<http://loicdescotte.github.io/posts/spark2-datasets-type-safety/>

[Spark Datasets and type-safety](http://loicdescotte.github.io/posts/spark2-datasets-type-safety/)

January 22, 2017

[Spark](http://spark.apache.org/) 2.0 has introduced the Datasets API (in a stable version). Datasets promise is to add type-safety to dataframes, that are a more SQL oriented API. I used to rely on the lower level RDD API (distributed Spark collections) on some parts of my code when I wanted more type-safety but it lacks some of the dataframe optimizations (for example on groupBy and aggregations operations). The recommended way is now to use datasets everywhere (except if you’re doing something very specific and if you need to use the low level RDD funcstions). Let’s see how it looks.

This is the classical word count using RDD :

val textFile = sc.textFile("hdfs://...")

val counts = textFile.flatMap(line => line.split(" "))

.map(word => (word.toLowerCase, **1**)) // put 1 with each word instance

.reduceByKey((accumulator, current) => accumulator+current) // add all words, grouped by value (by key)

And the word count with dataframes :

val df = sparkSession.read.text("hdfs://...")

val wordsDF = df.select(split(df("value")," ").alias("words"))

val wordDF = wordsDF.select(explode(wordsDF("words")).alias("word"))

val wordCount = wordDF.groupBy(lower($"word")).count

The drawback is that you loose the type information and the field names, so you need to use columns names as strings, which can be error-prone. Also, my personal opinion for this kind of example is that map/flatMap operations are easier to read.

With datasets, you have a mix of RDD and dataframes, with an high level API while retaining some type information :

sparkSession.read.text("hdfs://...").as[**String**]

.flatMap(\_.split(" "))

.groupByKey(\_.toLowerCase)

.count()

This version is the easiest to read and to understand. So, it seems perfect! But… if you try to do some more complex operations on rows, you will quick fallback on the non type-safe API.

For example, if you want to order the words by count :

sparkSession.read.text("hdfs://...").as[**String**]

.flatMap(\_.split(" "))

.groupByKey(\_.toLowerCase)

.count()

.orderBy($"count(1)".desc) // WTF

Note : The count column name is pretty weird, but it gets a little better if you use groupBy (which is not type-safe) instead of groupByKey :

sparkSession.read.text("hdfs://...").as[**String**]

.flatMap(\_.split(" "))

.map(\_.toLowerCase)

.groupBy($"value") // value is the default column name

.count()

.orderBy($"count".desc)

You will also have this kind of problem if you want to join 2 datasets :

//dataset creation from a Seq

val departments = **Seq**(

**Department**(**1**, "rh"),

**Department**(**2**, "it"),

**Department**(**3**, "marketing")

).toDS

val people = **Seq**(

**Person**("jane", **28**, "female", **2000**, **2**),

**Person**("bob", **31**, "male", **2000**, **1**),

//...

).toDS

people.joinWith(departments, people("deptId") === departments("id"))

Here again you have to pass the column names as strings. To be honest, it’s still easier than the RDD equivalent, where you would have to create (key, value) pair RDDs to be able to join data :

val departmentsById = departments.rdd.map{ department =>

(department.id, department)

}

val peopleByDepartmentId = people.rdd.map{ person =>

(person.deptId, person)

}

peopleByDepartmentId.join(departmentsById)

Let’s go back to the main subject of this post : datasets. With more complex operations, type-safety is still far from perfect :

people.filter(\_.age > **30**)

.join(departments, people("deptId") === departments("id"))

.group(departments("name"), $"gender")

.agg(avg(people("salary")), max(people("age"))) // average salary and max age

Yes, that can be disappointing… but, there is a library named [Frameless](https://github.com/adelbertc/frameless), based on the awesome Shapeless, that can add more type-safety to datasets!

import **frameless.TypedDataset**

val departments = **TypedDataset**.create(**Seq**(

**Department**(**1**, "rh"),

**Department**(**2**, "it"),

**Department**(**3**, "marketing")

))

val people = **TypedDataset**.create(**Seq**(

**Person**("jane", **28**, "female", **2000**, **2**),

**Person**("bob", **31**, "male", **2000**, **1**),

//...

))

val joined = people.join(departments, people('deptId), departments('id))

// val joined = people.join(departments, people('detppid), departments('id)) <-- Won't compile as 'detppid symbol is wrong

As you can see on the last line, if you provide a bad column name, you will get a compilation error. Pretty great isn’t it?

Another problem with regular datasets is that they can produce null values, for example using left outer joins :

departments.joinWith(people, departments("id") === people("deptId"), "left\_outer").show

The output is :

// +-------------+--------------------+

// | \_1| \_2|

// +-------------+--------------------+

// | [1,rh]|[linda,37,female,...|

// | [1,rh]|[john,45,male,300...|

// | [1,rh]|[bob,35,male,2200,1]|

// | [1,rh]|[bob,31,male,2000,1]|

// | [2,it]|[joe,40,male,3000,2]|

// | [2,it]|[jane,28,female,2...|

// |[3,marketing]| null| <--- WARNING : null

// +-------------+--------------------+

//

With Frameless, you won’t have this problem as it gives you Options :

val leftJoined: **TypedDataset**[(**Department**, **Option**[**Person**])] = departments.joinLeft(people, departments('id), people('deptId))

Frameless is still described as a “proof of concept” and does not cover all dataset operations, but I think it’s an interesting library to follow!

# Type safety and Spark Datasets in Scala

<https://codeburst.io/type-safety-and-spark-datasets-in-scala-20fa582024fc>

<https://gist.github.com/manishkkatoch>

W**orking with Spark Datasets**have been quite interesting and most of the time rewarding in our current project. It has a simple yet powerful API that abstracts out the need to code in complex transformations and computations. To be honest, we also have a fairly straightforward use case: few domain entities, fewer transformations based on simple joins.

However, there are also few things that have been counterproductive to us but I am going to focus on one of them: lack of type safety in some operations, particularly, joins.

dataSetA.join(dataSetB, "columnA")

The above code will fail on runtime if either of dataSetA and dataSetB (or both) don’t have “columnA” column. This is a waste of resources at multiple levels: from precious CPU cycles to developer’s time. In the remainder of this blog, we will add compile-time safety to join operations and learn a lot in the process.

***Before we proceed****, a disclaimer:*This is not an unsolved problem*.*[*Frameless*](https://github.com/typelevel/frameless)*does a fantastic job at providing the type-safety for Datasets. However, it is a very evolved and complete framework which provides a newer abstraction of TypedDatasets and we really did not want to add an external dependency when we just wanted to have type safety in our select Dataset methods. The solution we are going to formulate is what Frameless does which inturn leverages on generic programming using awesome*[*Shapeless*](https://github.com/milessabin/shapeless)*.*

# Problem Statement

Let’s come up with goals we want to achieve at the end of this post:

1. When we access a column by name, the compilation should fail if the column does not exist in the dataset.
2. When we join two datasets, the compilation should fail if the joining column is not part of either one of the dataset or if present, not of the same type.
3. Some good DSL for doing above never hurts!

# Step 0: Basics

For any Dataset of type T (case class/Product type), we need to understand all the properties of type T along with their types. This means that we want to move from a specialized T to generalized list of properties with types. and this, in a very very simplified way of explanation, is what Shapeless provides. It provides a conversion to and from a **case class**and a**heterogeneous list (HList)**and a bouquet of functions to apply on the list. The best material to read about shapeless is [this](https://books.underscore.io/shapeless-guide/shapeless-guide.html) and I strongly suggest to give it a thorough read.

For now, we can do with a knowledge that shapeless provides an interface LabelledGeneric which provides the interface.

This can be explained as below

case class Person(name: String, age: Int, isEmployee: Boolean)  
//defined class Persongeneric = LabelledGeneric[Person]  
//generic: shapeless.LabelledGeneric[Person]{type Repr = shapeless.::[String with shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String("name")],String],shapeless.::[Int with shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String("age")],Int],shapeless.::[Boolean with shapeless.labelled.KeyTag[Symbol with shapeless.tag.Tagged[String("isEmployee")],Boolean],shapeless.HNil]]]}//usage:  
val person = Person("John Doe", 32, true)val hlist = generic.to(person)  
//hlist: generic.Repr = John Doe :: 32 :: true :: HNilHNilgeneric.from(hlist)  
//res0: Person = Person(John Doe,32,true)

# Step 1: Property Exists?

*Given a type T, if there exists a property of name PName and type PType then yes, the conditions are satisfied*

Let’s break down the gist line by line:

1. We define a trait PropertyExists for type T which also expects types PName ( for Property Name) and PType ( for Property Type), we don’t worry about the properties/methods of the trait as the existence of such instance is truthfulness of our condition.
2. We define an apply method which accepts a Witness and implicitly expects an instance of PropertyExists for a certain PType. Witness is one of the utility abstractions of Shapeless which given a Symbol returns handle to its type and value.
3. But how to do we pass the implicit parameter of PropertyExists? Also, where are we looking for the properties? well, the implicit is provided by implicitProvider which rely on LabelledGeneric that we introduced above. It takes a couple of more implicitly created parameters. Let’s dissect them:

implicit gen: LabelledGeneric.Aux[T, H]

gen provides the heterogenous list (HList) representation of type T. It uses the [Aux pattern](http://gigiigig.github.io/posts/2015/09/13/aux-pattern.html) (another must read for type-level programming!) to forward the result type to the next implicit parameter creation

selector: Selector.Aux[H, PName, PType]

The Selector is one of the simpler abstraction of Shapeless which provides the PType given it finds the propertyName PName in record H.

So in simpler terms, the implicitProvider talks the following:

*For a given****type******T****, if you are able to create a****HList****of type****H****from****LabelledGeneric[T]****and then if you are able to also****select PType****from that HList****H****a property of name****PName****, then go ahead and provide a****PropertyExists****instance for type****T, PName and PType****.*

# Step 2: First Test of Type Safety

Now that we have our PropertyExists, let's have our first stab at type safety: Creating a Column instance from a key and failing on compile time if it doesn’t exist.

We define a RichDataset abstraction which extends spark Dataset to provide the functionality of type checking.

We add an apply method which takes a Symbol and implicitly tries to get a PropertyExists instance for the column type column.T (Aux pattern at play here too!). Like always this will compile only if the column exists in A.

If we take our above case class Person, the following behaviour should be observed:

personDs = Seq(persons).toDS().enrichedval ageColumn: Column = personDs('age) //compiles  
val nameColumn: Column = personDs('namesss)  
//Error:(36, 56) Symbol with shapeless.tag.Tagged[String("namesss")] not found in Person  
 val nameColumn: Column = personDs('namesss)

and that is our first milestone!

*PS: we need to expose enriched as the compile will pick apply method of Dataset and not that of RichDataset.*

# Step 3: Let’s Join

Now that we have established the usage of PropertyExists lets try to formulate a DSL we would want to use for carrying out our joins

//for left join//natural join single key reference  
datasetA.leftJoin(datasetB).withKey('key)//natural join multiple keys  
datasetA.leftJoin(datasetB).on('key1, 'key2)//for joins not natural.  
datasetA.leftJoin(datasetB) where {   
 datasetA('keyA) === datasetB('keyB)  
}

seems pretty ok. Let’s dive in!

We introduce a JoinDataSet which provides the syntactical sugar to facilitate the actual join operations. JoinDataSet will also provide us with the final methods of actual join as decided in DSL: withKey, on and where.

## .withKey

As we can see, withKey is identical to what we achieved in our step 2 with a couple of notable differences.

*for a Symbol*column, *we check if PropertyExists for both Dataset[L] and Dataset[R] and also for both datasets the type is K.*

This enforces that not only column name should be the same, but also their type.

## .where

.where is even simpler. It takes a nullary function which returns a Column and leverages on the way we express conditions on Column. To express Column we use the apply method we created

## .on

As one can observe .on is not a function at all! If we think on this and our definition of on method in the DSL, what we need to work on is the varargs of Symbol and for each such symbol have a PropertyExists created. Unfortunately, there is no way to convert a varargs to HList as varargs are Seq and Seq is not Product (case class type). For this Shapeless has provided a sugar abstraction SingletonProductArgs which uses dynamic programming to create an HList. the applyProduct is really an apply method on “on” object and allows us to achieve our syntax.

Here’s how the above code pans out:

*Given I have****varargs****to dynamically apply to a method named “****apply****” which gives out an****HList V****, and I can generate an implicit instance which gives****List of Symbols****out of it, and also for both the****Datasets****,****PropertiesExists****of****type K****in the****HList V****: do the Join.*

Heres the complete code:

## PropertiesExists?

For matching multiple properties, we create another trait like PropertyExists. While PropertyExists worked with single property PName, the PropertiesExists needs to work with HList. So we get our trait as:

trait PropertiesExists[T, PName <: HList, PType]

Now, like a List, HList also has 3 basic building blocks: Head, Tail and Nil (in this case HNil) where:

HList = Head :: Tail :: HNil

So all we need to do now is define implicitProviders for HNil, Tail and Head. Since the head is essentially a single Property, PropertyExists fits just fine! for the tail, we recursively try to create an implicit provider as we do for any List.

we can complete our RichDataSet as below:

# ****And That’s it!****

Pursuing type safety goes a long way in optimizing development flow, catching early issues (even before execution!) and most importantly helps writing meaningful unit tests. Apart from the type safety, I also wanted to share how Shapeless (and really generic type-level programming) can aid in writing succinct, compile-time and type-safe code and I hope I was able to do some justice to how awesome Shapeless and Frameless (for Spark Dataset) are!

## Thank you for reading! If you liked this article, please share/recommend. If not, please comment/critique so I can improve and learn more! :)

# Comparing TypedDatasets with Spark's Datasets

<https://typelevel.org/frameless/TypedDatasetVsSparkDataset.html>

**Goal:** This tutorial compares the standard Spark Datasets API with the one provided by Frameless' TypedDataset. It shows how TypedDatasets allow for an expressive and type-safe api with no compromises on performance.

For this tutorial we first create a simple dataset and save it on disk as a parquet file. [Parquet](https://parquet.apache.org/) is a popular columnar format and well supported by Spark. It's important to note that when operating on parquet datasets, Spark knows that each column is stored separately, so if we only need a subset of the columns Spark will optimize for this and avoid reading the entire dataset. This is a rather simplistic view of how Spark and parquet work together but it will serve us well for the context of this discussion.

import spark.implicits.\_

// import spark.implicits.\_

// Our example case class Foo acting here as a schema

case class Foo(i: Long, j: String)

// defined class Foo

// Assuming spark is loaded and SparkSession is bind to spark

val initialDs = spark.createDataset( Foo(1, "Q") :: Foo(10, "W") :: Foo(100, "E") :: Nil )

// initialDs: org.apache.spark.sql.Dataset[Foo] = [i: bigint, j: string]

// Assuming you are on Linux or Mac OS

initialDs.write.parquet("/tmp/foo")

val ds = spark.read.parquet("/tmp/foo").as[Foo]

// ds: org.apache.spark.sql.Dataset[Foo] = [i: bigint, j: string]

ds.show()

// +---+---+

// | i| j|

// +---+---+

// | 1| Q|

// | 10| W|

// |100| E|

// +---+---+

//

The value ds holds the content of the initialDs read from a parquet file. Let's try to only use field i from Foo and see how Spark's Catalyst (the query optimizer) optimizes this.

// Using a standard Spark TypedColumn in select()

val filteredDs = ds.filter($"i" === 10).select($"i".as[Long])

// filteredDs: org.apache.spark.sql.Dataset[Long] = [i: bigint]

filteredDs.show()

// +---+

// | i|

// +---+

// | 10|

// +---+

//

The filteredDs is of type Dataset[Long]. Since we only access field i from Foo the type is correct. Unfortunately, this syntax requires handholding by explicitly setting the TypedColumn in the select statement to return type Long (look at the as[Long] statement). We will discuss this limitation next in more detail. Now, let's take a quick look at the optimized Physical Plan that Spark's Catalyst generated.

filteredDs.explain()

// == Physical Plan ==

// \*Project [i#1771L]

// +- \*Filter (isnotnull(i#1771L) && (i#1771L = 10))

// +- \*FileScan parquet [i#1771L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/tmp/foo], PartitionFilters: [], PushedFilters: [IsNotNull(i), EqualTo(i,10)], ReadSchema: struct<i:bigint>

The last line is very important (see ReadSchema). The schema read from the parquet file only required reading column i without needing to access column j. This is great! We have both an optimized query plan and type-safety!

Unfortunately, this syntax is not bulletproof: it fails at run-time if we try to access a non existing column x:

scala> ds.filter($"i" === 10).select($"x".as[Long])

org.apache.spark.sql.AnalysisException: cannot resolve '`x`' given input columns: [i, j];;

'Project ['x]

+- Filter (i#1771L = cast(10 as bigint))

+- Relation[i#1771L,j#1772] parquet

at org.apache.spark.sql.catalyst.analysis.package$AnalysisErrorAt.failAnalysis(package.scala:42)

at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$2.applyOrElse(CheckAnalysis.scala:88)

at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$2.applyOrElse(CheckAnalysis.scala:85)

at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode.scala:289)

at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$transformUp$1.apply(TreeNode.scala:289)

at org.apache.spark.sql.catalyst.trees.CurrentOrigin$.withOrigin(TreeNode.scala:70)

at org.apache.spark.sql.catalyst.trees.TreeNode.transformUp(TreeNode.scala:288)

at org.apache.spark.sql.catalyst.plans.QueryPlan$$anonfun$transformExpressionsUp$1.apply(QueryPlan.scala:268)

at org.apache.spark.sql.catalyst.plans.QueryPlan$$anonfun$transformExpressionsUp$1.apply(QueryPlan.scala:268)

at org.apache.spark.sql.catalyst.plans.QueryPlan.transformExpression$1(QueryPlan.scala:279)

at org.apache.spark.sql.catalyst.plans.QueryPlan.org$apache$spark$sql$catalyst$plans$QueryPlan$$recursiveTransform$1(QueryPlan.scala:289)

at org.apache.spark.sql.catalyst.plans.QueryPlan$$anonfun$org$apache$spark$sql$catalyst$plans$QueryPlan$$recursiveTransform$1$1.apply(QueryPlan.scala:293)

at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)

at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)

at scala.collection.immutable.List.foreach(List.scala:392)

at scala.collection.TraversableLike$class.map(TraversableLike.scala:234)

at scala.collection.immutable.List.map(List.scala:296)

at org.apache.spark.sql.catalyst.plans.QueryPlan.org$apache$spark$sql$catalyst$plans$QueryPlan$$recursiveTransform$1(QueryPlan.scala:293)

at org.apache.spark.sql.catalyst.plans.QueryPlan$$anonfun$6.apply(QueryPlan.scala:298)

at org.apache.spark.sql.catalyst.trees.TreeNode.mapProductIterator(TreeNode.scala:187)

at org.apache.spark.sql.catalyst.plans.QueryPlan.mapExpressions(QueryPlan.scala:298)

at org.apache.spark.sql.catalyst.plans.QueryPlan.transformExpressionsUp(QueryPlan.scala:268)

at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.apply(CheckAnalysis.scala:85)

at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.apply(CheckAnalysis.scala:78)

at org.apache.spark.sql.catalyst.trees.TreeNode.foreachUp(TreeNode.scala:127)

at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$class.checkAnalysis(CheckAnalysis.scala:78)

at org.apache.spark.sql.catalyst.analysis.Analyzer.checkAnalysis(Analyzer.scala:91)

at org.apache.spark.sql.execution.QueryExecution.assertAnalyzed(QueryExecution.scala:52)

at org.apache.spark.sql.Dataset.<init>(Dataset.scala:165)

at org.apache.spark.sql.Dataset.<init>(Dataset.scala:171)

at org.apache.spark.sql.Dataset.select(Dataset.scala:1210)

... 454 elided

There are two things to improve here. First, we would want to avoid the as[Long] casting that we are required to type for type-safety. This is clearly an area where we may introduce a bug by casting to an incompatible type. Second, we want a solution where reference to a non existing column name fails at compilation time. The standard Spark Dataset can achieve this using the following syntax.

ds.filter(\_.i == 10).map(\_.i).show()

// +-----+

// |value|

// +-----+

// | 10|

// +-----+

//

This looks great! It reminds us the familiar syntax from Scala. The two closures in filter and map are functions that operate on Foo and the compiler will helps us capture all the mistakes we mentioned above.

scala> ds.filter(\_.i == 10).map(\_.x).show()

<console>:20: error: value x is not a member of Foo

ds.filter(\_.i == 10).map(\_.x).show()

^

Unfortunately, this syntax does not allow Spark to optimize the code.

ds.filter(\_.i == 10).map(\_.i).explain()

// == Physical Plan ==

// \*SerializeFromObject [input[0, bigint, false] AS value#1805L]

// +- \*MapElements <function1>, obj#1804: bigint

// +- \*Filter <function1>.apply

// +- \*DeserializeToObject newInstance(class $line14.$read$$iw$$iw$$iw$$iw$Foo), obj#1803: $line14.$read$$iw$$iw$$iw$$iw$Foo

// +- \*FileScan parquet [i#1771L,j#1772] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/tmp/foo], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<i:bigint,j:string>

As we see from the explained Physical Plan, Spark was not able to optimize our query as before. Reading the parquet file will required loading all the fields of Foo. This might be ok for small datasets or for datasets with few columns, but will be extremely slow for most practical applications. Intuitively, Spark currently does not have a way to look inside the code we pass in these two closures. It only knows that they both take one argument of type Foo, but it has no way of knowing if we use just one or all of Foo's fields.

The TypedDataset in Frameless solves this problem. It allows for a simple and type-safe syntax with a fully optimized query plan.

import frameless.TypedDataset

// import frameless.TypedDataset

import frameless.syntax.\_

// import frameless.syntax.\_

val fds = TypedDataset.create(ds)

// fds: frameless.TypedDataset[Foo] = [i: bigint, j: string]

fds.filter(fds('i) === 10).select(fds('i)).show().run()

// +---+

// | \_1|

// +---+

// | 10|

// +---+

//

And the optimized Physical Plan:

fds.filter(fds('i) === 10).select(fds('i)).explain()

// == Physical Plan ==

// \*Project [i#1771L AS \_1#1876L]

// +- \*Filter (isnotnull(i#1771L) && (i#1771L = 10))

// +- \*FileScan parquet [i#1771L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/tmp/foo], PartitionFilters: [], PushedFilters: [IsNotNull(i), EqualTo(i,10)], ReadSchema: struct<i:bigint>

And the compiler is our friend.

scala> fds.filter(fds('i) === 10).select(fds('x))

<console>:24: error: No column Symbol with shapeless.tag.Tagged[String("x")] of type A in Foo

fds.filter(fds('i) === 10).select(fds('x))

^

## Differences in Encoders

Encoders in Spark's Datasets are partially type-safe. If you try to create a Dataset using a type that is not a Scala Product then you get a compilation error:

class Bar(i: Int)

// defined class Bar

Bar is neither a case class nor a Product, so the following correctly gives a compilation error in Spark:

scala> spark.createDataset(Seq(new Bar(1)))

<console>:24: error: Unable to find encoder for type stored in a Dataset. Primitive types (Int, String, etc) and Product types (case classes) are supported by importing spark.implicits.\_ Support for serializing other types will be added in future releases.

spark.createDataset(Seq(new Bar(1)))

^

However, the compile type guards implemented in Spark are not sufficient to detect non encodable members. For example, using the following case class leads to a runtime failure:

case class MyDate(jday: java.util.Date)

// defined class MyDate

val myDateDs = spark.createDataset(Seq(MyDate(new java.util.Date(System.currentTimeMillis))))

// java.lang.UnsupportedOperationException: No Encoder found for java.util.Date

// - field (class: "java.util.Date", name: "jday")

// - root class: "MyDate"

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:632)

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:455)

// at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeConstraints.scala:56)

// at org.apache.spark.sql.catalyst.ScalaReflection$class.cleanUpReflectionObjects(ScalaReflection.scala:809)

// at org.apache.spark.sql.catalyst.ScalaReflection$.cleanUpReflectionObjects(ScalaReflection.scala:39)

// at org.apache.spark.sql.catalyst.ScalaReflection$.org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor(ScalaReflection.scala:455)

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1$$anonfun$10.apply(ScalaReflection.scala:626)

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1$$anonfun$10.apply(ScalaReflection.scala:614)

// at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:241)

// at scala.collection.TraversableLike$$anonfun$flatMap$1.apply(TraversableLike.scala:241)

// at scala.collection.immutable.List.foreach(List.scala:392)

// at scala.collection.TraversableLike$class.flatMap(TraversableLike.scala:241)

// at scala.collection.immutable.List.flatMap(List.scala:355)

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:614)

// at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor$1.apply(ScalaReflection.scala:455)

// at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeConstraints.scala:56)

// at org.apache.spark.sql.catalyst.ScalaReflection$class.cleanUpReflectionObjects(ScalaReflection.scala:809)

// at org.apache.spark.sql.catalyst.ScalaReflection$.cleanUpReflectionObjects(ScalaReflection.scala:39)

// at org.apache.spark.sql.catalyst.ScalaReflection$.org$apache$spark$sql$catalyst$ScalaReflection$$serializerFor(ScalaReflection.scala:455)

// at org.apache.spark.sql.catalyst.ScalaReflection$.serializerFor(ScalaReflection.scala:444)

// at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder$.apply(ExpressionEncoder.scala:71)

// at org.apache.spark.sql.Encoders$.product(Encoders.scala:275)

// at org.apache.spark.sql.LowPrioritySQLImplicits$class.newProductEncoder(SQLImplicits.scala:233)

// at org.apache.spark.sql.SQLImplicits.newProductEncoder(SQLImplicits.scala:33)

// ... 770 elided

In comparison, a TypedDataset will notify about the encoding problem at compile time:

TypedDataset.create(Seq(MyDate(new java.util.Date(System.currentTimeMillis))))

// <console>:25: error: could not find implicit value for parameter encoder: frameless.TypedEncoder[MyDate]

// TypedDataset.create(Seq(MyDate(new java.util.Date(System.currentTimeMillis))))

// ^

## Aggregate vs Projected columns

Spark's Dataset do not distinguish between columns created from aggregate operations, such as summing or averaging, and simple projections/selections. This is problematic when you start mixing the two.

import org.apache.spark.sql.functions.sum

// import org.apache.spark.sql.functions.sum

ds.select(sum($"i"), $"i"\*2)

// org.apache.spark.sql.AnalysisException: grouping expressions sequence is empty, and '`i`' is not an aggregate function. Wrap '(sum(`i`) AS `sum(i)`)' in windowing function(s) or wrap '`i`' in first() (or first\_value) if you don't care which value you get.;;

// Aggregate [sum(i#1771L) AS sum(i)#1889L, (i#1771L \* cast(2 as bigint)) AS (i \* 2)#1890L]

// +- Relation[i#1771L,j#1772] parquet

//

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$class.failAnalysis(CheckAnalysis.scala:39)

// at org.apache.spark.sql.catalyst.analysis.Analyzer.failAnalysis(Analyzer.scala:91)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1(CheckAnalysis.scala:239)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1$5.apply(CheckAnalysis.scala:253)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1$5.apply(CheckAnalysis.scala:253)

// at scala.collection.immutable.List.foreach(List.scala:392)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1(CheckAnalysis.scala:253)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1$5.apply(CheckAnalysis.scala:253)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1$5.apply(CheckAnalysis.scala:253)

// at scala.collection.immutable.List.foreach(List.scala:392)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.org$apache$spark$sql$catalyst$analysis$CheckAnalysis$class$$anonfun$$checkValidAggregateExpression$1(CheckAnalysis.scala:253)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$9.apply(CheckAnalysis.scala:280)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1$$anonfun$apply$9.apply(CheckAnalysis.scala:280)

// at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)

// at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.apply(CheckAnalysis.scala:280)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$$anonfun$checkAnalysis$1.apply(CheckAnalysis.scala:78)

// at org.apache.spark.sql.catalyst.trees.TreeNode.foreachUp(TreeNode.scala:127)

// at org.apache.spark.sql.catalyst.analysis.CheckAnalysis$class.checkAnalysis(CheckAnalysis.scala:78)

// at org.apache.spark.sql.catalyst.analysis.Analyzer.checkAnalysis(Analyzer.scala:91)

// at org.apache.spark.sql.execution.QueryExecution.assertAnalyzed(QueryExecution.scala:52)

// at org.apache.spark.sql.Dataset$.ofRows(Dataset.scala:67)

// at org.apache.spark.sql.Dataset.org$apache$spark$sql$Dataset$$withPlan(Dataset.scala:2884)

// at org.apache.spark.sql.Dataset.select(Dataset.scala:1150)

// ... 854 elided

In Frameless, mixing the two results in a compilation error.

// To avoid confusing frameless' sum with the standard Spark's sum

import frameless.functions.aggregate.{sum => fsum}

// import frameless.functions.aggregate.{sum=>fsum}

fds.select(fsum(fds('i)))

// <console>:26: error: polymorphic expression cannot be instantiated to expected type;

// found : [Out]frameless.TypedAggregate[Foo,Out]

// required: frameless.TypedColumn[Foo,?]

// fds.select(fsum(fds('i)))

// ^

As the error suggests, we expected a TypedColumn but we got a TypedAggregate instead.

Here is how you apply an aggregation method in Frameless:

fds.agg(fsum(fds('i))+22).show().run()

// +---+

// | \_1|

// +---+

// |133|

// +---+

//

Similarly, mixing projections while aggregating does not make sense, and in Frameless you get a compilation error.

fds.agg(fsum(fds('i)), fds('i)).show().run()

// <console>:26: error: polymorphic expression cannot be instantiated to expected type;

// found : [A]frameless.TypedColumn[Foo,A]

// required: frameless.TypedAggregate[Foo,?]

// fds.agg(fsum(fds('i)), fds('i)).show().run()

//