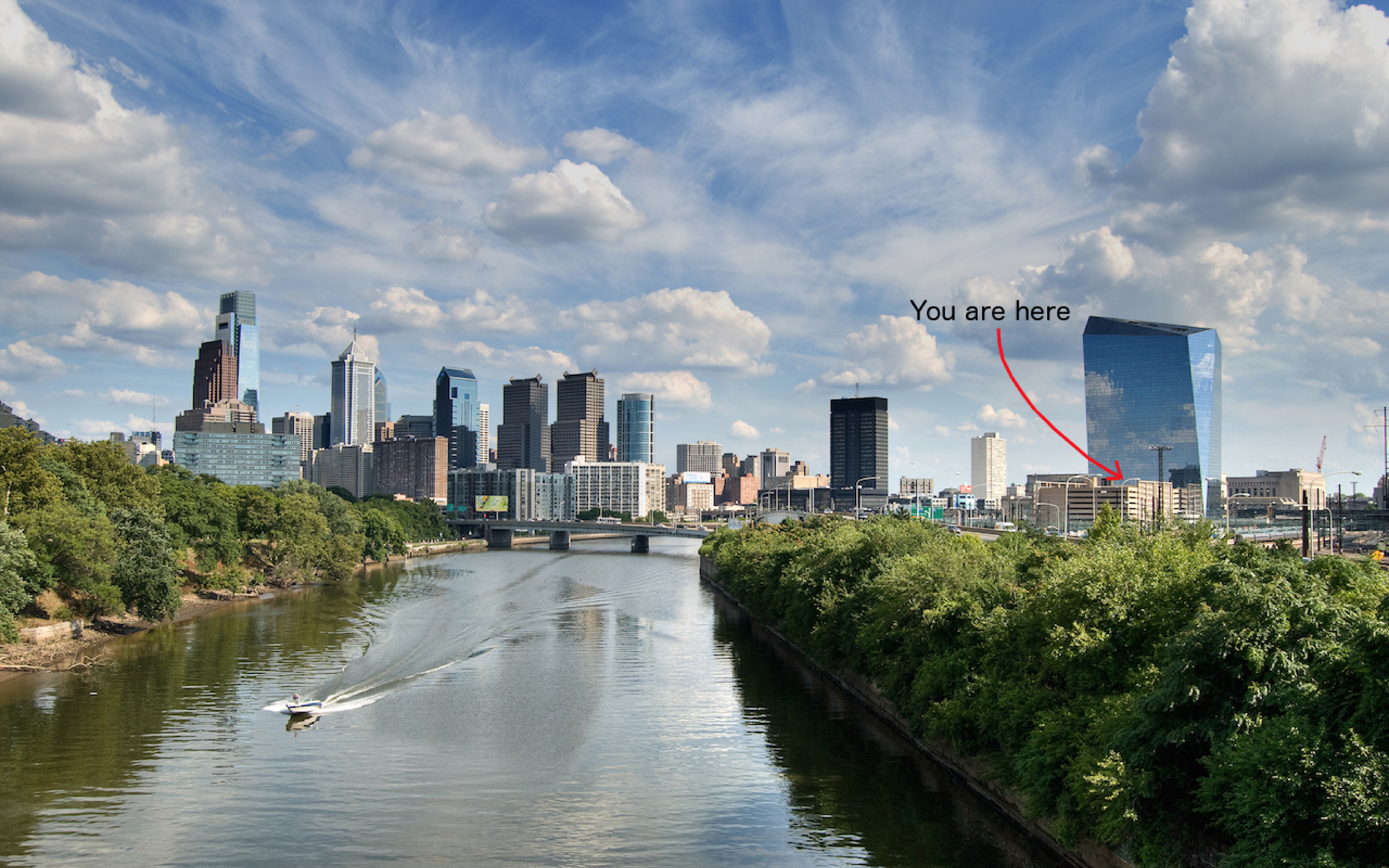

You are here

RDDs, DataFrames and Datasets in Apache Spark

Brian Clapper, *@brianclapper*

What we'll be discussing

Three Spark APIs

- Resilient Distributed Datasets (RDDs)
- DataFrames
- Datasets

RDDs are...

... compile-time type-safe

... lazy.

... based on the Scala collections API, so the operations are familiar to Scala programmers.

So much so, in fact, that it can be confusing to new users:

```
def collect[U](f: PartialFunction[T, U])(implicit arg0: ClassTag[U]): RDD[U]
```

Return a new RDD that contains all matching values by applying `f`.

```
def collect(): Array[T]
```

Return an array that contains all of the elements in this RDD.

RDDs: Some code

This blob of code creates an RDD from a file on a distributed file system.

```
val rdd = sc.textFile("hdfs:/user/bmc/wikipedia-pagecounts.gz")
val parsedRDD = rdd.flatMap { line =>
  line.split("""\s+""") match {
    case Array(project, _, numRequests, _) => Some((project, numRequests))
    case _ => None
  }
}
parsedRDD.filter { case (project, numRequest) => project == "en" }.
          reduceByKey(_ + _).
          take(100).
          foreach { case (project, requests) => println(s"$project: $requests") }
```

RDDs

RDDs are type-safe. However, they're also low-level, and they suffer from some problems, including:

- They express the *how* of a solution better than the *what*.
- They cannot be optimized by Spark.
- They're *slow* on non-JVM languages like Python.
- It's too easy to build an inefficient RDD transformation chain.

```
parsedRDD.reduceByKey(_ + _).                                    <--- INEFFICIENT
          filter { case (project, numRequest) => project == "en" }. <--- ORDERING
          take(100).
          foreach { case (project, requests) => println(s"project: $requests") }
```

The DataFrame API

The DataFrame API provides a higher-level abstraction (a DSL, really), allowing you to use a query language to manipulate data. In fact, you can use SQL, as well.

This code does essentially the same thing the previous RDD code does. Look how much easier it is to read.

```
val df = parsedRDD.toDF("project", "numRequests")
df.groupBy($"page").
   agg(sum($"numRequests").as("count")).
   limit(100).
   show(100)
```

The DataFrame API

SQL code

Here's the same thing in SQL.

```
df.registerTempTable("edits")
sqlContext.sql("SELECT project, sum(numRequests) AS count FROM edits LIMIT 100").show
```

DataFrame queries are optimized

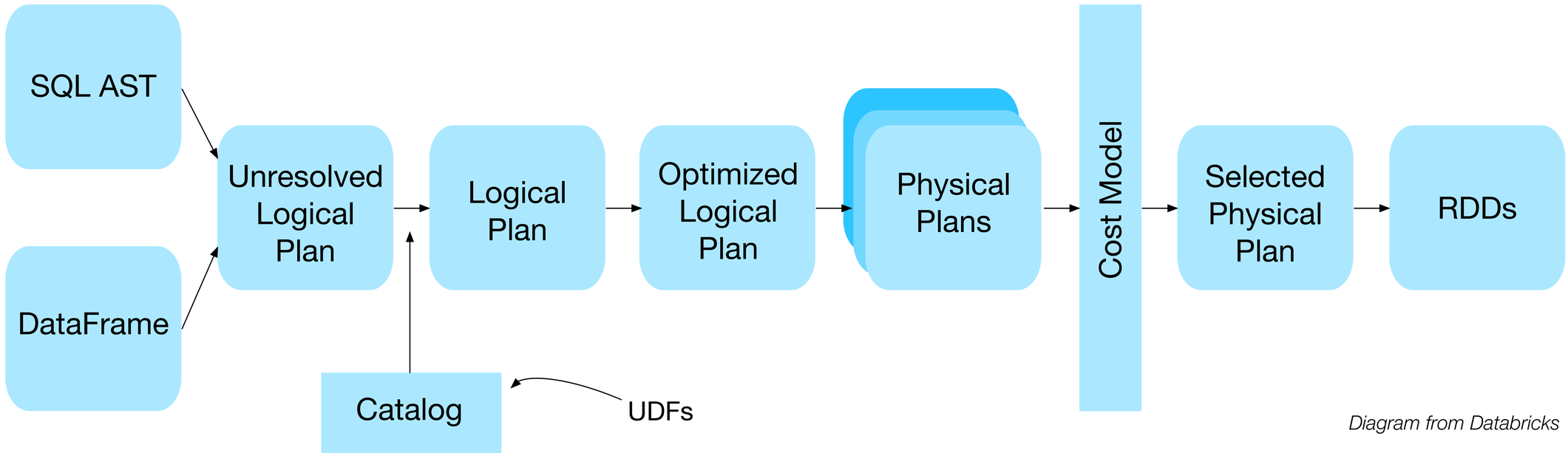
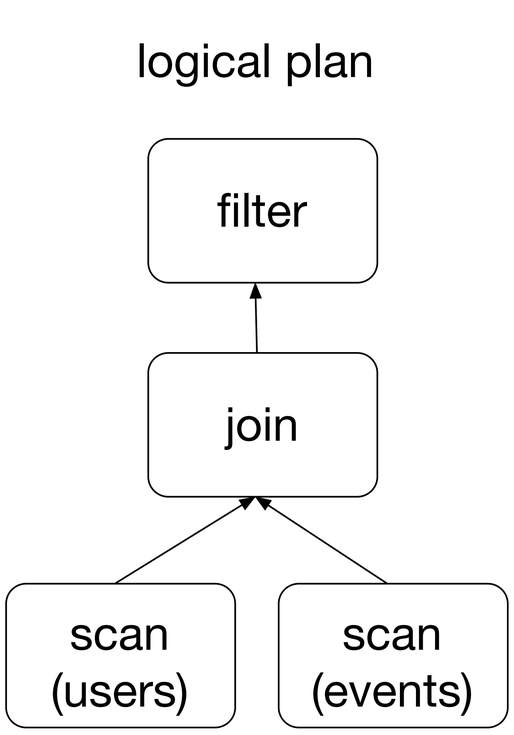
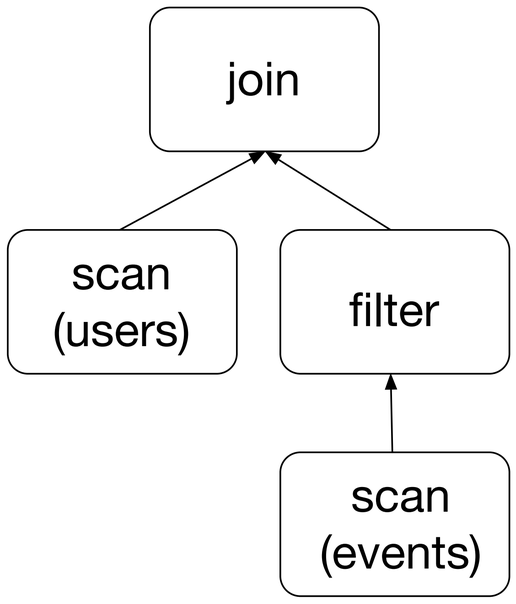


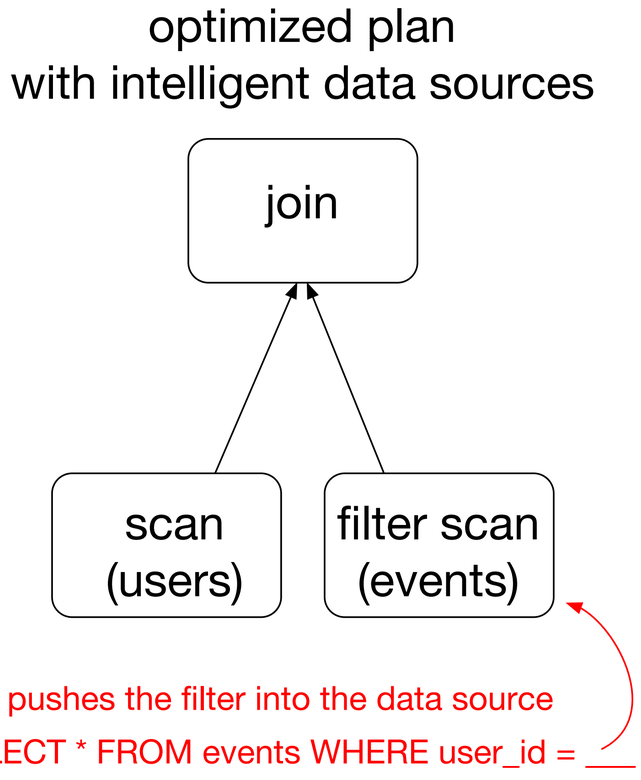
Diagram from Databricks

Example Optimization

```
users.join(events, users("id") === events("uid"))
     .filter(events("date") > "2015-01-01")
```

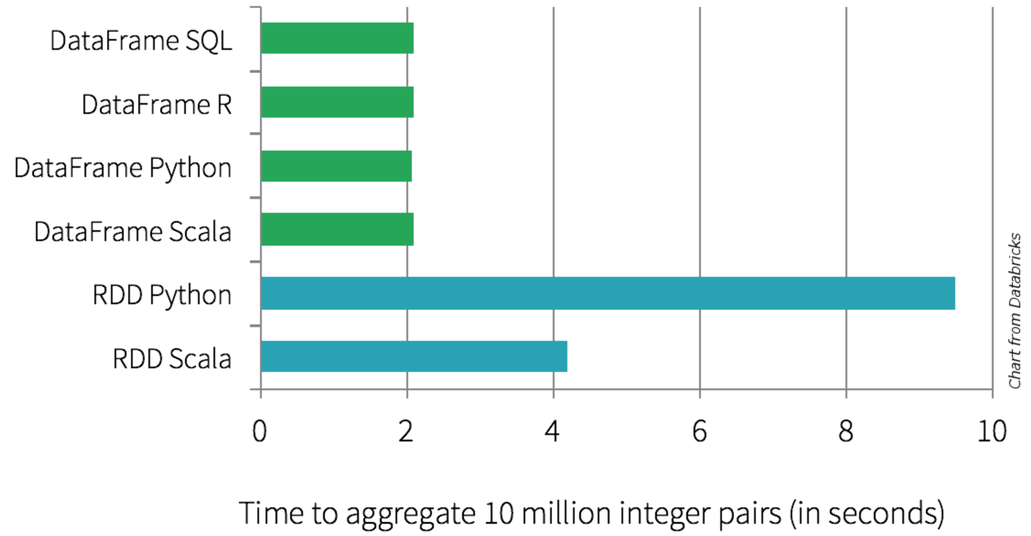






DataFrames are faster

Because of the optimization, they tend to outperform RDDs.



Time to aggregate 10 million integer pairs (in seconds)

But we've lost type safety

What happens if we call the `collect()` action?

```
scala> :type df.collect()
Array[org.apache.spark.sql.Row]
```

Unfortunately, `Row` isn't typesafe. It's defined as

```
trait Row extends Serializable
```

Mapping it back to something useful is ugly and error-prone:

```
df.collect().map { row =>
  val project = row(0).asInstanceOf[String]   // Yuck.
  val numRequests = row(1).asInstanceOf[Long] // Yuck.
}
```

What do we want?

We'd like to get back our compile-time type safety, *without* giving up the optimizations Catalyst can provide us.

Enter Datasets

Datasets are:

- An extension to the DataFrame API
- Conceptually similar to RDDs. (You can use lambdas and types again.)
- Use Tungsten's fast in-memory encoding (as opposed to JVM objects or serialized objects on the heap)
- Expose expressions and fields to the DataFrame query planner, where the optimizer can use them to make decisions. (This can't happen with RDDs.)
- Interoperate more easily with the DataFrame API
- Available in Spark 1.6 as an **experimental** API preview. (They're a development focus for the next several Spark versions.)

Enter Datasets

Like an RDD, Dataset has a *type*.

```
// Read a DataFrame from a JSON file
val df = sqlContext.read.json("people.json")
// Convert the data to a domain object.
case class Person(name: String, age: Long)
val ds: Dataset[Person] = df.as[Person]
```

A DataFrame is just a `Dataset[Row]`.

Datasets: A bit of both

With Datasets, you can still access a DataFrame-like query API. (You can also go back and forth between DataFrames and Datasets.)

RDDs:

```
val lines = sc.textFile("hdfs://path/to/some/ebook.txt")
val words = lines.flatMap(_.split("""\s+""")).filter(_.nonEmpty)
val counts = words.groupBy(_.toLowerCase).map { case (w, all) => (w, all.size) }
```

Datasets:

```
val lines = sqlContext.read.text("hdfs://path/to/some/ebook.txt").as[String]
val words = lines.flatMap(_.split("""\s+""")).filter(_.nonEmpty)
val counts = words.groupBy(_.toLowerCase).count()
```

Datasets: A bit of both

```
// RDD
val counts = words.groupBy(_.toLowerCase).map { case (w, all) => (w, all.size) }
// Dataset
val counts = words.groupBy(_.toLowerCase).count()
```

The Dataset version can use the built-in DataFrame-like `count()` aggregator function.

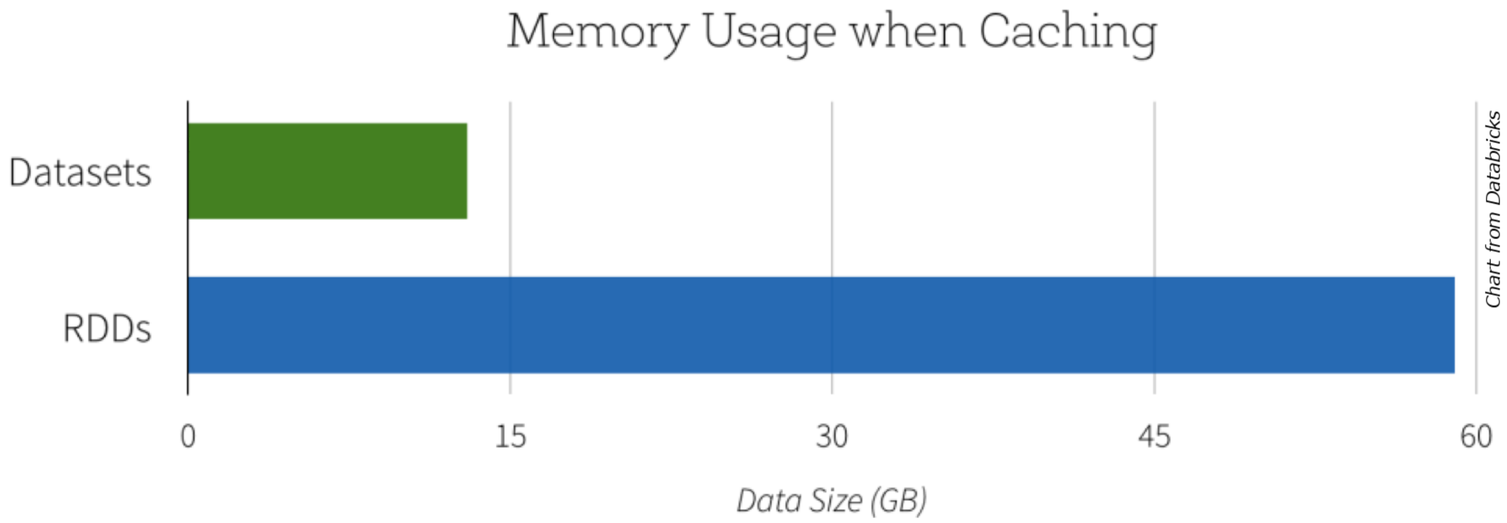
The Dataset code is more visually compact (less typing! yay!) *and* will tend to execute faster than the RDD counterpart.

Datasets and Memory

Datasets tend to use less memory.

- Spark understands the structure of data in Datasets, because they're *typed*.
- Spark uses *encoders* to translate between these types ("domain objects") and Spark's compact internal Tungsten data format.
- It generates these encoders via runtime code-generation. The generated code can operate *directly* on the Tungsten compact format.
- Memory is conserved, because of the compact format. Speed is improved by custom code-generation.

Space Efficiency

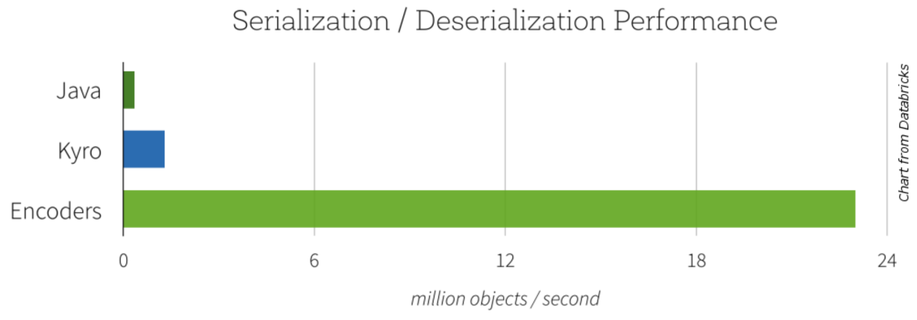


Datasets and Serialization

Spark has to serialize data ... a *lot*.

- Because of the efficiency of the code-generated encoders, serialization can be significantly faster than either native Java or Kryo serialization.
- The resulting serialized data will often be up to 2x smaller, as well, which reduces disk use and network use.

Datasets and Serialization



Some Dataset Limitations

- They're *experimental*, which means the APIs might change in upcoming Spark versions.
- The APIs are currently incomplete. For example, the Dataset API lacks some aggregators (like `sum()`) and lacks a `sortBy()` function.
- The term "Datasets" is kind of hard to Google...

Enough, already...

Let's look at some code.

More resources

This presentation: HTML (Reveal.js), PDF, Reveal.js and demo sourcea

Introducing Spark Datasets (Databricks blog post):
`https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html`

Spark SQL, DataFrames and Datasets Guide:
`http://spark.apache.org/docs/latest/sql-programming-guide.html`