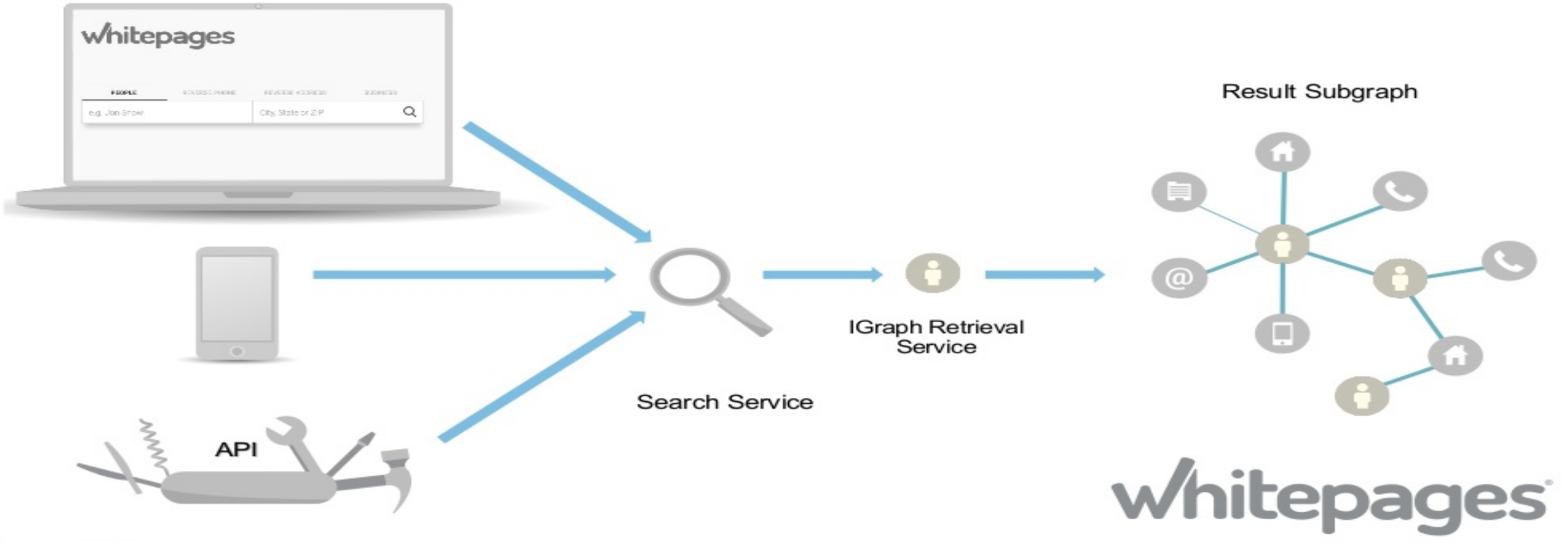
SPARK+AI
SUMMIT EUROPE

Spark Schema for Free

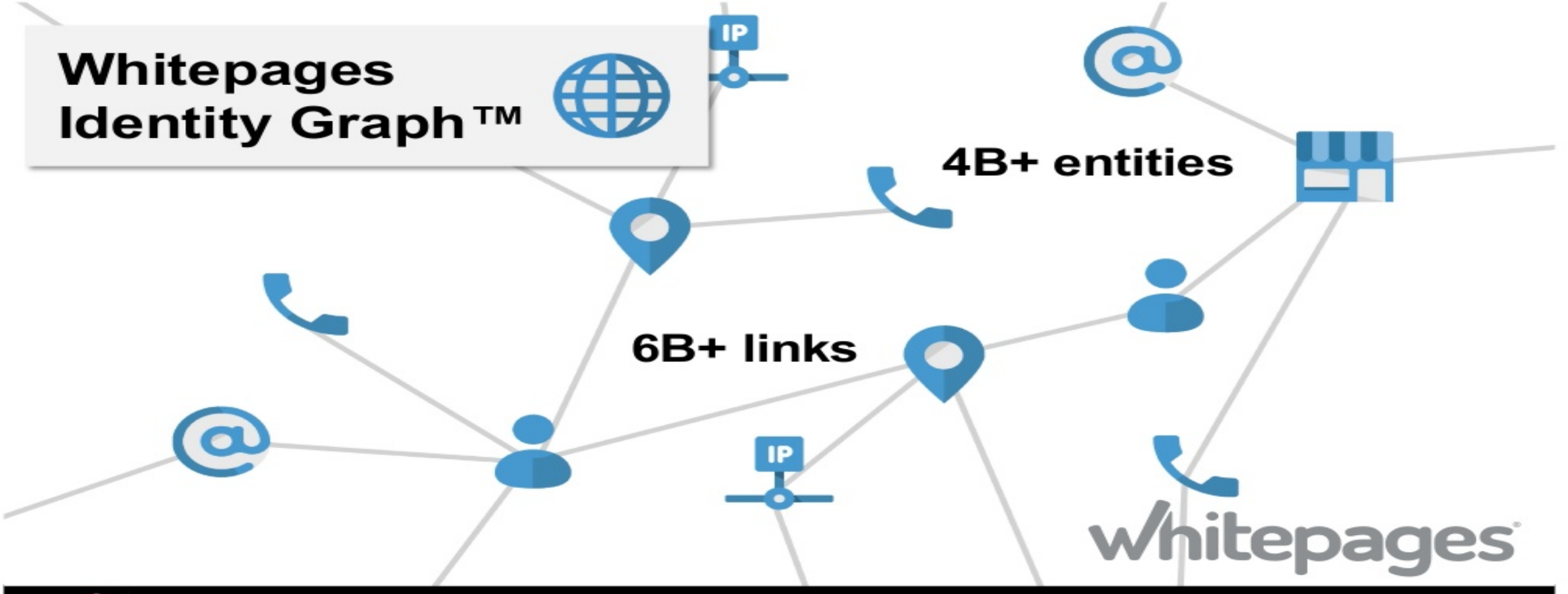
Dávid Szakállas, Whitepages
@szdavid92

Whitepages

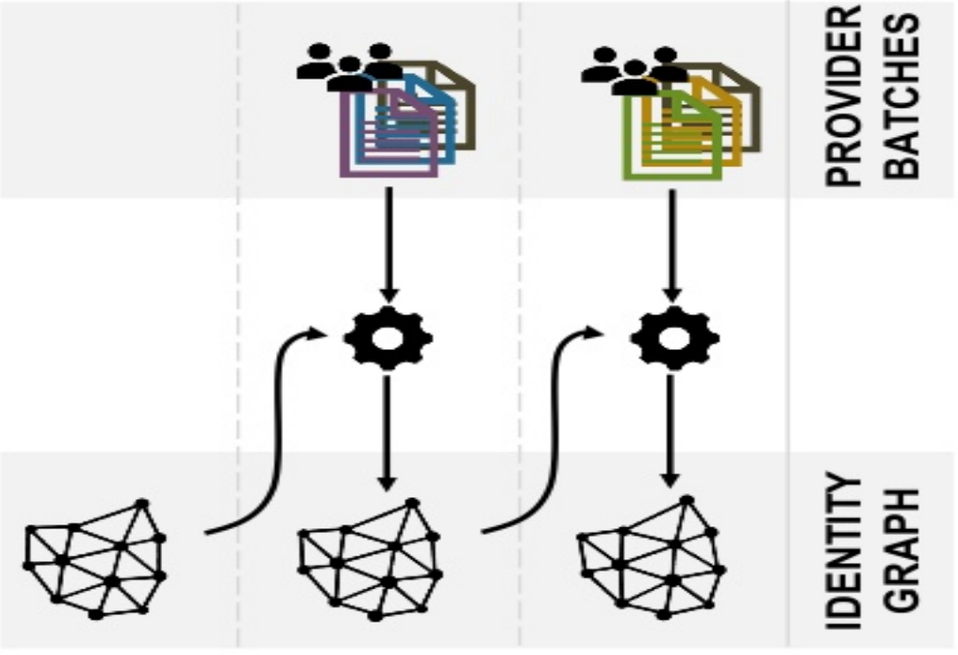
#schema4free


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

Whitepages Identity Graph™



Our story in a nutshell



> 30 000 SLOC

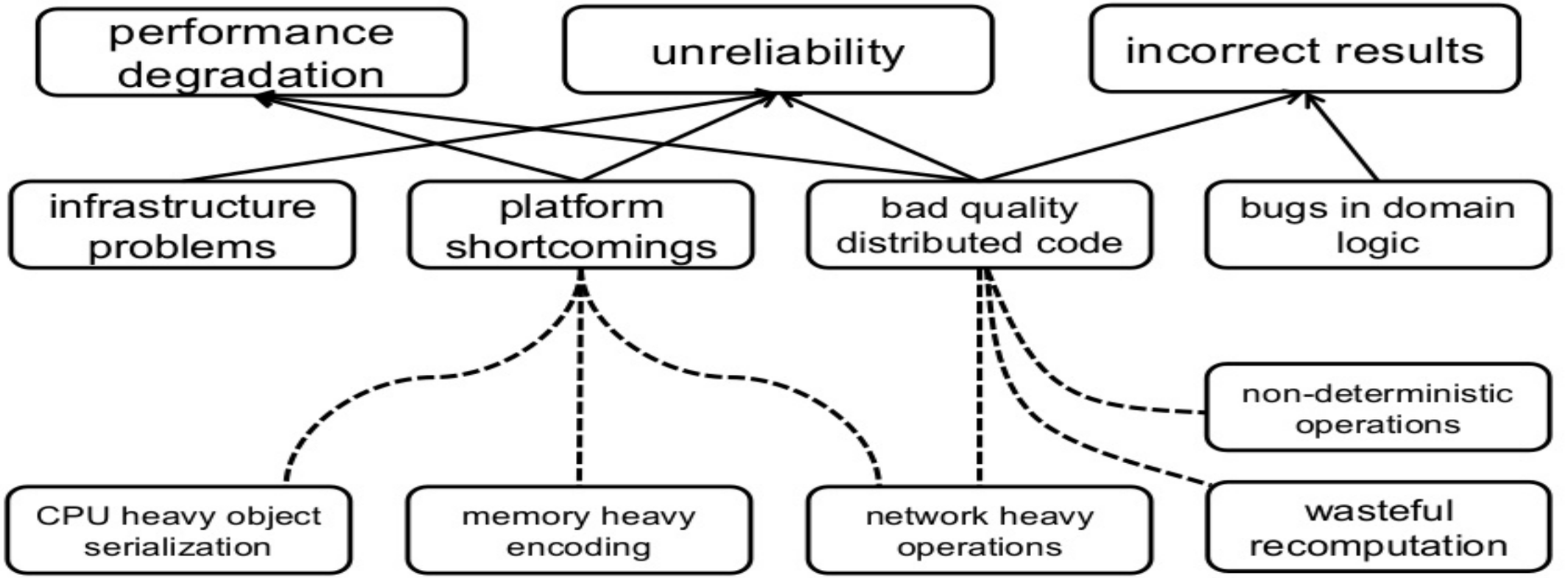
Spark

RDD

3rd PARTY LIBRARIES

RICH DOMAIN TYPES

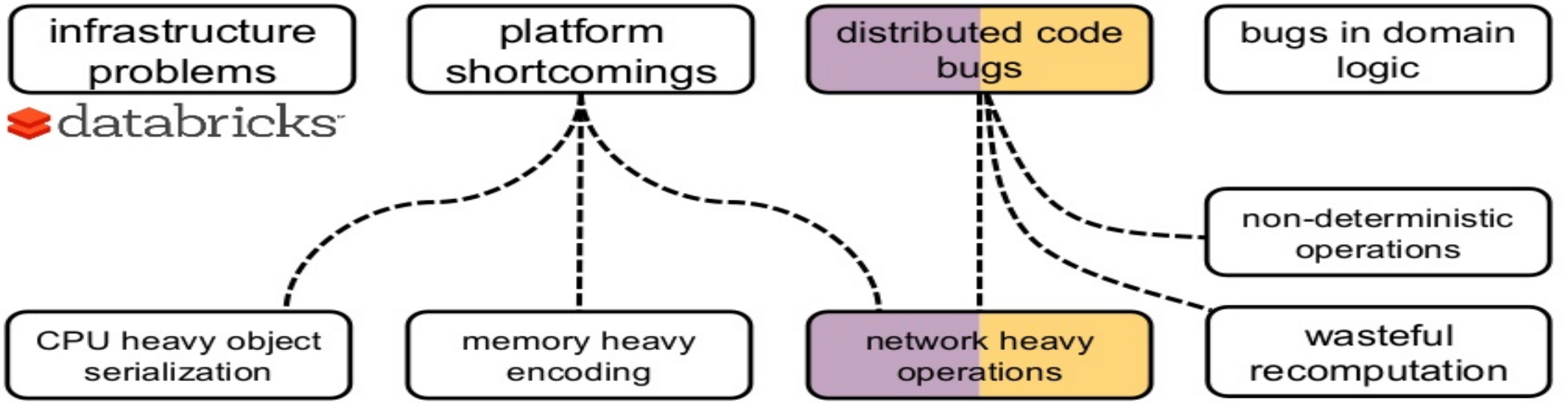
Whitepages


performance degradation
unreliability
incorrect results
infrastructure problems
platform shortcomings
bad quality distributed code
bugs in domain logic
non-deterministic operations
CPU heavy object serialization
memory heavy encoding
network heavy operations
wasteful recomputation

Fixing logic and perf bugs

Moving to Databricks

Switching to Spark SQL



Requirements

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

Two and a half APIs

	Dataset[T]	Dataset[Row]	SQL
	in logic	mistype the col	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	**U**
orderBy, ...	**U**

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

1. operations with performance problems
2. operations with type safety problems
3. encoders are not extendable

```
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[java.util.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", 
- root class: "scala.Tuple2"
  at org.apache.spark.sql.catalyst.ScalaReflectio  (...)
```

Creating extendable encoders

- Types are trees of products, sequences, maps with primitive serializable types as leaves
- Problem similar to JSON serialization
- Idea: use generic programming
  - Generate schema, serializers and deserializers
- Type-level programming with shapeless

```scala
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] = ???
implicit val uuidEncoder: ComposableEncoder[UUID] = ???
// ...
// other primitive types

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

ComposableEncoder[(UUID, UUID)]

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

ComposableEncoder[UUID]

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

- <u>Customizable, type safe encoders</u>
  - Not fully compatible semantics ☹

Multiple encoders

UUID

string

"98d406e2-1bc6-4e75-abfa-2abe530d53cb"

- large
- easy to inspect
- needed for compatibility

struct<msb: bigint, lsb: bigint>

[-7434309516683489675, -6054479752120675381]

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

1. operations with performance problems
2. operations with type safety problems
3. ~~encoders are not extendable~~

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 ☹

```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($"ag" <= 49)
```

RUNTIME ERROR

```
Name: org.apache.spark.sql.AnalysisException
Message: cannot resolve ‘`ag`' 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('ag) <= 49)
```

COMPILE TIME ERROR

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

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

filter
select
project
join
groupBy
orderBy
withColumn ...

1. operations with performance problems
2. ~~operations with type safety problems~~
3. ~~encoders are not extendable~~

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)
```

CURRENTLY

```
== Physical Plan ==
*(1) SerializeFromObject
+- *(1) Filter
   +- *(1) MapElements
      +- *(1) DeserializeToObject
         +- *(1) FileScan parquet PushedFilters: [], ReadSchema:
struct<id:bigint,name:string,gender:string,age:int>
```

λs to Catalyst exprs

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

[SPARK-14083]

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

CURRENTLY

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

Inlining deserialization into λs

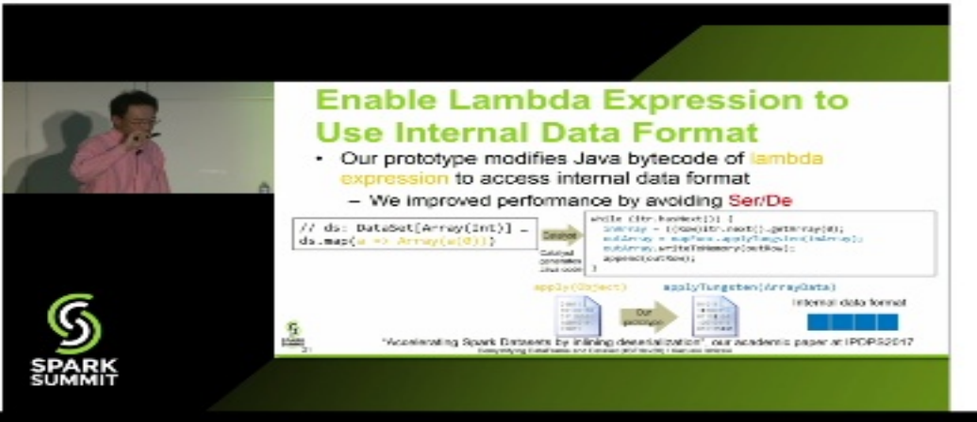
```
ds.filter(  age: Int => age == 20  )
```

```
== Physical Plan ==
*(1) SerializeFromObject
+- *(1) Filter
   +- *(1) DeserializeToObject
      +- *(1) FileScan
```

```
root
|-- id: long (nullable = true)
|-- name: string (nullable = true)
|-- gender: string (nullable = true)
|-- age: integer (nullable = true)
```

J. Wróblewski, K. Ishizaki, H. Inoue and M. Ohara, "Accelerating Spark Datasets by Inlining Deserialization," ... Demystifying DataFrame and Dataset

Analyzed and rewritten on bytecode level



λ → expr

- \+ type safe interface
- \+ catalyst analyzation & optimization
- \+ no refactor needed
- \- only *simple* closures

NOT IMPLEMENTED

param inlining

- \+ type safe interface
- \+ no refactor needed
- \+ more complex closures
- ○ only elides deser cost
- \- no optimization of body

NOT IMPLEMENTED

Requirements

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

Miscellaneous features

Strict parsing

- Spark infers input schema by default
- Specify the schema
  - validation
  - spare inference cost
- Schema could be generated
- Parsing extensions

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

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

implicit def encoder[T: TypedEncoder] =
  TypedExpressionEncoder[T]

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

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

Takeaways

- RDD -> Spark SQL: hard work
  - (with our requirements)
- needed to dig into Catalyst
- compile time overhead
  - Scala 2.12
- check out frameless, etc.

Whitepages

Questions

Thank you for listening!

#schema4free
@szdavid92

Whitepages®
SEARCH. FIND. KNOW.

https://www.whitepages.com/