```

**Oops! We're missing 2 methods implementations. What's an Encoder?**

# Encoders

`Encoders` are what convert your data between JVM objects and Spark SQL's specialized internal (tabular) representation. **They're required by all `Datasets`!**

Encoders are highly specialized, optimized code generators that generate custom bytecode for serialization and deserialization of your data.

The serialized data is stored using Spark internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization.

## What sets them apart from regular Java or Kryo serialization:

- Limited to and optimal for primitives and case classes, Spark SQL data types, which are well-understood.
- **They contain schema information**, which makes these highly optimized code generators possible, and enables optimization based on the shape of the data. Since Spark understands the structure of data in Datasets, it can create a more optimal layout in memory when caching Datasets.
- Uses significantly less memory than Kryo/Java serialization
- >10x faster than Kryo serialization (Java serialization orders of magnitude slower)

# Encoders

`Encoders` are what convert your data between JVM objects and Spark SQL's specialized internal representation. **They're required by all `Datasets`!**

**Two ways to introduce encoders:**

- **Automatically** (generally the case) via implicits from a `SparkSession`. `import spark.implicits._`
- **Explicitly** via `org.apache.spark.sql.Encoders` which contains a large selection of methods for creating `Encoders` from Scala primitive types and `Products`.

# Encoders

`Encoders` are what convert your data between JVM objects and Spark SQL's specialized internal representation. **They're required by all `Datasets`!**

## Two ways to introduce encoders:

- **Automatically** (generally the case) via implicits from a `SparkSession`. `import spark.implicits._`
- **Explicitly** via `org.apache.spark.sql.Encoder`, which contains a large selection of methods for creating `Encoders` from Scala primitive types and `Products`.

## Some examples of 'Encoder' creation methods in 'Encoders':

- `INT`/`LONG`/`STRING` etc, for *nullable* primitives.
- `scalaInt`/`scalaLong`/`scalaByte` etc, for Scala's primitives.
- `product`/`tuple` for Scala's `Product` and tuple types.

**Example:** Explicitly creating `Encoders`.

```
Encoders.scalaInt // Encoder[Int]
Encoders.STRING // Encoder[String]
Encoders.product[Person] // Encoder[Person], where Person extends Product/is a case class
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String= ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
  override def bufferEncoder: Encoder[BUF] = ???     // Step 4: Tell Spark which
  override def outputEncoder: Encoder[OUT] = ???     // Encoders you need.
}.toColumn

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String])
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String= ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r
  override def bufferEncoder: Encoder[String] = Encoders.STRING
  override def outputEncoder: Encoder[String] = Encoders.STRING
}.toColumn

keyValuesDS.groupByKey(pair => pair._1)
           .agg(strConcat.as[String])
```

# Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)"),(2,"cre"),(2,"t"))

val keyValuesDS = keyValues.toDS

val strConcat = new Aggregator[(Int, String), String, String]{
  def zero: String= ""
  def reduce(b: String, a: (Int, String)): String = b + a._2
  def merge(b1: String, b2: String): String = b1 + b2
  def finish(r: String): String = r                                 // +-----+--------------------+
  override def bufferEncoder: Encoder[String] = Encoders.STRING      // |value|anon$1(scala.Tuple2)|
  override def outputEncoder: Encoder[String] = Encoders.STRING      // +-----+--------------------+
}.toColumn                                                           // |    1|             ThisIsA|
                                                                     // |    3|          Message:-)|
keyValuesDS.groupByKey(pair => pair._1)                              // |    2|              Secret|
           .agg(strConcat.as[String]).show                           // +-----+--------------------+
```

# Common Dataset Actions

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

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

`first(): T/head(): T`
Returns the first row in this Dataset.

`foreach(f: T => Unit): Unit`
Applies a function f to all rows.

`reduce(f: (T, T) => T): T`
Reduces the elements of this Dataset using the specified binary function.

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

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

# When to use Datasets vs DataFrames vs RDDs?

**Use Datasets when...**

- you have structured/semi-structured data
- you want typesafety
- you need to work with functional APIs
- you need good performance, but it doesn't have to be the best

**Use DataFrames when...**

- you have structured/semi-structured data
- you want the best possible performance, automatically optimized for you

**Use RDDs when...**

- you have unstructured data
- you need to fine-tune and manage low-level details of RDD computations
- you have complex data types that cannot be serialized with `Encoders`

# Limitations of Datasets

## Catalyst Can't Optimize All Operations

Take filtering as an example.

**Relational filter operation** E.g., `ds.filter($"city".as[String] === "Boston")`.
Performs best because you're explicitly telling Spark which columns/attributes and conditions are required in your filter operation. With information about the structure of the data and the structure of computations, Spark's optimizer knows it can access only the fields involved in the filter without having to instantiate the entire data type. Avoids data moving over the network.
**Catalyst optimizes this case.**

**Functional filter operation** E.g., `ds.filter(p => p.city == "Boston")`.
Same filter written with a function literal is opaque to Spark – it's impossible for Spark to introspect the lambda function. All Spark knows is that you need a (whole) record marshaled as a Scala object in order to return true or false, requiring Spark to do porentially a lot more work to meet that implicit requirement.
**Catalyst cannot optimize this case.**

# Limitations of Datasets

## Catalyst Can't Optimize All Operations

**Takeaways:**

- When using `Datasets` with higher-order functions like `map`, you miss out on many Catalyst optimizations.
- When using `Datasets` with relational operations like `select`, you get all of Catalyst's optimizations.
- Though not all operations on `Datasets` benefit from Catalyst's optimizations, Tungsten is still always running under the hood of `Datasets`, storing and organizing data in a highly optimized way, which can result in large speedups over RDDs.

# Limitations of Datasets

## Limited Data Types

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

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

# Limitations of Datasets

## 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.