Spark Schema for Free

Yet another shapeless use-case

Szakállas Dávid

Whitepages®

OUR JOB

- We have terabytes of structured batch data
- We need to build an ETL job
- The technology is fixed: Apache Spark
- Using Scala, because ❤️
- It has to be blazingly fast, because 💸

OPTIONS

- RDDs (Resilient Distributed Datasets) of JVM objects
  - Strongly typed
  - Functional
  - Memory heavy
  - CPU heavy
  - Requires sound knowledge of distributed processing
  - No cost based optimization

NOTORIOUS RDD PITFALL

- Requires sound knowledge of distributed processing

Typical example

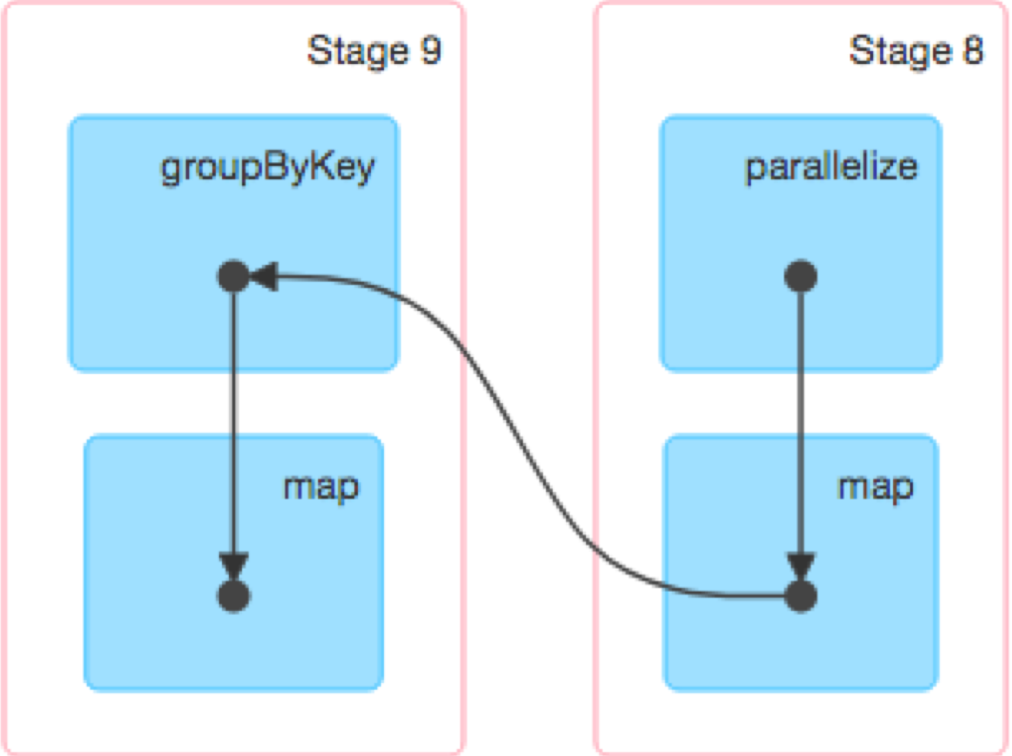
```
spark.sparkContext
.parallelize(Seq("a", "a", "b", "b", "b"))
.map((_, 1))

.groupByKey()

.map{case (k,v) ⇒ (k, v.foldLeft(0)(_ + _))}
.collect()
```

```
// Array("a", "a", "b", "b", "b")
// Array((a,1), (a,1), (b,1),
         (b,1), (b,1))
// Array((a,CompactBuffer(1, 1)),
         (b,CompactBuffer(1, 1, 1)))
// Array((a,2), (b,3))
```

```
spark.sparkContext
.parallelize(Seq("a", "a", "b", "b", "b"))
.map((_, 1))

.reduceByKey(_ + _)
.collect()
```

```
// Array("a", "a", "b", "b", "b")
// Array((a,1), (a,1), (b,1),
         (b,1), (b,1))
// Array((a,2), (b,3))
```

NOTORIOUS RDD PITFALL



Shuffle Read	Shuffle Write
198.0 B	
	198.0 B

Stage 13

parallelize

map

Stage 14

reduceByKey

Shuffle Read	Shuffle Write
192.0 B	
	192.0 B

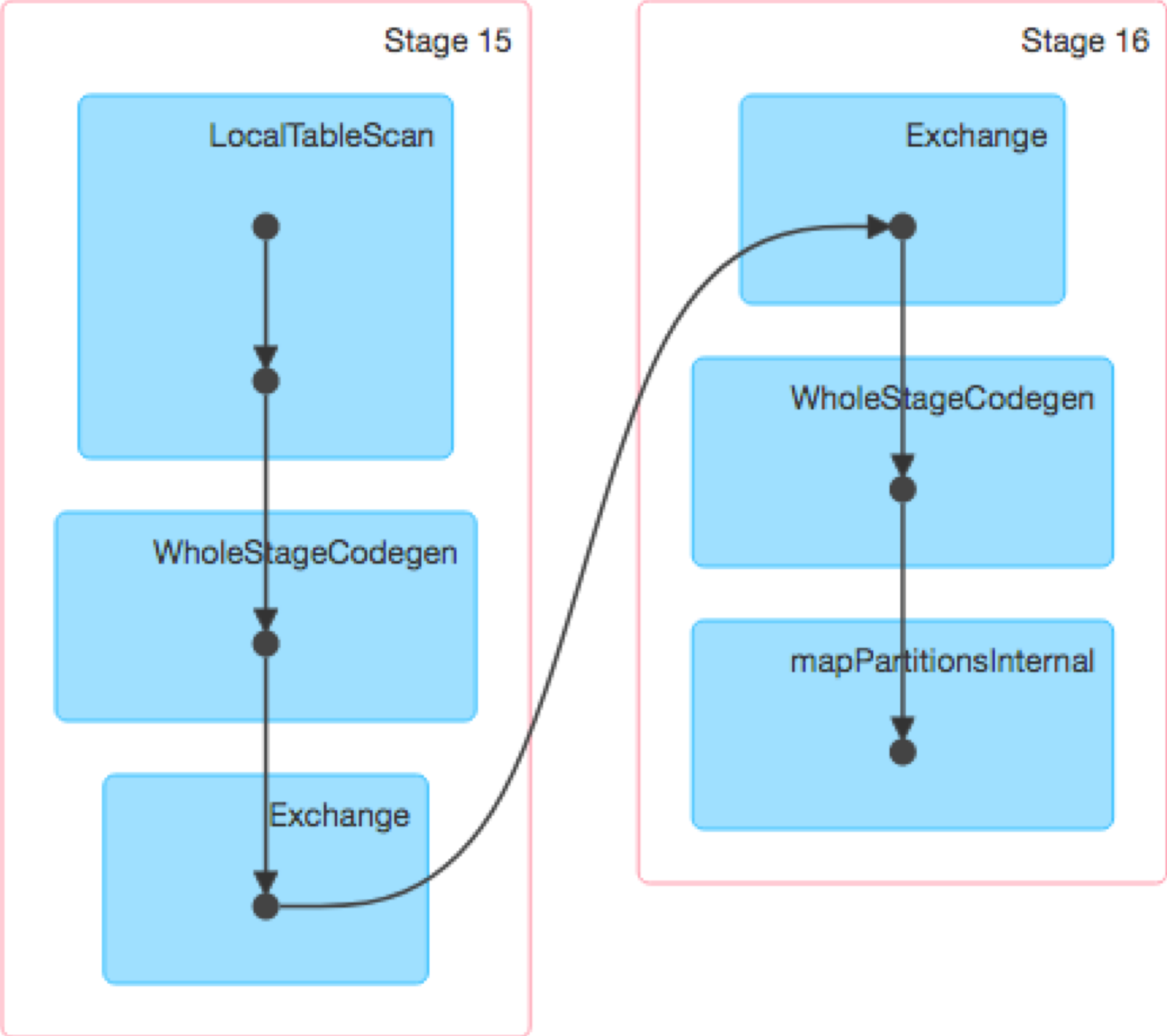
OPTIONS

- RDDs (Resilient Distributed Datasets) of JVM objects
- DataFrames
  - SQL type system −
  - Declarative +
  - Memory and bandwidth friendly (Tungsten) +
  - CPU-friendly (vectorized, no GC) (Tungsten) +
  - Full range of SQL optimizations +
    - Predicate pushdown
    - Join optimization
    - Network transfer minimalization

DATAFRAMES

```
val df = spark
  .createDataset(Seq("a", "a", "b", "b", "b"))
  .toDF
  .groupBy("value")
  .count()

df.collect() // Array[org.apache.spark.sql.Row] = Array([b,3], [a,2])

df.explain(extended = true)

== Optimized Logical Plan ==
Aggregate [value#381], [value#381, count(1) AS count#386L]
+- LocalRelation [value#381]

== Physical Plan ==
*(2) HashAggregate(keys=[value#381], functions=[count(1)], output=[value#381,
count#386L])
+- Exchange hashpartitioning(value#381, 200)
   +- *(1) HashAggregate(keys=[value#381], functions=[partial_count(1)],
output=[value#381, count#391L])
      +- LocalTableScan [value#381]
```

DATAFRAMES



Shuffle Read	Shuffle Write
281.0 B	
	281.0 B

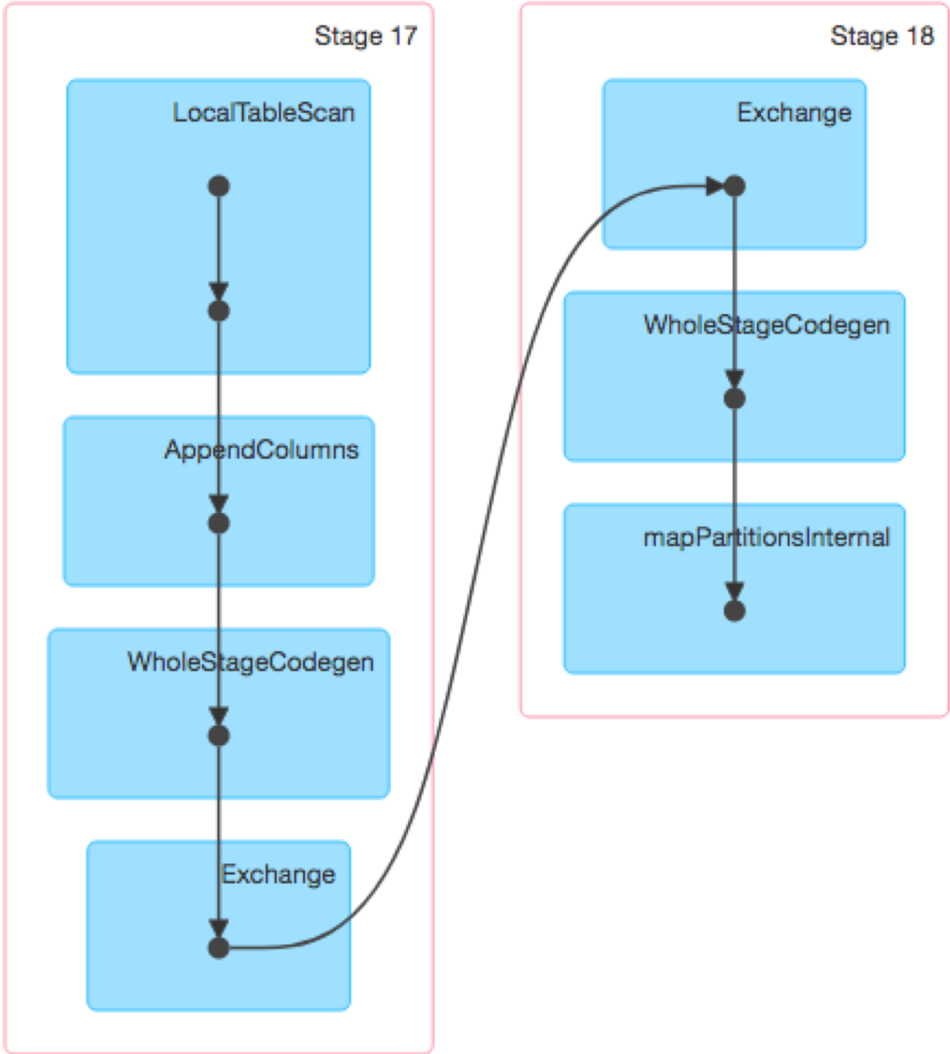
OPTIONS

- RDDs (Resilient Distributed Datasets) of JVM objects
- DataFrames
- Datasets
  - Scala on top of SQL
  - Serde required between Scala ⇔ SQL
  - Strongly typed
  - Optimization barrier
  - Encoding overhead
  - DataFrame is Dataset[Row]

DATASETS

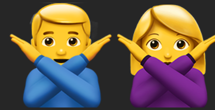
```
val ds = spark
  .createDataset(Seq("a", "a", "b", "b", "b"))
  .groupByKey(x => x)
  .count()

ds.collect() // Array[(String, Long)] = Array((b,3), (a,2))


ds.explain(extended = true)

== Optimized Logical Plan ==
Aggregate [value#405], [value#405, count(1) AS count(1)#409L]
+- Project [value#405]
   +- AppendColumns <function1>, class java.lang.String,
[StructField(value,StringType,true)], value#402.toString, [staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0,
java.lang.String, true], true, false) AS value#405]
      +- LocalRelation [value#402]

== Physical Plan ==
…
```

DATASETS



Shuffle Read	Shuffle Write
281.0 B	
	281.0 B

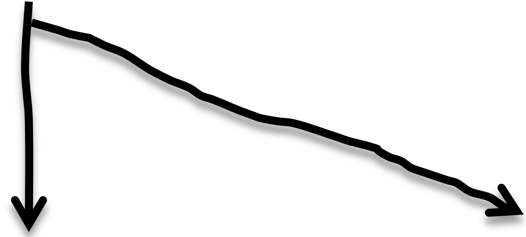
we need performance
but
we are not going to write type level proofs
for SQL

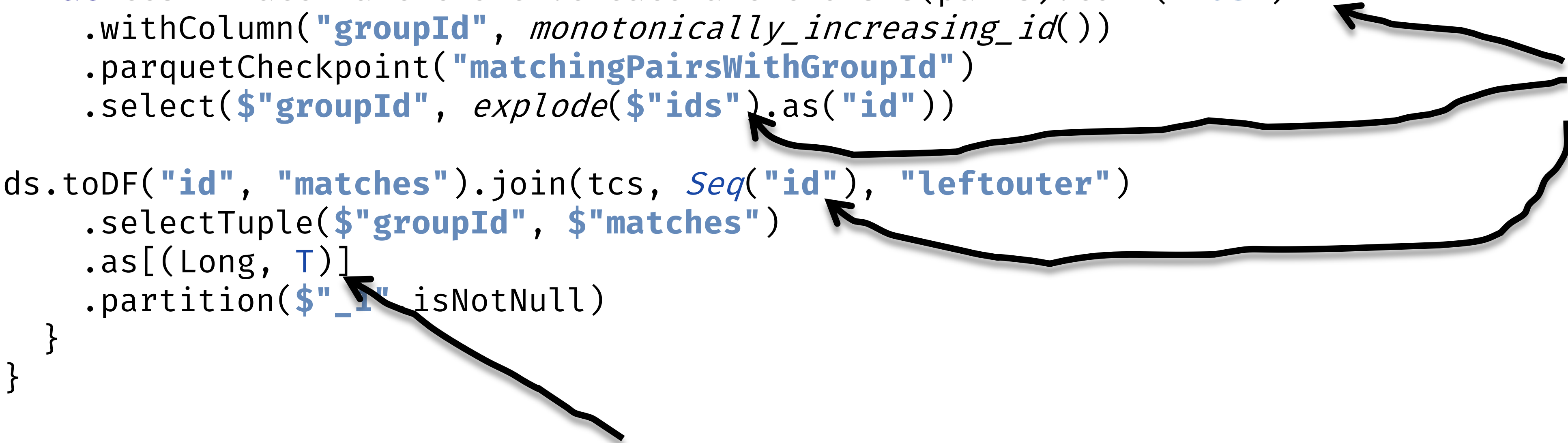


so
use DataFrame operations where possible
cast to Dataset[T] to communicate the type
on public interfaces

EXAMPLE

typed dataset in





```scala
trait MatchingGrouperDS {
  def partitionByMatchingPairs[T: TypedEncoder](ds: Dataset[(Long, T)], pairs: Dataset[(Long, Long)])
  (implicit spark: SparkSession): (Dataset[(Long, T)], Dataset[(Long, T)]) = {
  val tcs = MatchPartitioner.createPartitionsDS(pairs).toDF("ids")
    .withColumn("groupId", monotonically_increasing_id())
    .parquetCheckpoint("matchingPairsWithGroupId")
    .select($"groupId", explode($"ids").as("id"))

ds.toDF("id", "matches").join(tcs, Seq("id"), "leftouter")
    .selectTuple($"groupId", $"matches")
    .as[(Long, T)]
    .partition($"_1".isNotNull)
  }
}
```

untyped
operations

cast to dataset

ENCODER[T]

- serde between optimized (Tungsten) and Scala representation
- shuffle always carried out with optimized repr!
- needed whenever we want a Dataset[T]
  - `def as[U : Encoder]: Dataset[U] = ???`
  - `def createDataset[T : Encoder](data: Seq[T]): Dataset[T] = ???`
  - `def emptyDataset[T: Encoder]: Dataset[T] = ???`

OOTB ENCODER SUPPORT

```
case class Name(firstName: String, middleName: String,
lastName: String)
case class Person(names: Seq[Name], birthDate:
java.sql.Timestamp)

spark.emptyDataset[Person]



spark.emptyDataset[java.util.UUID]
```

Message: <console>:67: 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.emptyDataset[java.util.UUID]

ENCODER[UUID] ...

```scala
import org.apache.spark.sql.catalyst.analysis._
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.catalyst.expressions._
import org.apache.spark.sql.catalyst.expressions.objects._
import org.apache.spark.sql.types._

val catalystRepr = StructType(Array(
  StructField("msb", LongType, nullable = false),
  StructField("lsb", LongType, nullable = false)
))

val jvmRepr = ObjectType(classOf[UUID])

// create a table that contains two columns
// msb: long - derived by calling uuid.getMostSignificantBits()
val serializer = CreateNamedStruct(Seq(
  Literal("msb"),
  Invoke(BoundReference(0, jvmRepr, false), "getMostSignificantBits", LongType),
  Literal("lsb"),
  Invoke(BoundReference(0, jvmRepr, false), "getLeastSignificantBits", LongType)
)).flatten
```

... ENCODER[UUID]

```scala
// derive the JVM object by calling `new UUID(col0, col1)`
val deserializer = NewInstance(
  classOf[UUID],
  Seq(
    GetColumnByOrdinal(0, LongType),
    GetColumnByOrdinal(1, LongType)
  ),
  ObjectType(classOf[UUID])
)

import scala.reflect.classTag

val uuidEncoder = new ExpressionEncoder[UUID](
  schema = catalystRepr,
  flat = false,
  serializer = serializer,
  deserializer = deserializer,
  clsTag = classTag[UUID]
)
```

IS IT WORKING?

```scala
implicit val enc = uuidEncoder

spark.emptyDataset[java.util.UUID]
```

```scala
spark.emptyDataset[(java.util.UUID, java.util.UUID)]
```

```
java.lang.UnsupportedOperationException: No Encoder found for java
- field (class: "java.util.UUID", name: "_1")
- root class: "scala.Tuple2"
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
  at scala.reflect.internal.tpe.TypeConstraints$UndoLog.undo(TypeC
  at org.apache.spark.sql.catalyst.ScalaReflection$class.cleanUpRe
  at org.apache.spark.sql.catalyst.ScalaReflection$.cleanUpReflect
  at org.apache.spark.sql.catalyst.ScalaReflection$.org$apache$spa
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
```

IS IT WORKING?

```
implicit val enc = uuidEncoder

spark.emptyD

spark.emptyD                                                ID)]
java.lang.Uns                                              er found for java
- field (clas
- root class:
  at org.apac                                              n$$anonfun$org$ap
  at org.apac                                              n$$anonfun$org$ap
  at scala.re                                              ndoLog.undo(TypeC
  at org.apac                                              n$class.cleanUpRe
  at org.apac                                              n$.cleanUpReflect
  at org.apache.spark.sql.catalyst.ScalaReflection$.org$apache$spa
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
```

IS IT WORKING?

```
implicit val enc = uuidEncoder

spark.e

spark.e                                                  ID)]
java.la                                          er found for java
- field
- root
  at or                                          n$$anonfun$org$ap
  at or                                          n$$anonfun$org$ap
  at sc                                          ndoLog.undo(TypeC
  at or                                          n$class.cleanUpRe
  at or                                          n$.cleanUpReflect
  at or                                        lection$.org$apache$spa
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
  at org.apache.spark.sql.catalyst.ScalaReflection$$anonfun$org$ap
```

IS IT WORKING?

```scala
implicit val enc = uuidEncoder

spark.e

spark.e
```

```
java.la                                      java
- field
- root
  at or                                      rg$ap
  at or                                      rg$ap
  at sc                                      TypeC
  at or                                      nUpRe
  at or                                      flect
  at or                                      e$spa
  at org.apache.spark.sql.c                  rg$ap
```

OOTB ENCODER SUPPORT

- Predefined set of types
  - Unsupported type => compile time error
- Encoder search is not recursive
  - Uses reflection to serialize fields of Product
  - Unsupported field => runtime error
- Not extendable

SOLUTION

- A types are composites of products, sequences, custom serializable types
- Problem similar to JSON serialization
- Idea: use generic programming
- Shapeless

```scala
import org.apache.spark.sql.Encoder
import shapeless.HList

trait ComposableEncoder[T] {
  // ???
}
object ComposableEncoder {
  // derive Spark Encoder
  implicit def getEncoder[T: ComposableEncoder]: Encoder[T] = ???
}

implicit val intEncoder: ComposableEncoder[Int] = ???
implicit val longEncoder: ComposableEncoder[Long] = ???
  // ...
  // other primitive types
  // ...
implicit val uuidEncoder: ComposableEncoder[UUID] = ???

// compound types
implicit def productEncoder[G, Repr <: HList]: ComposableEncoder[T] =
  ???
implicit def arrayEncoder[T: ClassTag]: ComposableEncoder[Array[T]] =
  ???
// Option, Either, etc.
```

FRAMELESS

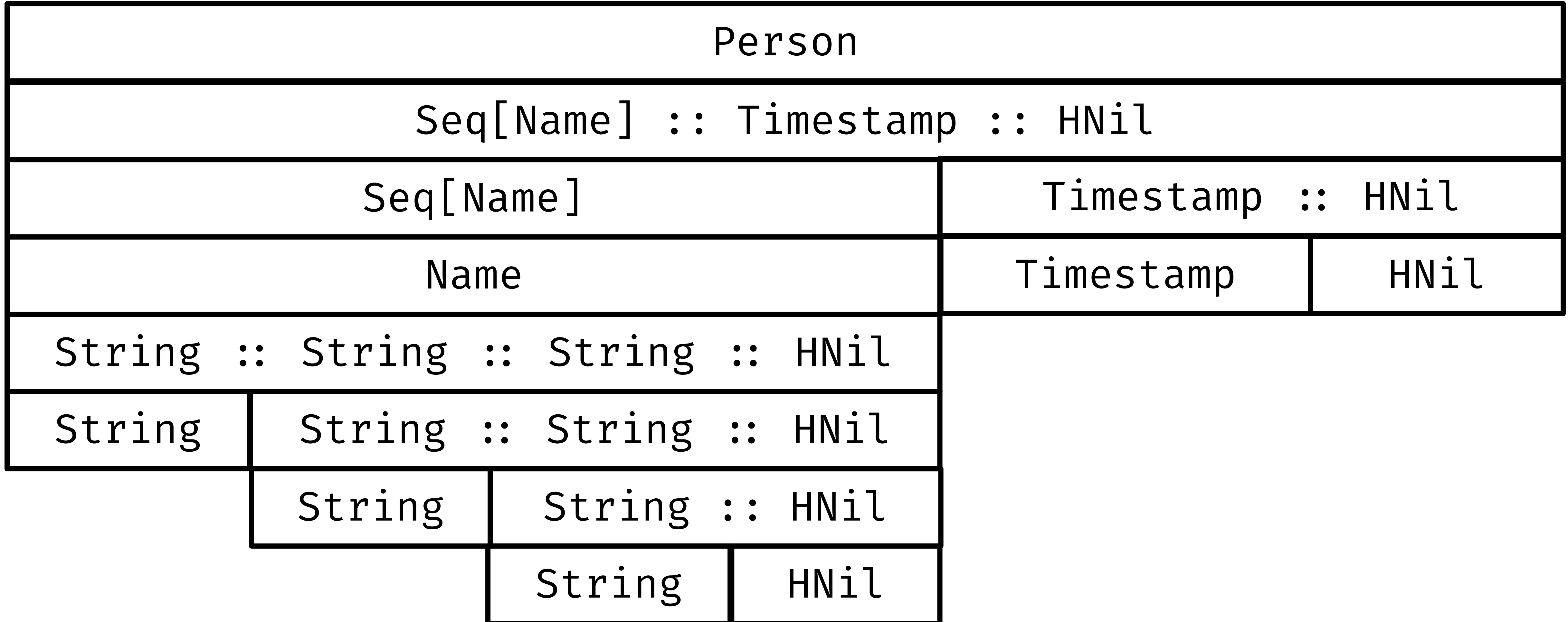
- Expressive types for Spark
- https://github.com/typelevel/frameless
- More type safe Datasets
  - Typesafe columns referencing
  - Customizable, typesafe encoders
  - Enhanced type signature for built-in functions
  - Typesafe casting and projections

PRODUCT DERIVATION OUT OF THE BOX!

USING FRAMELESS

```scala
import frameless.{TypedEncoder, TypedExpressionEncoder}

// similarly to a view bound, this enables a final implicit conversion
// from TypedEncoder[T] ⇒ Encoder[T]
implicit def typedEncoder[T: TypedEncoder]: Encoder[T] = TypedExpressionEncoder[T]

// this is the actual encoder
implicit def uuidTypedEncoder: TypedEncoder[UUID] = new TypedEncoder[UUID] {
  def nullable: Boolean = false

  def jvmRepr: DataType = ObjectType(classOf[UUID])
  def catalystRepr: DataType = TypedEncoder[(Long, Long)].catalystRepr

  def toCatalyst(path: Expression): Expression = {
    val msb = Invoke(path, "getMostSignificantBits", LongType)
    val lsb = Invoke(path, "getLeastSignificantBits", LongType)
    CreateNamedStruct(Seq(Literal("_1"), msb, Literal("_2"), lsb))
  }

  def fromCatalyst(path: Expression): Expression = {
    val msb = GetStructField(path, 0, Some("msb"))
    val lsb = GetStructField(path, 1, Some("lsb"))
    NewInstance(classOf[UUID], Seq(msb, lsb), jvmRepr)
  }
}
```

```scala
case class Name(firstName: String, middleName: String, lastName: String)
case class Person(names: Seq[Name], birthDate: java.sql.Timestamp)

spark.emptyDataset[Person]
```



DEMO

CONCLUSION

- Use DataFrames/Datasets on Spark
- Don't give up type-safety where not necessary
- Scala + custom types => frameless
  - We actually dropped it and reimplemented the encoders from scratch
  - but good luck typing SQL
- Spark still stuck on Scala 2.11
  - very slow implicit derivation
  - Compiler bugs

Scala 2.12 support will likely land in Spark 2.4

Much faster implicit derivation!