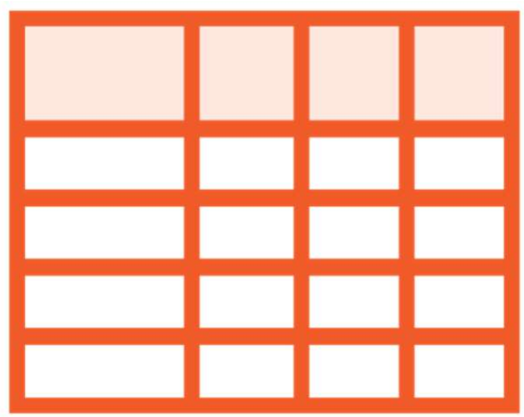
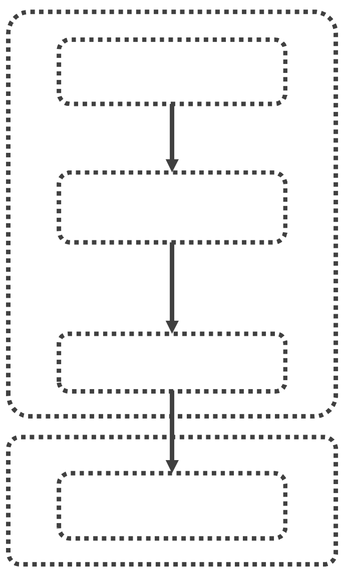
Understanding a Typed API: Datasets

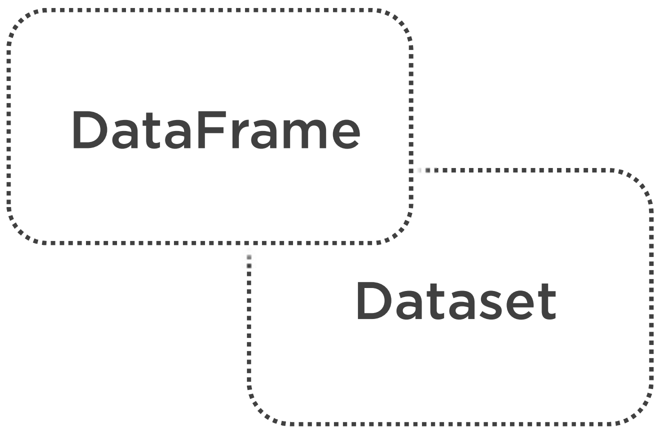
Xavier Morera
HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA
@xmorera www.xaviermorera.com

Understanding a Typed API: Datasets



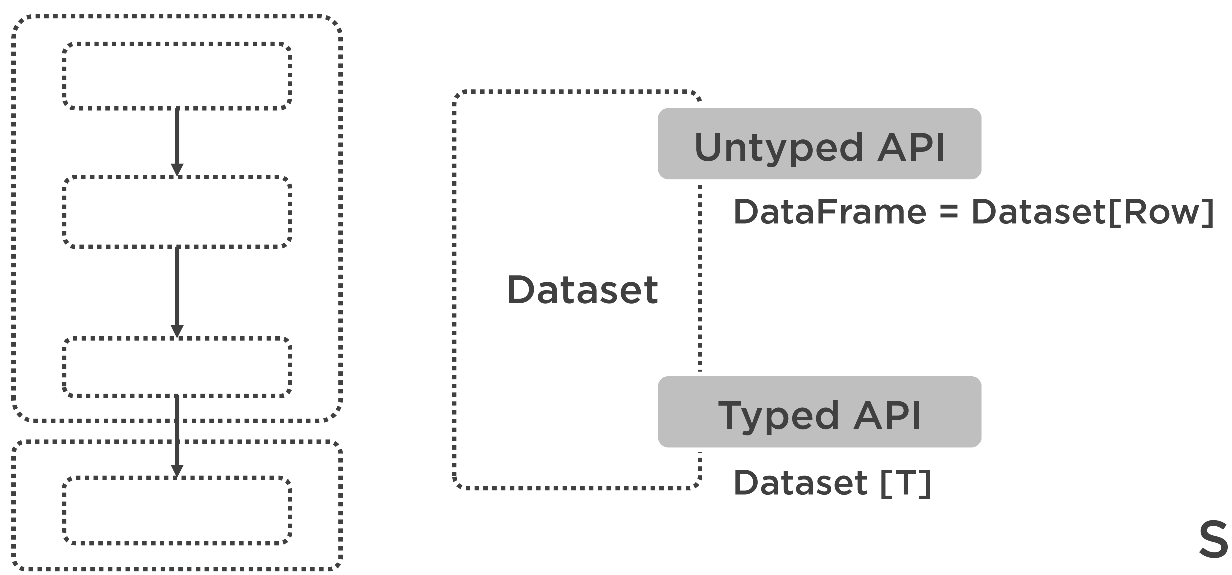
Understanding a Typed API: Datasets





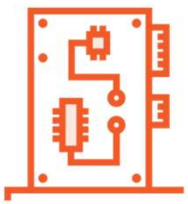
Since 1.x

Understanding a Typed API: Datasets



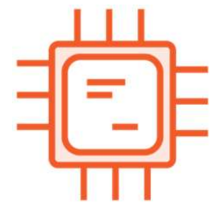
Since 2.0

The Motivation Behind Datasets



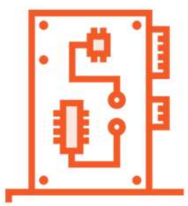
Network

Storage



CPU

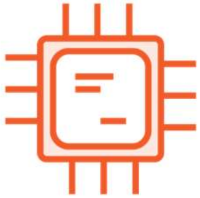
The Motivation Behind Datasets



Network

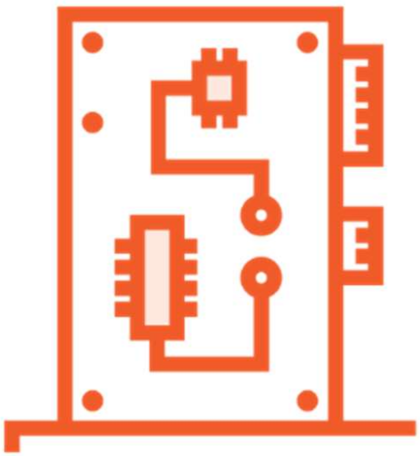
1 GBPS

Storage



CPU

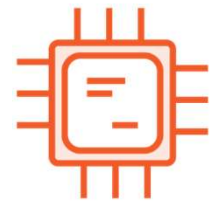
The Motivation Behind Datasets



Network

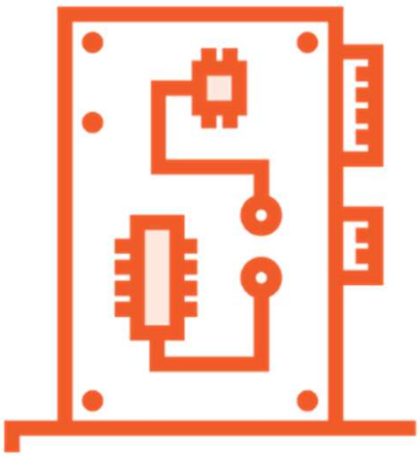
10 Gbps

Storage



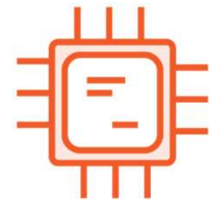
CPU

The Motivation Behind Datasets



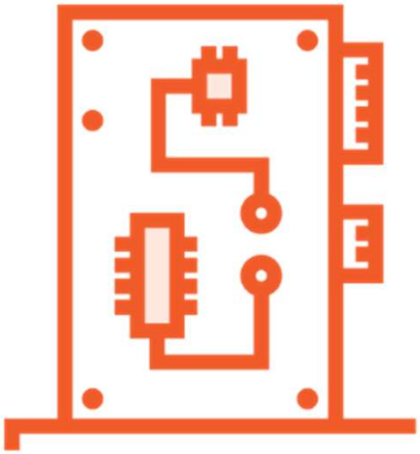
Network
10 Gbps

Storage
50 Mbps

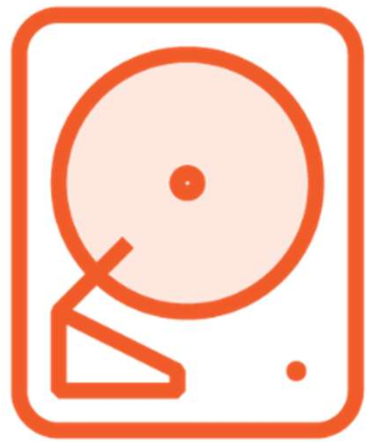


CPU

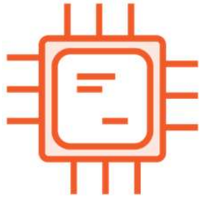
The Motivation Behind Datasets



Network
10 Gbps

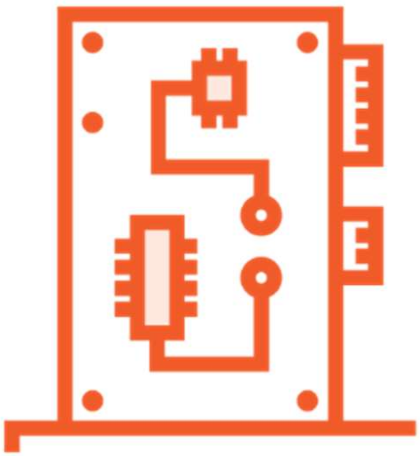


Storage
500 Mbps

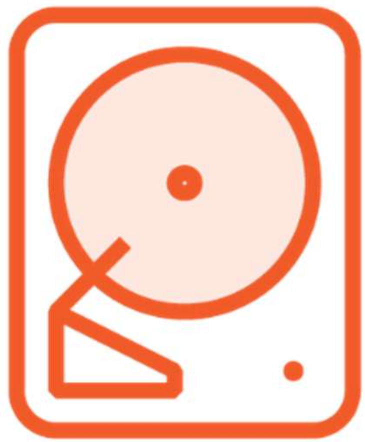


CPU
3 Ghz

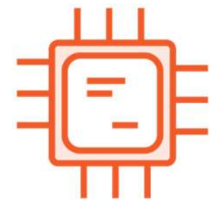
The Motivation Behind Datasets



Network
10 Gbps



Storage
500 Mbps



CPU
3 Ghz

The Motivation Behind Datasets

Network
10 Gbps

Storage
500 Mbps

CPU
3 Ghz

```
val postsRDD =
sc.textFile("/user/cloudera/stackexchange/simple_titles_txt")

val postsDS =
spark.read.text("/user/cloudera/stackexchange/simple_titles_txt").
  as[String]
postsRDD.setName("postsRDD")

val postsDF =
spark.read.text("/user/cloudera/stackexchange/simple_titles_txt")
```

Size Matters

Create an RDD and Dataset

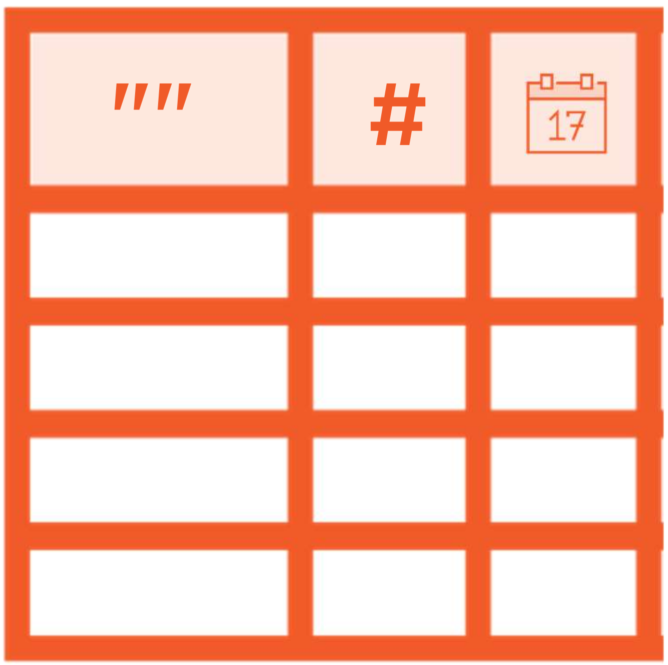
- Cache and count
- Same with a DataFrame

Think beyond MBs, imagine TB or PB

```
spark.sql("Selct Score from PostsSE").show(5)

spark.sql("Select Scre from PostsSE").show(5)

postsDF.selct("Score").show(5)

postsDF.select($"Scre").first

postsDS.first.Scre

val firstScore: String = postsDS.first.Score
```

	SQL	DataFrames	Datasets
Syntax errors			
Analysis errors			

What's a Dataset



Dataset

- Strongly typed collection of objects
- Domain specific objects
- Type safe
- Functional programming
- Query optimization

You know the API

- More than just Row

What Can You Store in a Dataset?

.as[Post]

.toDS()

Encoder

- JVM objects <-> Spark's internal representation

Supported objects

- Primitive types
- Complex types
- Product objects
- Row objects

DataFrames

Remember Row objects?

DataFrame - org.apache

Not secure | spark.apache.org/docs/1.6.3/api/scala/index.html#org.apache.spark.sql.DataFrame

dataframe

#ABCDEFGHIJKLMNOPQRSTUVWXYZ

display packages only

org.apache.spark.sql hide focus

- DataFrame
- DataFrameHolder
- DataFrameNaFunctions
- DataFrameReader
- DataFrameStatFunctions
- DataFrameWriter

org.apache.spark.sql

DataFrame

```
class DataFrame extends Queryable with Serializable
```

A distributed collection of data organized into named columns.

Experimental

A DataFrame is equivalent to a relational table in Spark SQL. The following example creates a DataFrame by pointing Spark SQL to a Parquet data set.

```
val people = sqlContext.read.parquet("...")  // in Scala
DataFrame people = sqlContext.read().parquet("...")  // in Java
```

Once created, it can be manipulated using the various domain-specific-language (DSL) functions defined in: DataFrame (this class), Column, and functions.

To select a column from the data frame, use `apply` method in Scala and `col` in Java.

```
val ageCol = people("age")  // in Scala
Column ageCol = people.col("age")  // in Java
```

Note that the Column type can also be manipulated through its various functions.

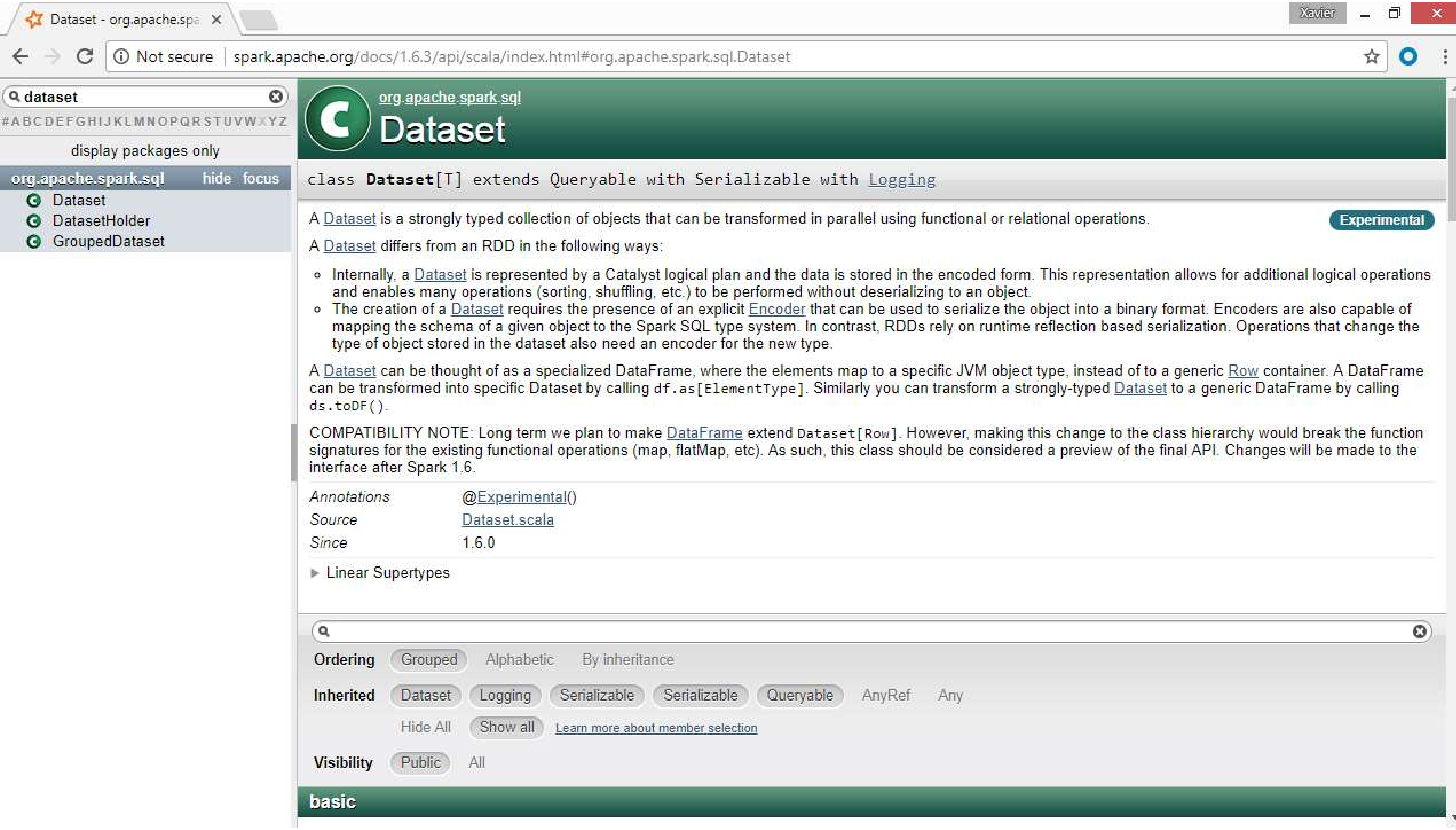
```
// The following creates a new column that increases everybody's age by 10.
people("age") + 10  // in Scala
people.col("age").plus(10);  // in Java
```

A more concrete example in Scala:

```
// To create DataFrame using SQLContext
val people = sqlContext.read.parquet("...")
val department = sqlContext.read.parquet("...")

people.filter("age > 30")
  .join(department, people("deptId") === department("id"))
  .groupBy(department("name"), "gender")
  .agg(avg(people("salary")), max(people("age")))
```

and in Java:

```
// To create DataFrame using SQLContext
DataFrame people = sqlContext.read().parquet("...");
DataFrame department = sqlContext.read().parquet("...");
```



Dataset - org.apache.spark.sql

Not secure | spark.apache.org/docs/1.6.3/api/scala/index.html#org.apache.spark.sql.Dataset

dataset

#ABCDEFGHIJKLMNOPQRSTUVWXYZ

display packages only

org.apache.spark.sql hide focus

- Dataset
- DatasetHolder
- GroupedDataset

org.apache.spark.sql

Dataset

```
class Dataset[T] extends Queryable with Serializable with Logging
```

Experimental

A Dataset is a strongly typed collection of objects that can be transformed in parallel using functional or relational operations.

A Dataset differs from an RDD in the following ways:

- Internally, a Dataset is represented by a Catalyst logical plan and the data is stored in the encoded form. This representation allows for additional logical operations and enables many operations (sorting, shuffling, etc.) to be performed without deserializing to an object.
- The creation of a Dataset requires the presence of an explicit Encoder that can be used to serialize the object into a binary format. Encoders are also capable of mapping the schema of a given object to the Spark SQL type system. In contrast, RDDs rely on runtime reflection based serialization. Operations that change the type of object stored in the dataset also need an encoder for the new type.

A Dataset can be thought of as a specialized DataFrame, where the elements map to a specific JVM object type, instead of to a generic Row container. A DataFrame can be transformed into specific Dataset by calling `df.as[ElementType]`. Similarly you can transform a strongly-typed Dataset to a generic DataFrame by calling `ds.toDF()`.

COMPATIBILITY NOTE: Long term we plan to make DataFrame extend `Dataset[Row]`. However, making this change to the class hierarchy would break the function signatures for the existing functional operations (map, flatMap, etc). As such, this class should be considered a preview of the final API. Changes will be made to the interface after Spark 1.6.

Annotations	@Experimental()
Source	Dataset.scala
Since	1.6.0

▸ Linear Supertypes

Ordering Grouped Alphabetic By inheritance

Inherited Dataset Logging Serializable Serializable Queryable AnyRef Any

Hide All Show all Learn more about member selection

Visibility Public All

basic

Dataset

×

#ABCDEFGHIJKLMNOPQRSTUVWXY

– deprecated

display packages only

org.apache.spark.sql hide focus

- Dataset
- DatasetHolder
- KeyValueGroupedDataset
- RelationalGroupedDataset

org.apache.spark.sql

Dataset

Related Doc: package sql

`class Dataset[T] extends Serializable`

A Dataset is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a `DataFrame`, which is a Dataset of Row.

Operations available on Datasets are divided into transformations and actions. Transformations are the ones that produce new Datasets, and actions are the ones that trigger computation and return results. Example transformations include map, filter, select, and aggregate (`groupBy`). Example actions count, show, or writing data out to file systems.

Datasets are "lazy", i.e. computations are only triggered when an action is invoked. Internally, a Dataset represents a logical plan that describes the computation required to produce the data. When an action is invoked, Spark's query optimizer optimizes the logical plan and generates a physical plan for efficient execution in a parallel and distributed manner. To explore the logical plan as well as optimized physical plan, use the `explain` function.

To efficiently support domain-specific objects, an Encoder is required. The encoder maps the domain specific type `T` to Spark's internal type system. For example, given a class `Person` with two fields, `name` (string) and `age` (int), an encoder is used to tell Spark to generate code at runtime to serialize the `Person` object into a binary structure. This binary structure often has much lower memory footprint as well as are optimized for efficiency in data processing (e.g. in a columnar format). To understand the internal binary representation for data, use the `schema` function.

There are typically two ways to create a Dataset. The most common way is by pointing Spark to some files on storage systems, using the `read` function available on a `SparkSession`.

```
val people = spark.read.parquet("...").as[Person]  // Scala
Dataset<Person> people = spark.read().parquet("...").as(Encoders.bean(Person.class)); // Java
```

Datasets can also be created through transformations available on existing Datasets. For example, the following creates a new Dataset by applying a filter on the existing one:

```
val names = people.map(_.name)  // in Scala; names is a Dataset[String]
Dataset<String> names = people.map((Person p) -> p.name, Encoders.STRING));
```

Dataset operations can also be untyped, through various domain-specific-language (DSL) functions defined in: Dataset (this

DataFrame

A B C D E F G H I J K L M N O P Q R S T U V W X Y

– deprecated

display packages only

org.apache.spark.sql hide focus

- DataFrameNaFunctions
- DataFrameReader
- DataFrameStatFunctions
- DataFrameWriter

org.apache

spark

`package spark`

Core Spark functionality. org.apache.spark.SparkContext serves as the main entry point to Spark, while org.apache.spark.rdd.RDD is the data type representing a distributed collection, and provides most parallel operations.

In addition, org.apache.spark.rdd.PairRDDFunctions contains operations available only on RDDs of key-value pairs, such as `groupByKey` and `join`; org.apache.spark.rdd.DoubleRDDFunctions contains operations available only on RDDs of Doubles; and org.apache.spark.rdd.SequenceFileRDDFunctions contains operations available on RDDs that can be saved as SequenceFiles. These operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)] through implicit conversions.

Java programmers should reference the org.apache.spark.api.java package for Spark programming APIs in Java.

Classes and methods marked with **Experimental** are user-facing features which have not been officially adopted by the Spark project. These are subject to change or removal in minor releases.

Classes and methods marked with **Developer API** are intended for advanced users want to extend Spark through lower level interfaces. These are subject to changes or removal in minor releases.

Source package.scala

▸ Linear Supertypes

Ordering Alphabetic By Inheritance

Inherited spark AnyRef Any

Hide All Show All

Visibility Public All

Type Members

class **AnalysisException** extends Exception with Serializable

Thrown when a query fails to analyze, usually because the query itself is invalid.

---

class **Column** extends Logging

A column that will be computed based on the data in a `DataFrame`.

---

class **ColumnName** extends Column

A convenient class used for constructing schema.

---

type **DataFrame** = Dataset[Row]

---

final class **DataFrameNaFunctions** extends AnyRef

Functionality for working with missing data in `DataFrame`s.

---

class **DataFrameReader** extends Logging

Interface used to load a Dataset from external storage systems (e.g.

---

final class **DataFrameStatFunctions** extends AnyRef


We already know quite
about Datasets!


Dataset[Row]

Dataset[T]

Case Class

~~new~~

apply()

Like a regular Scala class

- But with a few differences

Modeling immutable data

An instance is called a Product

Initialized a little bit differently

```
case class Post(Id: Integer, UserId: String, Score: Integer)

val post = Post(1, "1", 25)

post.Id

post.UserId

post.UserId = "2"
```

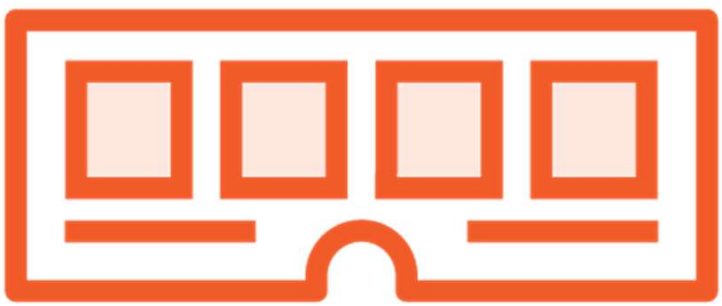
Case Class

Define case class

On object construction new is not required

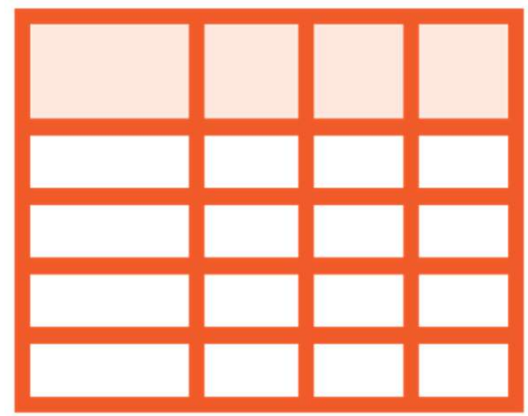
- By default `apply` method is used

Creating Datasets



Memory

.toDS()



DataFrame

.as[CaseClass]

```
val primitiveDS = Seq(10, 20, 30).toDS()

val differentDS = Seq(10, "20", 30).toDS()

primitiveDS.map(_ + 1).show()

val complexDS = Seq(("Xavier", 1), ("Irene", 2)).toDS()

complexDS.map(x => x._2 + 10).show()
```

From Data in Memory

Use createDataset(), with toDS()

Q: What did we use a lot in RDDs but not in DataFrames?

- Functional programming
- Higher-order functions with typed transformations

```
val postsRDD =
sc.textFile("/user/cloudera/stackexchange/simple_titles_txt")

val postsDSfromRDD = spark.createDataset(postsRDD)

postsDSfromRDD.show(5)
```

Using RDDs

Create Dataset from an RDD

Also using createDataset()

```
import org.apache.spark.sql.types._
val postsSchema =
StructType(Array(
StructField("Id", IntegerType),
StructField("PostTypeId", IntegerType),
StructField("AcceptedAnswerId", IntegerType),
StructField("CreationDate", TimestampType),
StructField("Score", IntegerType),
StructField("ViewCount", IntegerType),
StructField("OwnerUserId", IntegerType),
StructField("LastEditorUserId", IntegerType),
StructField("LastEditDate", TimestampType),
StructField("Title", StringType),
StructField("LastActivityDate", TimestampType),
StructField("Tags", StringType),
StructField("AnswerCount", IntegerType),
StructField("CommentCount", IntegerType),
StructField("FavoriteCount", IntegerType)))
```

```
case class Post(Id: Int, PostTypeId: Int, Score: Integer, ViewCount:
Integer, AnswerCount: Integer, OwnerUserId: Integer)

val posts_all = spark.read.schema(postsSchema)
.csv("/user/cloudera/stackexchange/posts_all_csv")

val postsDF = posts_all.select($"Id", $"PostTypeId", $"Score",
$"ViewCount", $"AnswerCount", $"OwnerUserId")
```

```
val postsDSfromDF = postsDF.as[Post]

postsDSfromDF.show(5)

postsDSfromDF
  .groupByKey(row => row.OwnerUserId).count().show()
```

From DataFrames with Case Classes

Create a Dataset from a DataFrame

- Using as[] and the case class

Perform strongly typed operations

Provide an API that allows performing transformations just like RDDs But with the performance and robustness of the Spark SQL execution engine

Dataset API

Typed Transformations

Untyped Transformations

```
postsDS

val postsLessDS = postsDS.filter('ViewCount < 533)

postsLessDS
```

Typed Transformations

Return a Dataset

- Type information is preserved

i.e. filter, sort, distinct...

```
val postsNotDS = postsDS.select('Id, 'ViewCount)

postsNotDS
```

Untyped Transformations

Return a DataFrame

- Type information not preserved

i.e. select, groupBy, join...

Typed Transformations

- Filter
- Distinct
- Limit

Untyped Transformations

- Select
- Join
- GroupBy

```
postsDS.describe("ViewCount").show()

val postsLessDS = postsDS.filter(p => (p.ViewCount == 533)).show()

postsDS.filter(p => p.OwnerUserId == 51).count()

val mini_postsDS = postsDS.map(d => (d.Id, d.Score))

val mini_postsDF = postsDS.select($"Id", $"Score")

postsDS.groupBy($"UserId")

postsDS.groupBy($"UserId"). //tab
```

Dataset Operations

- Transformations with domain specific objects

- What we learned earlier still applies
- Create new Datasets

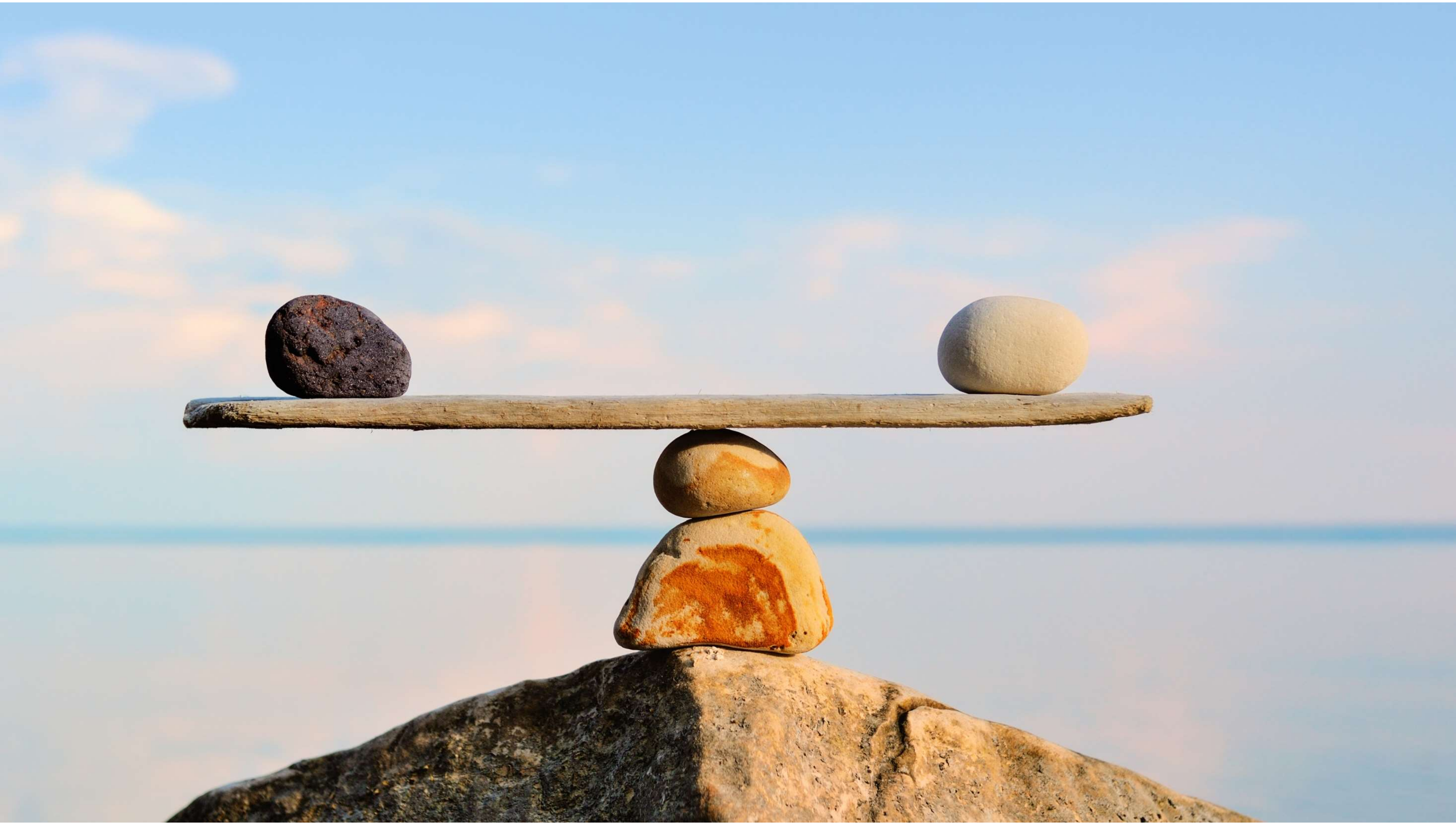
Explore the Dataset API

Performance

RDD	DataFrame	Dataset
Lower-level API	**Untyped Higher-level API**	**Typed Higher-level API**
Unstructured data	Structured data	Structured data
Fine tune	Semi-structured data	Semi-structured data
Manage low level details	"Think in SQL"	Type safety
Complex data types	Performance is key	Functional APIs

Performance

RDD	<	Dataset	<	DataFrame
Lower-level API		**Typed Higher-level API**		**Untyped Higher-level API**
Unstructured data		Structured data		Structured data
Fine tune		Semi-structured data		Semi-structured data
Manage low level details		Type safety		"Think in SQL"
Complex data types		Functional APIs		Performance is key



Takeaway

Typed API: Datasets

Motivation behind Datasets

- Performance

Syntax Errors vs. Analysis Errors

Takeaway

What's a Dataset?

- Strongly typed collection
- Domain specific objects

What can we store in a Dataset?

- Primitive types
- Complex types
- Product objects
- Row objects

Takeaway

Create Datasets

Typed transformations

Untyped transformations

Higher-order functions

Explore the API

RDDs vs. DataFrames vs. Datasets