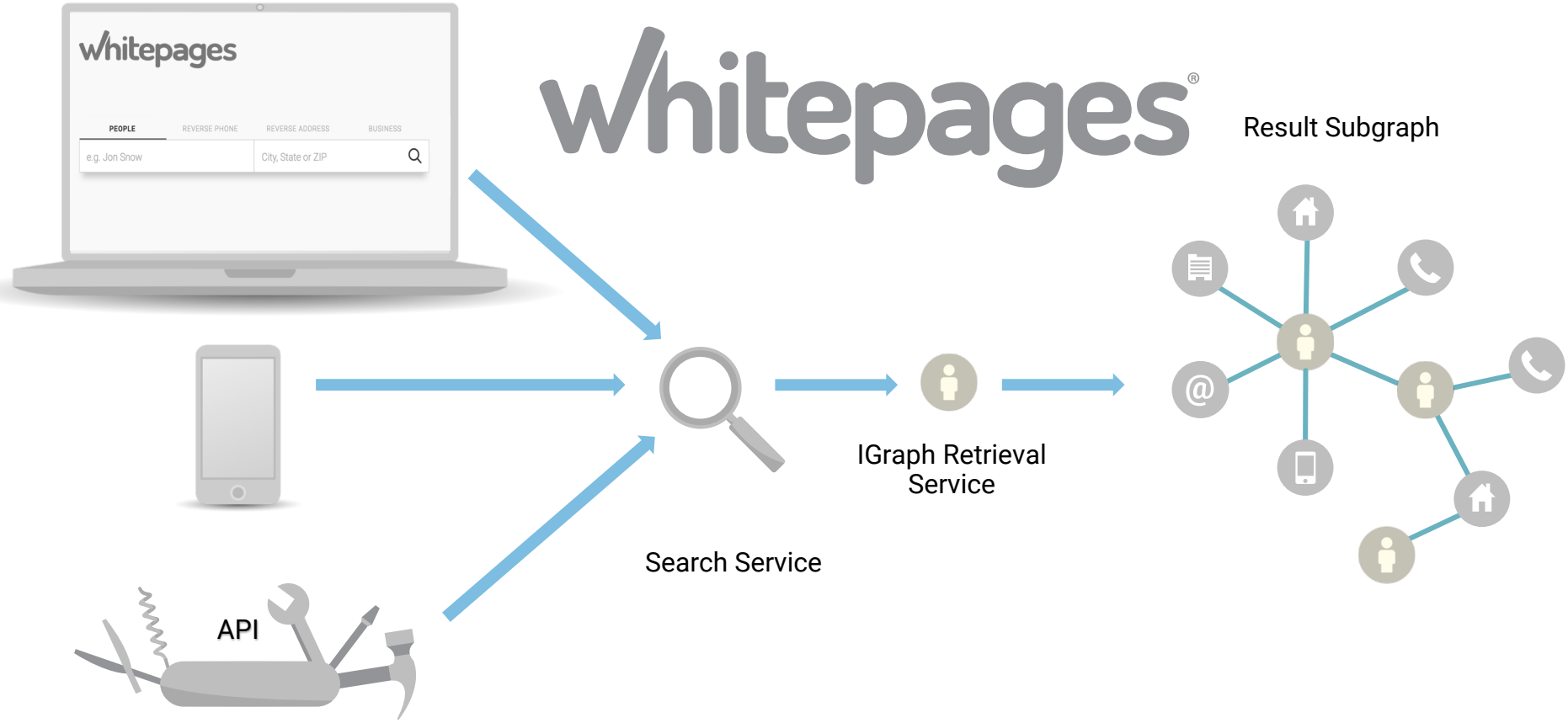
Spark Schema for Free

Dávid Szakállas, Whitepages
@szdavid92

Whitepages®

#schema4free


whitepages
PEOPLE
REVERSE PHONE
REVERSE ADDRESS
BUSINESS
e.g. Jon Snow
City, State or ZIP
whitepages
Result Subgraph
IGraph Retrieval Service
Search Service
API

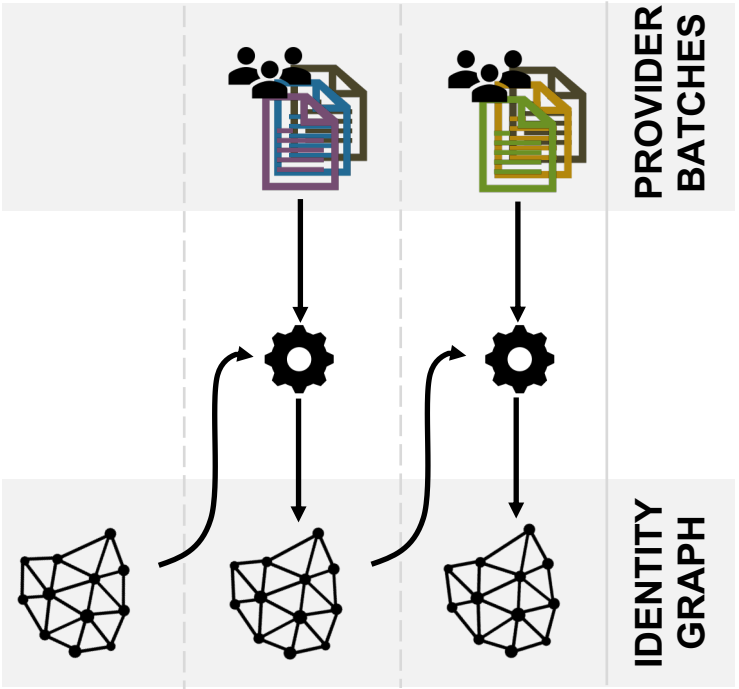
Whitepages Identity Graph™

4B+ entities

6B+ links

Whitepages®

Our story in a nutshell



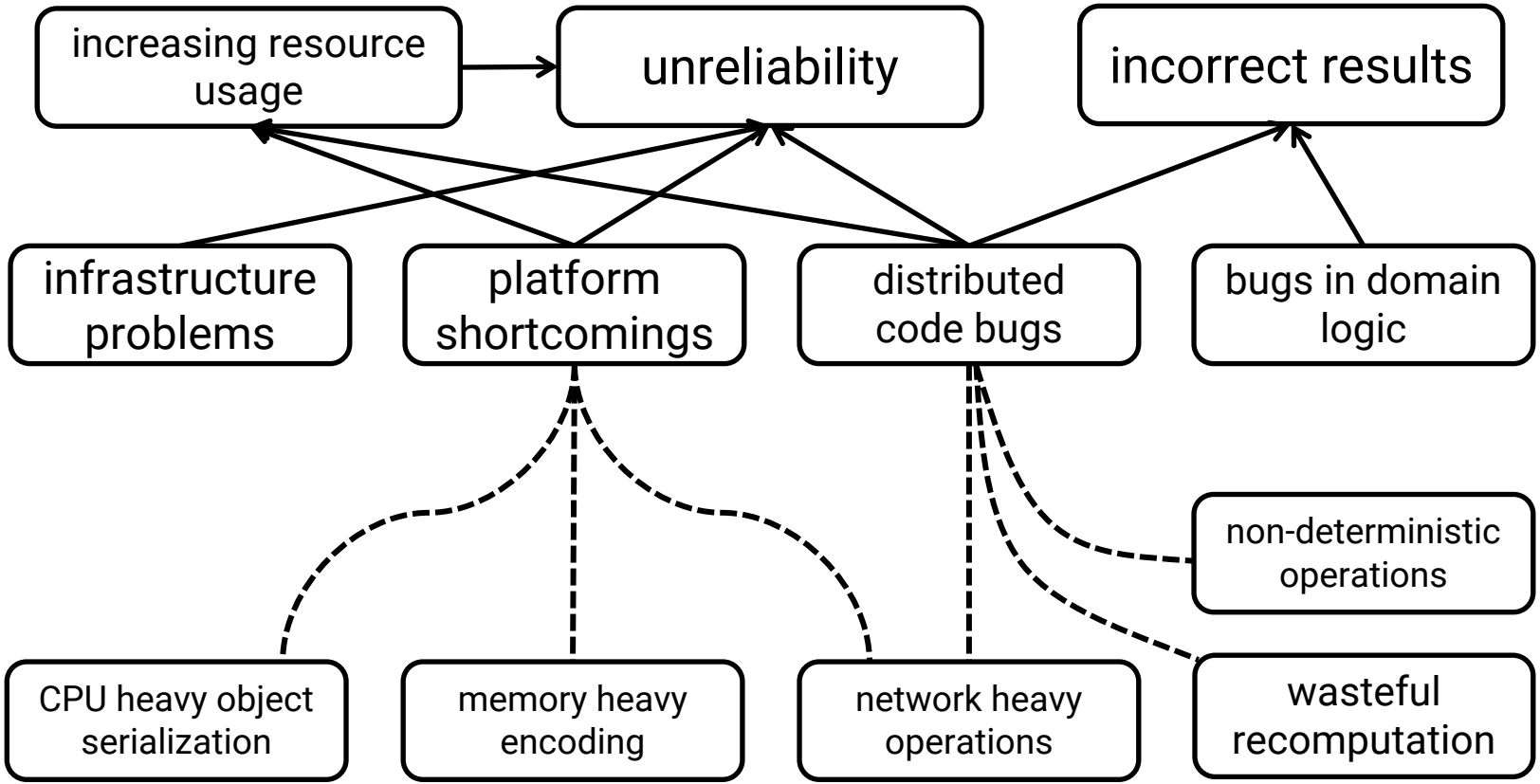
> 30 000 SLOC

RDDs

3rd PARTY LIBRARIES

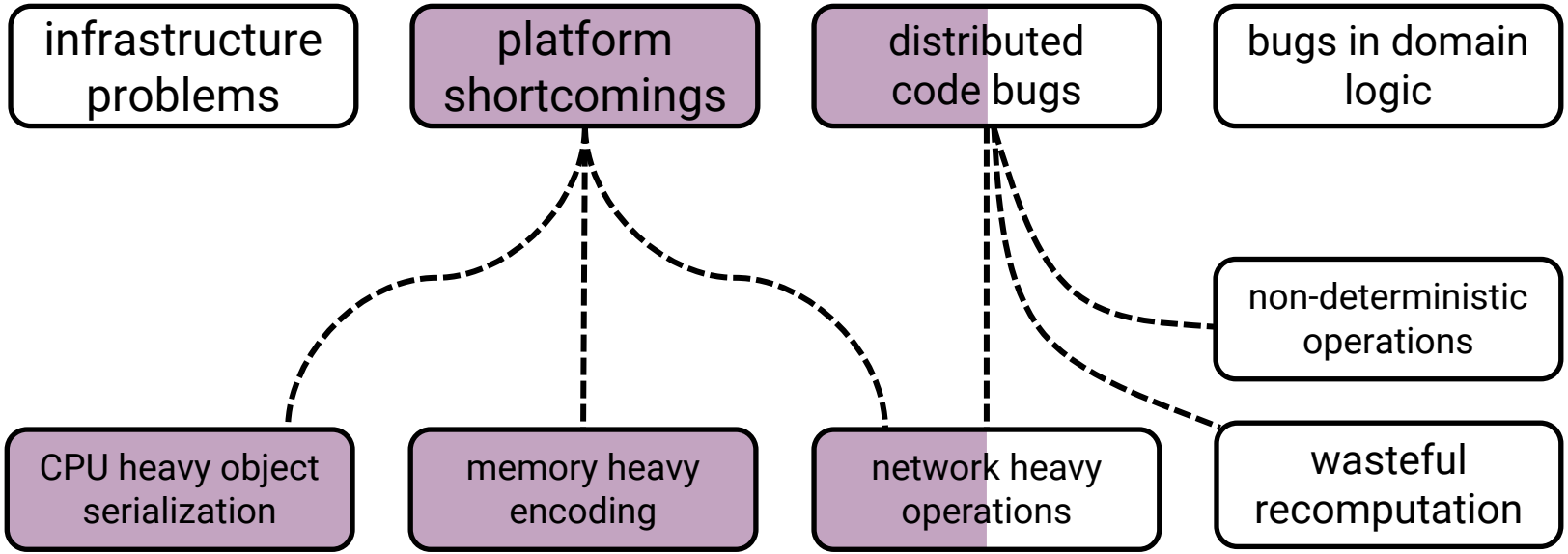
RICH DOMAIN TYPES

FUNCTIONAL STYLE


increasing resource usage
unreliability
incorrect results
infrastructure problems
platform shortcomings
distributed code bugs
bugs in domain logic
non-deterministic operations
CPU heavy object serialization
memory heavy encoding
network heavy operations
wasteful recomputation

Switching to Spark SQL

20% speedup just by s/RDD/Dataset/g



Additional requirements

- Keep compatibility with existing output schema
- Use Scala
- Retain compile-time type safety
- Reuse existing domain types
  - ~ 30 core types as case classes
- Leverage the query optimizer
- Minimize memory footprint
  - spare object allocations where possible
- Reduce boilerplate

Two and a half APIs

	Dataset[T]	DataFrame (Dataset[Row])	SQL
	only logic errors	mistyped column name	syntax error
			
			

: Dataset[T]

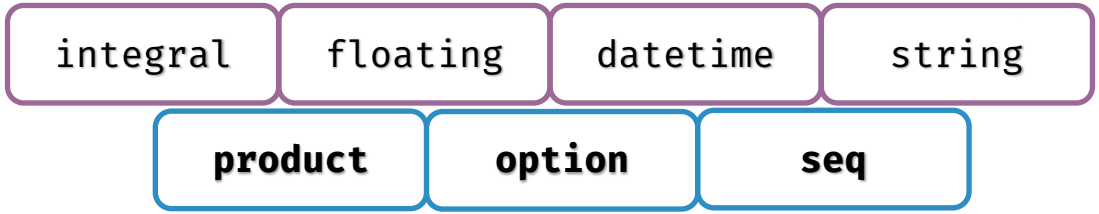
`filter(T => Boolean)`	**DO**
`map(T => U)`	**SDO**
`mapPartitions(T => U)`	**SDO**
`flatMap`	**SDO**
`groupByKey`	**SDO**
`reduce`	**SDO**
`joinWith`	**U**
`dropDuplicates(c: Column)`	**U**
`orderBy, ...`	**U**
`as[T]`	**C**

S: Serialization cost
D: Deserialization cost
O: Optimization barrier
U: Untyped column referencing
C: Checked cast

1. operations with performance problems
2. operations with type safety problems
3. creating a dataset of an arbitrary type

Encoders

- JVM ⇔ Tungsten
- out of the box we get:



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

COMPILE TIME ERROR

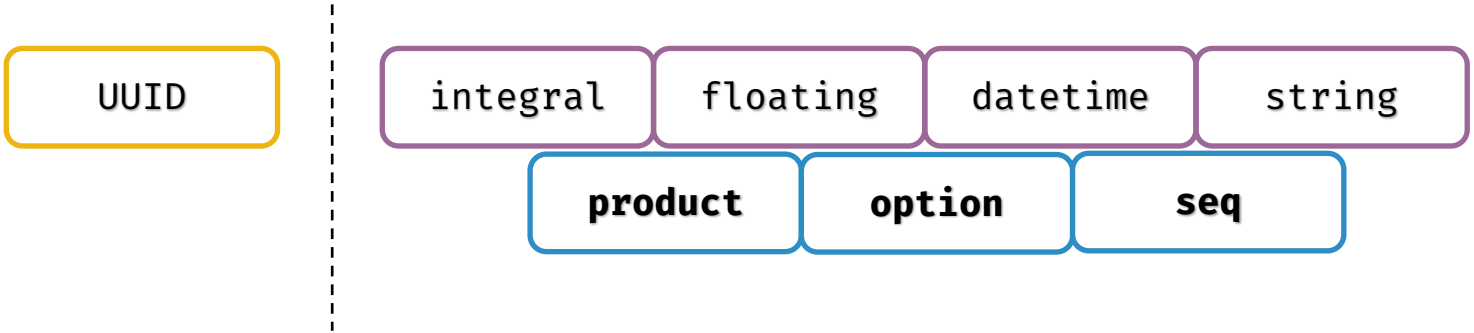
```
error: Unable to find encoder for type stored in a Dataset.
Primitive types (Int, String, etc) and Product types (case
classes) are supported by importing spark.implicits._
Support for serializing other types will be added in future
releases. spark.emptyDataset[java.util.UUID]
```

```scala
val jvmRepr = ObjectType(classOf[UUID])

val serializer = CreateNamedStruct(Seq(
  Literal("msb"),
  Invoke(BoundReference(0, jvmRepr, false), "getMostSignificantBits", LongType),
  Literal("lsb"),
  Invoke(BoundReference(0, jvmRepr, false), "getLeastSignificantBits", LongType)
)).flatten

val deserializer = NewInstance(classOf[UUID],
  Seq(GetColumnByOrdinal(0, LongType), GetColumnByOrdinal(1, LongType)),
  ObjectType(classOf[UUID]))

implicit val uuidEncoder = new ExpressionEncoder[UUID](
  schema = StructType(Array(
    StructField("msb", LongType, nullable = false),
    StructField("lsb", LongType, nullable = false)
  )),
  flat = false,
  serializer = serializer,
  deserializer = deserializer,
  clsTag = classTag[UUID])
```

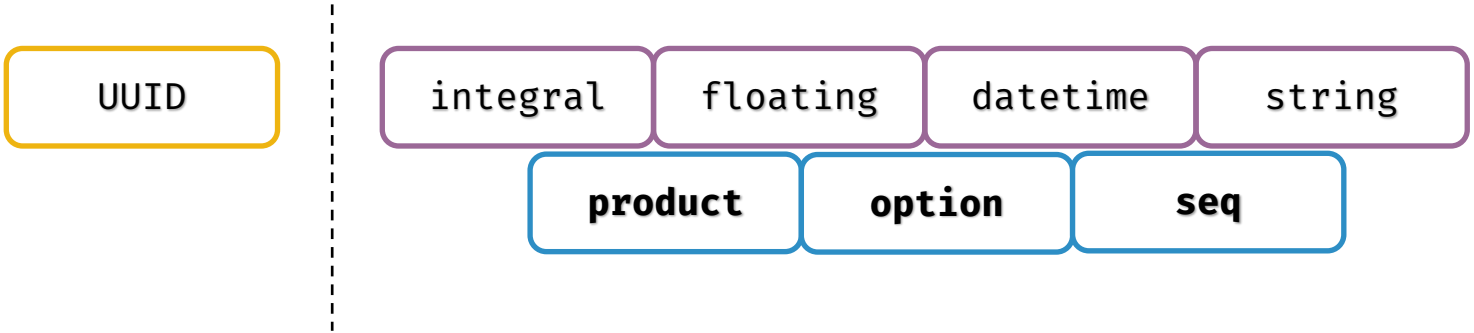
```
spark.emptyDataset[UUID]
```

```
spark.emptyDataset[(UUID, UUID)]
```

```
Message: No Encoder found for java.util.UUID
- field (class: "java.util.UUID", name: "_1")
-  root class: "scala.Tuple2"
StackTrace: - field (class: "java.util.UUID", n
- root class: "scala.Tuple2"
  at org.apache.spark.sql.catalyst.ScalaReflectio  (...)
```




```
/**
 * An encoder for Scala's product type (tuples, case classes, etc).
 * @since 2.0.0
 */
def product[T <: Product : TypeTag]: Encoder[T] = ExpressionEncoder()
```

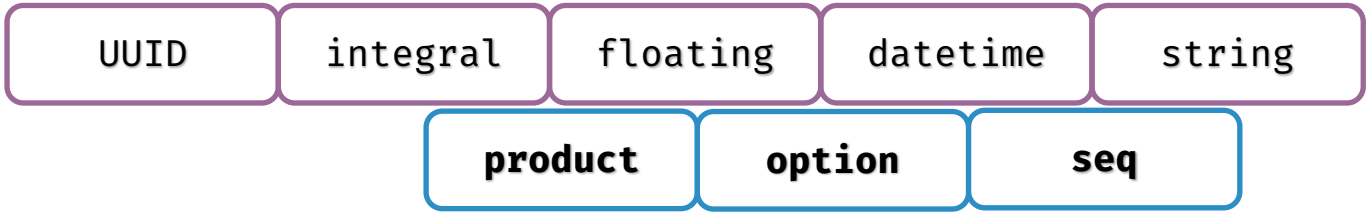



```
object ExpressionEncoder {
  def apply[T : TypeTag](): ExpressionEncoder[T] = {
    // ...
    val serializer = ScalaReflection.serializerFor[T](nullSafeInput)
    val deserializer = ScalaReflection.deserializerFor[T]
```

```
/**
 * An
 * @s
 */
def product[T <: Product : TypeTag]: Encoder[T] = ExpressionEncoder()
```


```
tpe.dealias match {
  case t if !dataTypeFor(t).isInstanceOf[ObjectType] => getPath

  case t if t <:< localTypeOf[Option[_]] =>
    val TypeRef(_, _, Seq(optType)) = t
    val className = getClassNameFromType(optType)
    val newTypePath = s"""- option value class: "$className""" +: walkedTypePath
    WrapOption(deserializerFor(optType, path, newTypePath), dataTypeFor(optType))

  case t if t <:< localTypeOf[java.lang.Integer] =>
    val boxedType = classOf[java.lang.Integer]
    val objectType = ObjectType(boxedType)
    StaticInvoke(boxedType, objectType, "valueOf", getPath :: Nil, returnNullable = false)

  case t if t <:< localTypeOf[java.lang.Long] =>
    val boxedType = classOf[java.lang.Long]
    val objectType = ObjectType(boxedType)
    StaticInvoke(boxedType, objectType, "valueOf", getPath :: Nil, returnNullable = false)

  case t if t <:< localTypeOf[java.lang.Double] =>
    val boxedType = classOf[java.lang.Double]
    val objectType = ObjectType(boxedType)
    StaticInvoke(boxedType, objectType, "valueOf", getPath :: Nil, returnNullable = false)
```

```
/**
 * An
 * @s
 */
def produc
```

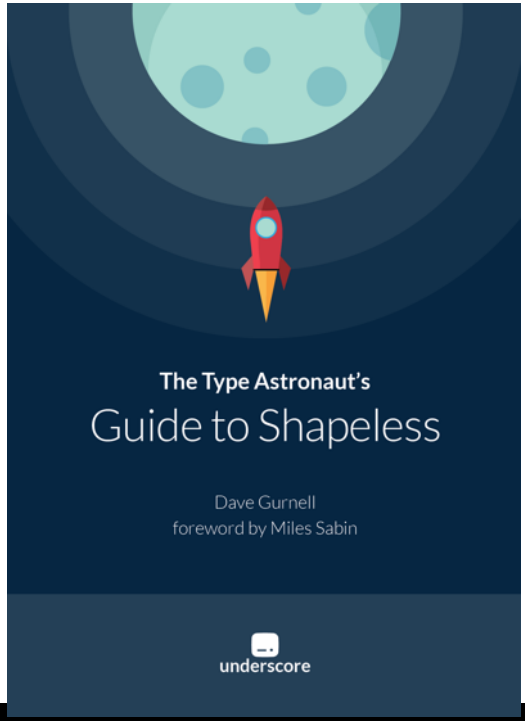


Encoders should be additive

- rewrite encoders from scratch!
- doesn’t require changing Spark internals

Do it in a robust, type-safe way!

- generic programming with shapeless



```scala
trait TypedEncoder[T]
object TypedEncoder {
  // derive Spark Encoder
  implicit def apply[T: TypedEncoder]: ExpressionEncoder[T] = ???
}

implicit val intEncoder: TypedEncoder[Int] = ???
implicit val longEncoder: TypedEncoder[Long] = ???
implicit val uuidEncoder: TypedEncoder[UUID] = ???


implicit def optionTypedEncoder[T](implicit enc: TypedEncoder[T]):
  TypedEncoder[Option[T]] = ???

implicit def seqTypedEncoder[C[X] <: Seq[X], T](implicit
  enc: TypedEncoder[T],
  classTag: ClassTag[C[T]]
): TypedEncoder[C[T]] = ???

implicit def productEncoder[F, G <: HList, H <: HList](implicit
  i0: LabelledGeneric.Aux[F, G],
  i1: DropUnitValues.Aux[G, H],
  i2: IsHCons[H],
  i3: Lazy[RecordEncoderFields[H]],
  i4: Lazy[NewInstanceExprs[G]],
  i5: ClassTag[F]
): TypedEncoder[F] = ???
```

Primitive types

Recursive types

```
(implicit enc: TypedEncoder[(UUID, UUID)])
```


`implicit val uuidEnc: TypedEncoder[UUID]`	**`implicit def`** `productEnc[Repr <: HList]: TypedEncoder[T]`	`implicit val intEnc: TypedEncoder[Int]`


```
(implicit enc: TypedEncoder[UUID])
```


`implicit val` `uuidEnc: TypedEncoder[UUID]`	`implicit def productEnc[Repr <: HList]: TypedEncoder[T]`	`implicit val intEnc: TypedEncoder[Int]`

DONE

typelevel / frameless

Watch 44 | Unstar 469 | Fork 89

Code | Issues 31 | Pull requests 17 | Projects 0 | Wiki | Insights

Expressive types for Spark.

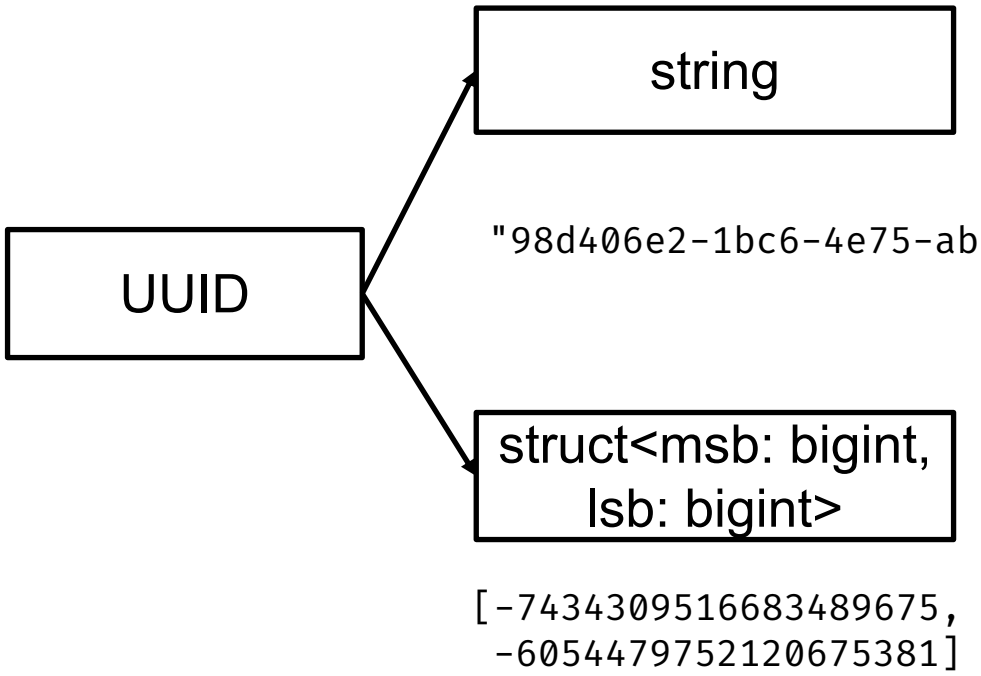
scala spark typelevel fp functional-programming

562 commits | 5 branches | 9 releases | 32 contributors | Apache-2.0

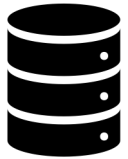
- Typesafe columns referencing
- Enhanced type signature for built-in functions

- Customizable, type safe encoders
  - Not fully compatible semantics ☹

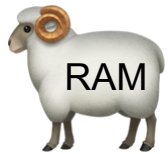
Multiple encoders



- large
- easy to inspect
- needed for compatibility



- 2.25x smaller
- not for humans (or SQL)



Strict parsing

- Spark infers input schema by default
- Specify the schema
  - validation
  - spare inference cost
- Encoder schema can be used
- Parsing extensions

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

val schema = StructType(
  StructField("id", LongType) ::
    StructField("name", StringType) ::
    StructField("gender", StringType) ::
    StructField("age", IntegerType) ::
    Nil
)
val persons = spark
  .read
  .schema(schema)
  .csv("data.csv")
  .as[Person]
```

Strict parsing

- Spark infers input schema by default
- Specify the schema
  - validation
  - spare inference cost
- Encoder schema can be used
- Parsing extensions

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

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

def encoder[T: Encoder] =
  implicitly[Encoder[T]]

val persons = spark.read
  .schema(encoder[Person].schema)
  .csv("data.csv")
  .as[Person]
```

Parsing extensions

- Mostly for flat CSV files
- Nest repeated columns into sequences
- Group columns into objects

```
id | name | gender | age | t1 | t2 | t3 | t4
```

```scala
import ste._

case class ProviderFile(
  @Flatten
  person: Person,
  @Flatten(times=4)
  telephones: Seq[String]
)
```

GitHub: BenFradet/struct-type-encoder

1. operations with performance problems
2. operations with type safety problems
3. ~~creating a dataset of an arbitrary type~~
   - custom encoder framework

typed dataset in

```
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

typelevel / frameless

Watch 44 | Unstar 469 | Fork 89

Code | Issues 31 | Pull requests 17 | Projects 0 | Wiki | Insights

Expressive types for Spark.

scala | spark | typelevel | fp | functional-programming

562 commits | 5 branches | 9 re | e-2.0

- Typesafe columns referencing
- Enhanced type signature for built-in functions

```
filter
select
join
groupBy
orderBy
withColumn ...
```

```scala
case class Person(id: Long, name: String, gender: String, age: Int)

spark.read.parquet("data").as[Person]
  .select($"name", $"age")
  .filter($"age" >= 18)
  .filter($”agd" <= 49)
```

RUNTIME ERROR

```
Name: org.apache.spark.sql.AnalysisException
Message: cannot resolve ‘`agd`' given input
columns: (...)
```

```scala
case class NameAge(name: String, age: Int)

val tds = TypedDataset.create(spark.read.parquet("data").as[Person])
val pTds = tds.project[NameAge]
val fTds = pTds.filter(pTds('age) >= 18)
               .filter(pTds(‘agd) <= 49)
```

COMPILE TIME ERROR

```
error: No column Symbol with shapeless.tag.Tagged[String("agd")]
of type A in NameAge
      .filter(pTds(‘agd) <= 49)
```

```scala
spark.read.parquet("data").as[Person]
  .select($"name", $"age")
  .filter($"age" >= 18)
  .filter($"age" <= 49)
```

```scala
val tds = TypedDataset.create(
  spark.read.parquet("data").as[Person])
val pTds = tds.project[NameAge]
val fTds = pTds.filter(pTds('age) >= 18)
  .filter(pTds(‘age) <= 49)
```

```
== Physical Plan ==
*(1) Project
+- *(1) Filter
   +- *(1) FileScan parquet PushedFilters: [IsNotNull(age),
GreaterThanOrEqual(age,18), LessThanOrEqual(age,49)], ReadSchema:
struct<name:string,age:int>
```

1. operations with performance problems
2. ~~operations with type safety problems~~
   - `frameless.TypedDataset`
3. ~~creating a dataset of an arbitrary type~~
   - custom encoder framework

λs to Catalyst exprs

Compile simple closures to Catalyst expressions [SPARK-14083]

```
ds.map(_.name)                    // ~ df.select($"name")
ds.groupByKey(_.gender)           // ~ df.groupBy($"gender")
ds.filter(_.age > 18)             // ~ df.where($"age" > 18)
ds.map(_.age + 1)                 // ~ df.select($"age" + 1)
```

λs to Catalyst exprs

```
spark.read.parquet("data").as[Person]
  .map(x => NameAge(x.name, x.age))
  .filter(_.age >= 18)
  .filter(_.age <= 49)
```

```
spark.read.parquet("data")
  .select($"name", $"age")
  .filter($"age" >= 18)
  .filter($"age" <= 49)
```

[SPARK-14083]


```
== Physical Plan ==
*(1) Project
+- *(1) Filter
   +- *(1) FileScan parquet PushedFilters: [IsNotNull(age),
GreaterThanOrEqual(age,18), LessThanOrEqual(age,49)], ReadSchema:
struct<name:string,age:int>
```

UDF + tuples = 💥

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

val processPerson = udf((p: Person) => p.name)

providerFile
  .withColumn("results", processPerson($"person"))
  .collect()
```

[SPARK-12823]

RUNTIME ERROR

```
Caused by: java.lang.ClassCastException:
org.apache.spark.sql.catalyst.expressions.GenericRowWithSchema cannot be
cast to Person at (...)
  ... 16 more
```

UDF + tuples

```
import typedudf.TypedUdf
import typedudf.ParamEncoder._

val processPerson = TypedUdf((p : Person) => p.name)

spark.read.parquet("provider").as[ProviderFile]
  .withColumn("results", processPerson($"person"))
  .collect()
```

GitHub: lesbroot/typedudf

1. operations with performance problems
   - only use if no alternatives exist
2. ~~operations with type safety problems~~
   - `frameless.TypedDataset`
3. ~~encoders are not extendable~~
   - custom encoder framework

Takeaways

- RDD -> Spark SQL:
  - offering same level of type safety is hard
- need to dig into Catalyst
- compile time overhead
  - for Spark 2.4 will support Scala 2.12
- check out frameless, etc

Questions

Thank you for listening!

#schema4free

@szdavid92

whitepages®

SEARCH. FIND. KNOW.

https://www.whitepages.com/