ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Structure and Optimization

Big Data Analysis with Scala and Spark

Heather Miller

Example: Selecting Scholarship Recipients

Let's imagine that we are an organization, CodeAward, offering scholarships to programmers who have overcome adversity. Let's say we have the following two datasets.

```scala
case class Demographic(id: Int,
                       age: Int,
                       codingBootcamp: Boolean,
                       country: String,
                       gender: String,
                       isEthnicMinority: Boolean,
                       servedInMilitary: Boolean)
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)

case class Finances(id: Int,
                    hasDebt: Boolean,
                    hasFinancialDependents: Boolean,
                    hasStudentLoans: Boolean,
                    income: Int)
val finances = sc.textfile(...)... // Pair RDD, (id, finances)
```

Example: Selecting Scholarship Recipients

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

Example: Selecting Scholarship Recipients

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- Swiss students
- who have debt & financial dependents

How might we implement this Spark program?

```
// Remember, RDDs available to us:
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)
val finances = sc.textfile(...)...     // Pair RDD, (id, finances)
```

Example: Selecting Scholarship Recipients

Possibility 1:

(Int, (Demographic, Finances))
id

```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

Example: Selecting Scholarship Recipients

Possibility 1:

```
demographics.join(finances)
            .filter { p =>
              p._2._1.country == "Switzerland" &&
              p._2._2.hasFinancialDependents &&
              p._2._2.hasDebt
            }.count
```

Steps:

1. Inner join first
2. Filter to select people in Switzerland
3. Filter to select people with debt & financial dependents

Example: Selecting Scholarship Recipients

Possibility 2:

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == ”Switzerland”)
            .join(filtered)
            .count
```

Example: Selecting Scholarship Recipients

Possibility 2:

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == ”Switzerland”)
            .join(filtered)
            .count
```

Steps:

1. Filter down the dataset first (look at only people with debt & financial dependents)
2. Filter to select people in Switzerland (look at only people in Switzerland)
3. Inner join on smaller, filtered down dataset

Example: Selecting Scholarship Recipients

Possibility 3:

```
val cartesian
  = demographics.cartesian(~~demographics~~ finances)

cartesian.filter {
  case (p1, p2) => p1._1 == p2._1    ← same ids
}
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
                   (p2._2.hasFinancialDependents) &&
                   (p2._2.hasDebt)
}.count
```

Example: Selecting Scholarship Recipients

Possibility 3:

```
val cartesian                        finances
  = demographics.cartesian(~~demographics~~)

cartesian.filter {
  case (p1, p2) => p1._1 == p2._1
}
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
                   (p2._2.hasFinancialDependents) &&
                   (p2._2.hasDebt)
}.count
```

Steps:

1. Cartesian product on both datasets
2. Filter to select resulting of cartesian with same IDs
3. Filter to select people in Switzerland who have debt and financial dependents

Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

150,000 people

Possibility 1

```
> ds.join(fs)
    .filter(p => p._2._
    .count
```

▸ (1) Spark Jobs

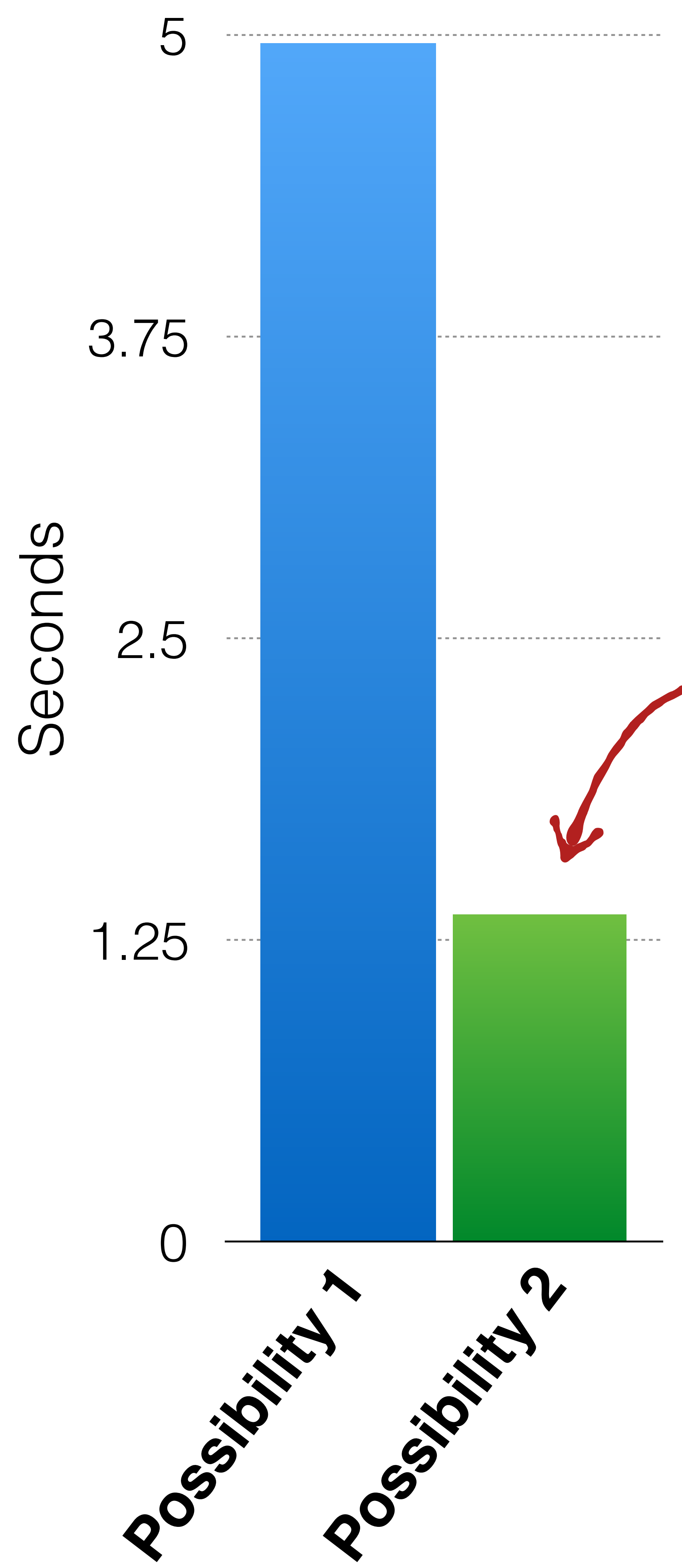
```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2.
    .join(fsi)
    .count
```

▸ (1) Spark Jobs

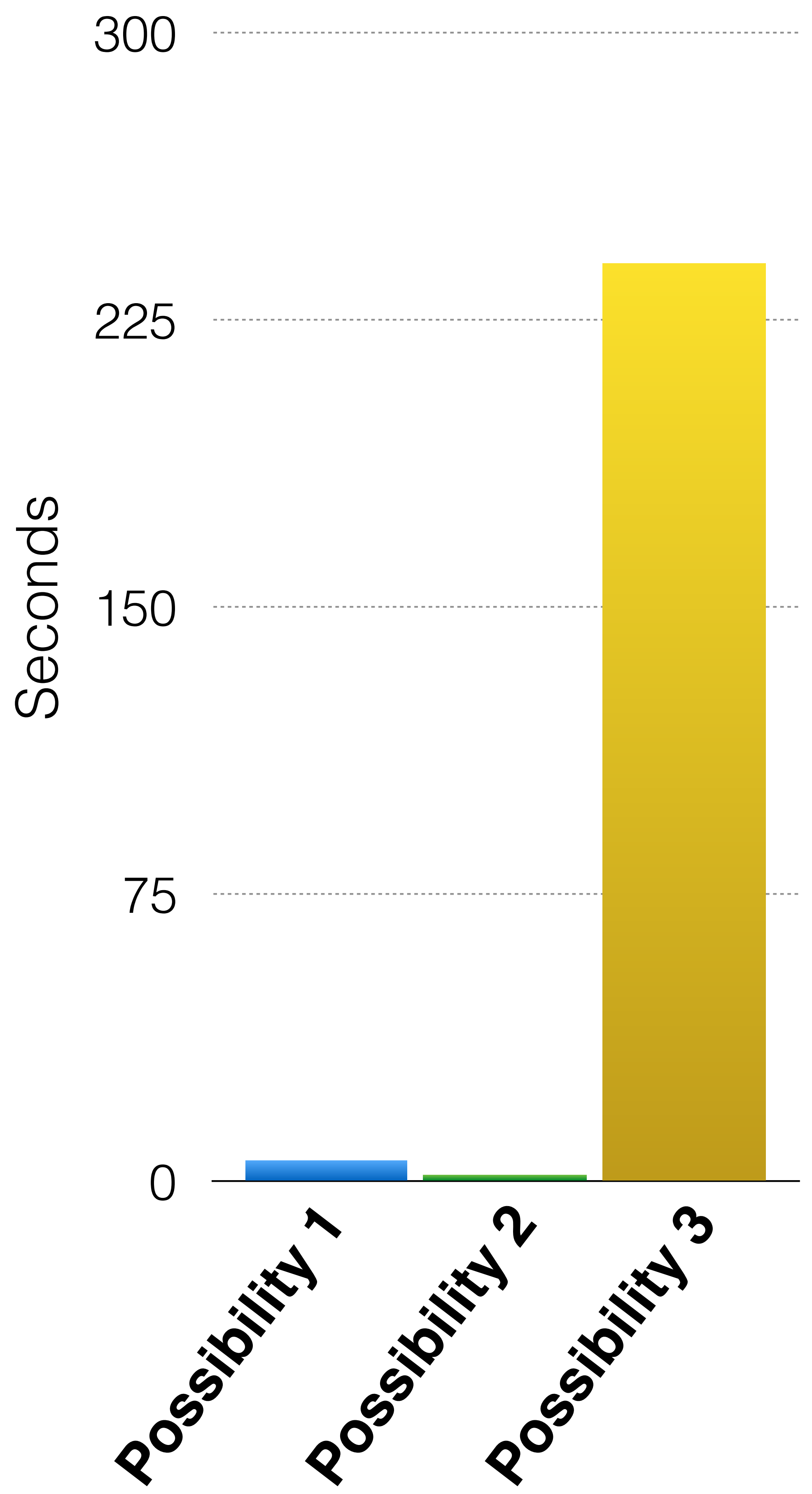
```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds
```



Filtering data first is 3.6x faster!

Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



177x slower!

~4 mins to complete, versus 1.35 and 4.97 seconds

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

Given a bit of extra structural information, Spark can do many optimizations *for* you!

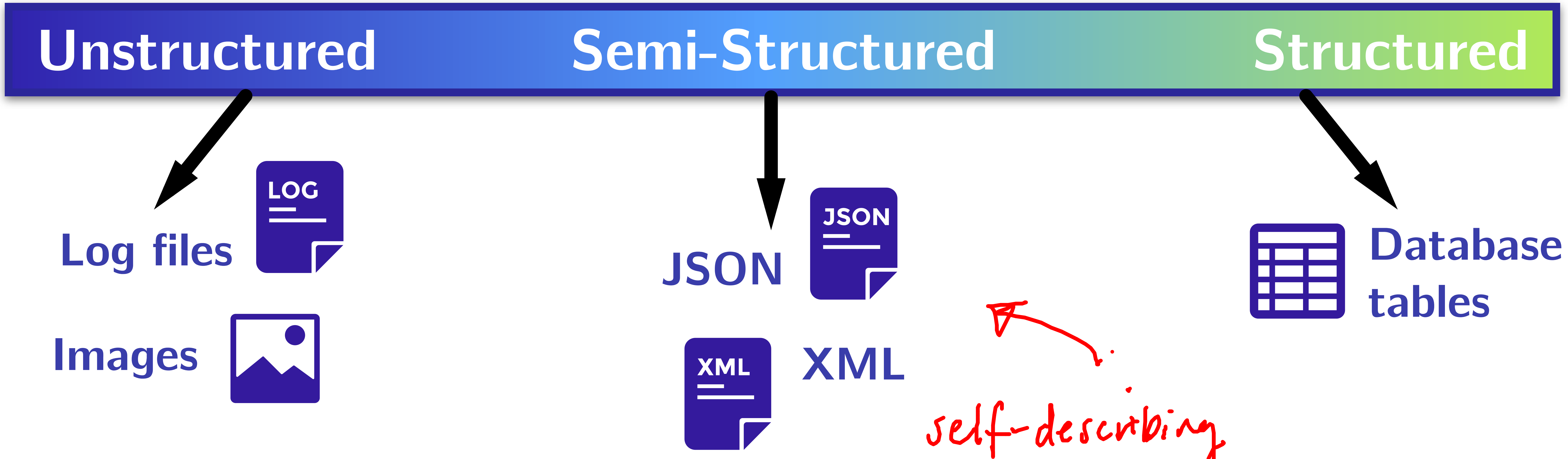
Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.

Unstructured **Semi-Structured** **Structured**

Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.



self-describing

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the ***schema*** of the data it's dealing with.

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the ***schema*** of the data it's dealing with.

Given an arbitrary RDD, Spark knows that the RDD is parameterized with arbitrary types such as,

- `Person`
- `Account`
- `Demographic`

but it doesn't know anything about these types' structure.

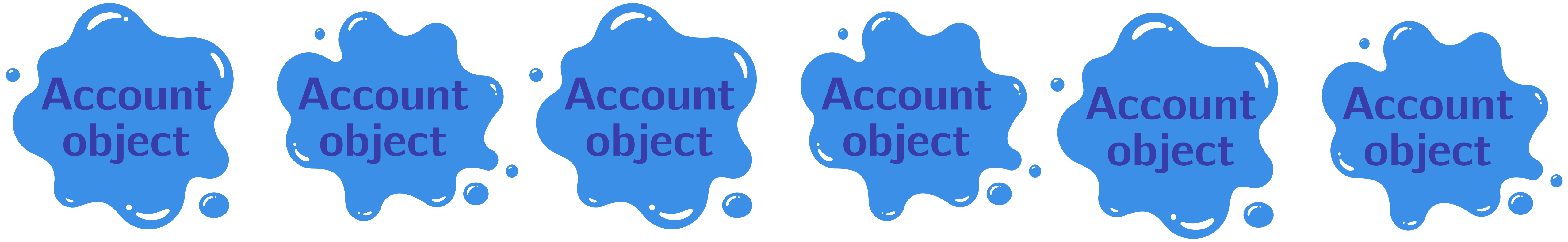
Structured Data vs RDDs

Assuming we have a dataset of **Account** objects:

```scala
case class Account(name: String, balance: Double, risk: Boolean)
```

RDD[Account]

Spark/RDDs see:



Blobs of objects we know nothing about, except that they're called **Account**.

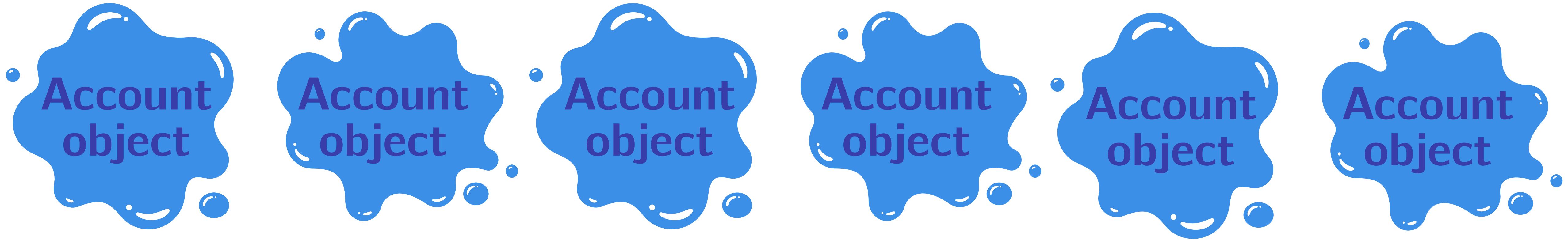
Spark can't see inside this object or analyze how it may be used, and to optimize based on that usage. It's opaque.

Structured Data vs RDDs

Assuming we have a dataset of **`Account`** objects:

```scala
case class Account(name: String, balance: Double, risk: Boolean)
```

Spark/RDDs see:



A database/Hive sees:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

Columns of named and typed values.

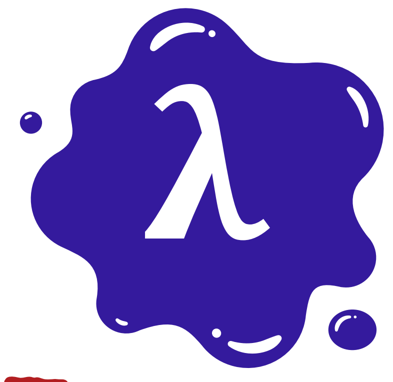
If Spark could see data this way, it could break up and only select the datatypes it needs to send around the cluster.

Structured vs Unstructured Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.



Like the data Spark operates on, function literals too are completely opaque to Spark.

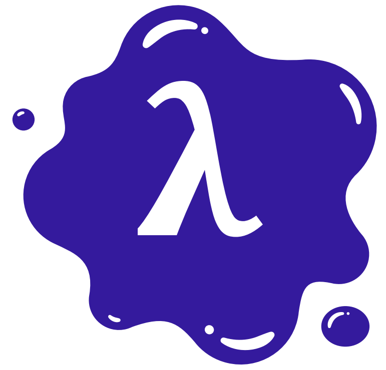
A user can do anything inside of one of these, and all Spark can see is something like: **`$anon$1@604f1a67`**

Structured vs Unstructured Computation

The same can be said about ***computation***.

In Spark:

- We do **functional transformations** on data.
- We pass user-defined function literals to higher-order functions like **`map`**, **`flatMap`**, and **`filter`**.



In a database/Hive:

- We do **declarative transformations** on data.
- Specialized/structured, pre-defined operations.

Fixed set of operations, fixed set of types they operate on.

Optimizations the norm!

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*

Not much structure.

Difficult to aggressively optimize.

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*

Not much structure.

Difficult to aggressively optimize.

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

```
SELECT
WHERE
ORDER BY
GROUP BY
COUNT
```

Lots of structure.

Lots of optimization opportunities!

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

Spark SQL makes this possible!

We've got to give up some of the freedom, flexibility, and generality of the functional collections API in order to give Spark more opportunities to optimize though.