High Performance Spark - June 3'rd, 2018

DataFramess, Datasets, and Spark SQL

https://github.com/rh01

- Spark SQL 与它的 DataFramess 、Datasets 接口是 Spark 性能的未来;
- 提供了更加高效的存储选项，更高级的优化器和对序列化数据的直接操作，这些特性对 Spark 性能有很大影响.
- Datasets were introduced in Spark 1.6,DataFramess in Spark1.3, and the SQL engine in Spark 1.0.

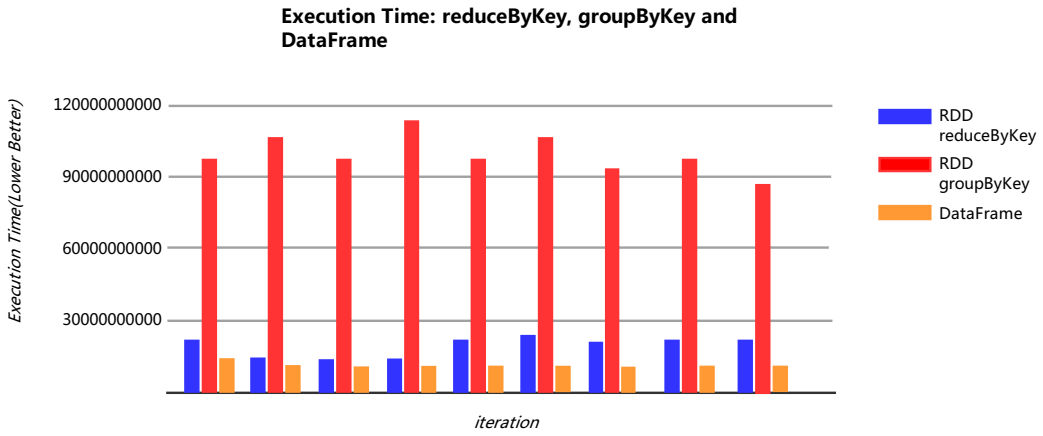

图 1: *Relative performance for RDD versus* ***DataFramess*** *based on SimplePerfTest computing aggregate average fuzziness of pandas.*

Spark's DataFramess have very different functionality compared to traditional DataFramess like Panda's and R's. While these all deal with structured data, it is important not to depend on your existing intuition surrounding DataFramess.

- Like RDDs, DataFramess and Datasets represent distributed collections, with additional schema information not found in RDDs.
- DataFramess are Datasets of a special Row object, which doesn't provide any compile-time type checking.

While Spark SQL, DataFramess, and Datasets provide many excellent enhancements, they still have some rough edges compared to traditional processing with "regular" RDDs. The Dataset API, being brand new at the time of this writing, is likely to experience some changes in future versions.

Getting Started with the SparkSession (or HiveContext or SQLContext)

就像 SparkContext 是所有 Spark 应用程序的入口点和 StreamingContext 是所有 Spark 流应用程序的入口点一样，SparkSession 也是 Spark SQL 的入口点。

```
1  import org.apache.spark.sql.{Dataset, DataFrame, SparkSession, Row}
2  import org.apache.spark.sql.catalyst.expressions.aggregate._
3  import org.apache.spark.sql.expressions._
4  import org.apache.spark.sql.functions._
```

Listing 1: Spark SQL imports

SparkSession 通常使用构建器模式以及 getOrCreate() 创建，如果会话已经存在，则将返回现有会话。构建器可以采用基于字符串的配置键 config(key,value)，并且存在许多常见参数的快捷方式。其中一个重要的快捷方式是 enableHiveSupport()，它可以让您访问 Hive UDF 并且不需要安装 Hive - 但需要一些额外的 JAR。

下面代码显示了如何创建支持 Hive 的 SparkSession。enableHiveSupport() 方法不仅使用这些 Hive JAR 配置 Spark SQL，而且还可以检查它们是否可以加载.

```
1  val session = SparkSession.builder()
2      .enableHiveSupport()
3      .getOrCreate()
4  // Import the implicits, unlike in core Spark the implicits are defined
5  // on the context.
6  import session.implicits._
```

Listing 2: Create a SparkSession

When using getOrCreate, if an existing session exists your configuration values may be ignored and you will simply get the existing SparkSession. Some options, like master, will also only apply if there is no existing SparkContext running; otherwise, the existing SparkContext will be used.

数据库/模式/架构（Schemas）的基础知识

架构信息及其启用的优化是 Spark SQL 与 Spark core 之间的核心差异之一。检查模式对于 DataFramess 特别有用，因为您没有使用 RDD 或 Datasets 进行模板化的类型。模式通常由 Spark SQL 自动处理，在加载数据时推断或基于父 DataFramess 应用的转换生成的。

DataFrames 以人类可读或编程格式开放模式信息。printSchema() 将会打印出 DataFrames 的架构/模式信息，并且最常用于在 Spark shell 中以找出您正在使用的内容。这对于数据格式尤其有用，例如 JSON，其中通过仅查看少量记录或读取标题可能无法立即看到模式。对于程序化用法，您可以通过简单地调用 schema 来获取模式，这通常在 ML pipeline 中使用.

```
{"name":"mission","pandas":[{"id":1,"zip":"94110","pt":"giant", "happy":true,
    "attributes":[0.4,0.5]}]}
```

Listing 3: JSON data that would result in an equivalent schema.

```
case class RawPanda(id: Long, zip: String, pt: String,
                            happy: Boolean, attributes: Array[Double])
case class PandaPlace(name: String, pandas: Array[RawPanda])
```

Listing 4: Equivalent case class.

```
def createAndPrintSchema() = {
    val damao = RawPanda(1, "M1B␣5K7", "giant", true, Array(0.1, 0.1))
    val pandaPlace = PandaPlace("toronto", Array(damao))
    val df = session.createDataFrame(Seq(pandaPlace))
    df.printSchema()
}
```

Listing 5: Create a Dataset with the case class.

```
root
 |-- name: string (nullable = true)
 |-- pandas: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- id: long (nullable = false)
 |    |    |-- zip: string (nullable = true)
 |    |    |-- pt: string (nullable = true)
 |    |    |-- happy: boolean (nullable = false)
 |    |    |-- attributes: array (nullable = true)
 |    |    |    |-- element: double (containsNull = false)
```

Listing 6: Sample schema information for nested structure (.printSchema()) .

Scala type	SQL type	Details
Byte	ByteType	1-byte signed integers (-128,127)
Short	ShortType	2-byte signed integers (-32768,32767)
Int	IntegerType	4-byte signed integers (-2147483648,2147483647)
Long	LongType	8-byte signed integers (- 9223372036854775808, 9223372036854775807)
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4-byte floating-point number
Double	DoubleType	8-byte floating-point number
Array[Byte]	BinaryType	Array of bytes
Boolean	BooleanType	true/false
java.sql.Date	DateType	Date without time information
java.sql.Timestamp	TimestampType	Date with time information (second precision)
String	StringType	Character string values (stored as UTF8)

表 1: Basic Spark SQL types

Scala type	SQL type	Details	Example
Array[T]	ArrayType(elementType, containsNull)	Array of single type of element, containsNull true if any null elements.	Array[Int] => ArrayType(IntegerType, true)
Map[K,V]	MapType(elementType, valueType, valueContainsNull)	Key/value map,valueContainsNull if any values are null.	Map[String, Int] => MapType(StringType, IntegerType, true)
case class	StructType(List[StructFields])	Named fields of possible heterogeneous types, similar to a case class or JavaBean.	case class Panda(name: String, age: Int) => StructType(List(StructField("name", StringType, true), StructField("age", IntegerType, true)))

表 2: Basic Spark SQL types

DataFrames API

Spark SQL 的 DataFrames API 允许我们使用 DataFramess 而无需注册临时表或生成 SQL 表达式。DataFrames API 具有 transforms 和 actions。DataFramess 上的转换本质上更具关系性.

Transformations

- DataFrames 上的转换在概念上与 RDD 转换类似，但具有更多的关系风格。与 RDD 一样，我们可以将转换大致分解为简单的单个 DataFramess ， 多个 DataFramess ， 键/值和分组/窗口转换。
- 简单的 DataFrames 转换和 SQL 表达式简单的 DataFrames 转换允许我们执行一次处理一行时可以执行的大多数标准操作。您仍然可以执行许多在 RDD 上定义的相同操作，除了使用 Spark SQL 表达式而不是任意功能。为了说明这一点，我们将首先检查 DataFrames 上可用的不同类型的过滤器操作。

- DataFrames 函数（如 filter）接受 Spark SQL 表达式而不是 lambdas 表达式。这些表达式允许优化器理解条件表示的内容，并且通过过滤器，它通常可用于跳过读取不必要的记录。

```
val pandaInfo = pandaPlace.explode(pandaPlace("pandas")){
    case Row(pandas: Seq[Row]) =>
    pandas.map{
        case (Row(
        id: Long,
        zip: String,
        pt: String,
        happy: Boolean,
        attrs: Seq[Double])) =>
        RawPanda(id, zip, pt, happy, attrs.toArray)
    }}
pandaInfo.select(
    (pandaInfo("attributes")(0) / pandaInfo("attributes")(1))
    .as("squishyness"))
```

Listing 7: Spark SQL select and explode operators

```
/**
* Encodes pandaType to Integer values instead of String values.
*
* @param pandaInfo the input DataFrame
* @return Returns a DataFrame of pandaId and integer value for pandaType.
*/
def encodePandaType(pandaInfo: DataFrame): DataFrame = {
    pandaInfo.select(pandaInfo("id"),
        (when(pandaInfo("pt") === "giant", 0).
        when(pandaInfo("pt") === "red", 1).
        otherwise(2)).as("encodedType")
    )
}
```

Listing 8: If/else in Spark SQL

Specialized DataFrames transformations for missing and noisy data

- Spark SQL also provides special tools for handling missing, null, and invalid data. By using `isNan` or isNull along with filters, you can create conditions for the rows you want to keep. For example, if you have a number of different columns, perhaps with different levels of precision (some of which may be null), you can use `coalesce(c1, c2, ..)` to return the first nonnull column. Similarly, for numeric data, `nanvl` returns the first non-NaN value (e.g., `nanvl(0/0, sqrt(-2), 3)` results in 3). To simplify working with missing data, the na function on `DataFrames` gives us access to some common routines for handling missing data in `DataFrames`NaFunctions.

Beyond row-by-row transformations

- Sometimes applying a row-by-row decision, as you can with filter, isn't enough. Spark SQL also

allows us to select the unique rows by calling dropDuplicates, but as with the similar operation on RDDs (distinct), this can require a shuffle, so is often much slower than filter. Unlike with RDDs, dropDuplicates can optionally drop rows based on only a subset of the columns, such as an ID field.

```
pandas.dropDuplicates(List("id"))
```

Aggregates and groupBy

- Spark SQL has many powerful aggregates, and thanks to its optimizer it can be easy to combine many aggregates into one single action/query. Like with Pandas' `DataFramess, groupBy` returns special objects on which we can ask for certain aggregations to be performed. In pre-2.0 versions of Spark, this was a generic `GroupedData`, but in versions 2.0 and beyond, `DataFramess groupBy` is the same as one `Datasets`.
- Aggregations on `Datasets` have extra functionality, returning a `GroupedDataset` (in pre-2.0 versions of Spark) or a `KeyValueGroupedDataset` when grouped with an arbitrary function, and a `RelationalGroupedDataset` when grouped with a relational/Dataset DSl expression.
- `min, max, avg`, and `sum` are all implemented as convenience functions directly on GroupedData, and more can be specified by providing the expressions to agg.

```
1  def maxPandaSizePerZip(pandas: DataFrame): DataFrame = {
2      pandas.groupBy(pandas("zip")).max("pandaSize")
3  }
```

Listing 9: Compute the max panda size by zip code.

```
1   // Compute the count, mean, stddev, min, max summary stats for all
2  // of the numeric fields of the provided panda infos. non-numeric
3  // fields (such as string (name) or array types) are skipped.
4  val df = pandas.describe()
5  // Collect the summary back locally
6  println(df.collect())
```

Listing 10: Compute some common summary stats, including count, mean, stddev, and more, on the entire `DataFrames`.

Windowing

```
1  val windowSpec = Window
2      .orderBy(pandas("age"))
3      .partitionBy(pandas("zip"))
4      .rowsBetween(start = -10, end = 10) // can use rangeBetween for range instead
```

Listing 11: Define a window on the +/-10 closest (by age) pandas in the same zip code.

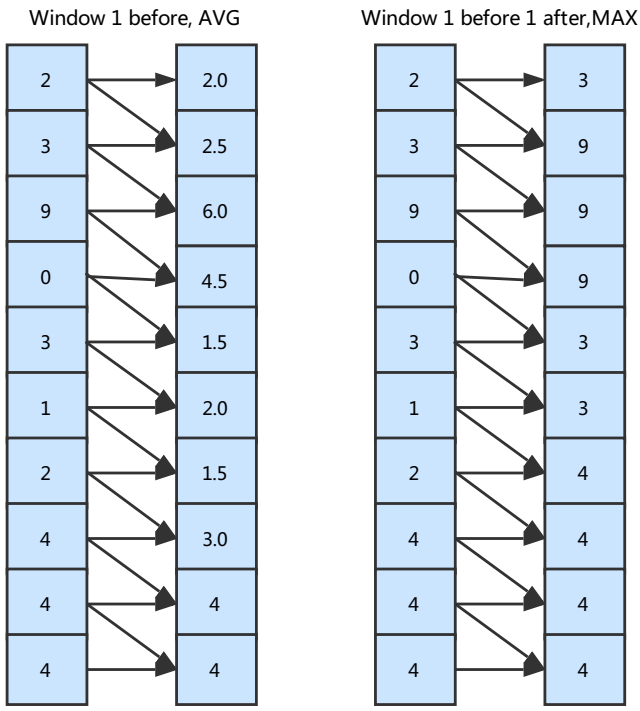

图 2: *Spark SQL windowing.*

Sorting

Sorting supports multiple columns in ascending or descending order, with ascending as the default.

```
1   pandas.orderBy(pandas("pandaSize").asc, pandas("age").desc)
```

Listing 12: Sort by panda age and size in opposite orders

Multi-DataFrames Transformations

Set-like operations

The DataFrames set-like operations allow us to perform many operations that are most commonly thought of as set operations. These operations behave a bit differently than traditional set operations since we don't have the restriction of unique elements.

Operation name	Cost
unionAll	Low
intersect	Expensive
except	Expensive
distinct	Expensive

表 3: Set operations

Data Representation in DataFramess and Datasets

DataFramess are more than RDDs of Row objects; DataFramess and Datasets have a specialized representation and columnar cache format. The specialized representation is not only more space efficient, but also can be much faster to encode than even Kryo serialization. To be clear, like RDDs, DataFramess and Datasets are generally lazily evaluated and build up a lineage of their dependencies (except in DataFramess this is called a logical plan and contains more information).

Data Loading and Saving Functions

Spark SQL has a different way of loading and saving data than core Spark. To be able to push down certaintypes of operations to the storage layer, Spark SQL has its own Data Source API. Data sources are able to specify and control which type of operations should be pushed down to the data source. As developers, you don't need to worry too much about the internal activity going on here, unless the data sources you are looking for are not supported.

DataFramesWriter and DataFramesReader

The DataFramesWriter and the DataFramesReader cover writing and reading from external data sources. The DataFramesWriter is accessed by calling write on a DataFrames or Dataset. The DataFramesReader can be accessed through read on a SQLContext.

Formats

When reading or writing you specify the format by calling format(formatName) on the DataFramesWriter/DataFramesR
Format-specific parameters, such as number of records to be sampled for JSON, are specified by either providing a map of options with options or setting option-by-option with option on the reader/writer.

JSON

Loading and writing JSON is supported directly in Spark SQL, and despite the lack of schema information in JSON, Spark SQL is able to infer a schema for us by sampling the records. Loading JSON data is more expensive than loading many data sources, since Spark needs to read some of the records to determine the schema information.

JDBC

The JDBC data source represents a natural Spark SQL data source, one that supports many of the operations. Since different database vendors have slightly different JDBC implementations, you need to add the JAR for your JDBC data sources. Since SQL field types vary as well, Spark uses JdbcDialects with built-in dialects for DB2, Derby, MsSQL, MySQL, Oracle, and Postgres. While Spark supports many different JDBC sources, it does not ship with the JARs required to talk to all of these databases. If you are submitting your Spark job with spark-submit you can download the required JARs to the host you are launching and include them by specifying –jars or supply the Maven coordinates to –packages. Since the Spark Shell is also launched this way, the same syntax works and you can use it to include the MySQL JDBC JAR.

JdbcDialects allow Spark to correctly map the JDBC types to the corresponding Spark SQL types. If there isn't a JdbcDialect for your database vendor, the default dialect will be used, which will likely work for many of the types. The dialect is automatically chosen based on the JDBC URL used.

As with the other built-in data sources, there exists a convenience wrapper for specifying the properties required to load JDBC data, illustrated in Listing 13. The convenience wrapper JDBC accepts the URL, table, and a java.util.Properties object for connection properties (such as authentication information).

The properties object is merged with the properties that are set on the reader/writer itself. While the properties object is required, an empty properties object can be provided and properties instead specified on the reader/writer.

```
session.read.jdbc("jdbc:dialect:serverName;user=user;password=pass",
    "table", new Properties)
session.read.format("jdbc")
    .option("url", "jdbc:dialect:serverName")
    .option("dbtable", "table").load()
```

Listing 13: Create a `DataFrames` from a JDBC data source.

```
df.write.jdbc("jdbc:dialect:serverName;user=user;password=pass",
    "table", new Properties)
df.write.format("jdbc")
    .option("url", "jdbc:dialect:serverName")
    .option("user", "user")
    .option("password", "pass")
    .option("dbtable", "table").save()
```

Listing 14: Write a `DataFrames` to a JDBC data source.

RDDs

Spark SQL `DataFrames`s can easily be converted to RDDs of Row objects, and can also be created from RDDs of Row objects as well as JavaBeans, Scala case classes, and tuples. For RDDs of strings in JSON format, you can use the methods discussed in "JSON". `Datasets` of type T can also easily be converted to RDDs of type T, which can provide a useful bridge for `DataFrames`s to RDDs of concrete case classes instead of Row objects. RDDs are a special-case data source, since when going to/from RDDs, the data remains inside of Spark without writing out to or reading from an external system.

Converting a `DataFrames` to an RDD is a transformation (not an action); however, converting an RDD to a `DataFrames` or Dataset may involve computing (or sampling some of) the input RDD.

Creating a `DataFrames` from an RDD is not free in the general case. The data must be converted into Spark SQL's internal format.

When you create a `DataFrames` from an RDD, Spark SQL needs to add schema information. If you are creating the `DataFrames` from an RDD of case classes or plain old Java objects (POJOs), Spark SQL is able to use reflection to automatically determine the schema, as shown in Listing 15. You can also manually specify the schema for your data using the structure discussed in "Basics of Schemas". This can be especially useful if some of your fields are not nullable. You must specify the schema yourself if Spark SQL is unable to determine the schema through reflection, such as an RDD of Row objects (perhaps from calling .rdd on a `DataFrames` to use a functional transformation, as shown in Listing 15).

```
def createFromCaseClassRDD(input: RDD[PandaPlace]) = {
    // Create DataFrame explicitly using session and schema inference
    val df1 = session.createDataFrame(input)
    // Create DataFrame using session implicits and schema inference
    val df2 = input.toDF()
    // Create a Row RDD from our RDD of case classes
    val rowRDD = input.map(pm => Row(pm.name,
        pm.pandas.map(pi => Row(pi.id, pi.zip, pi.happy, pi.attributes))))
    val pandasType = ArrayType(StructType(List(
      StructField("id", LongType, true),
      StructField("zip", StringType, true),
      StructField("happy", BooleanType, true),
      StructField("attributes", ArrayType(FloatType), true))))
    // Create DataFrame explicitly with specified schema
    val schema = StructType(List(StructField("name", StringType, true),
      StructField("pandas", pandasType)))
    val df3 = session.createDataFrame(rowRDD, schema)
  }
```

Listing 15: Creating `DataFramess` from RDDs.

Converting a `DataFrames` to an RDD is incredibly simple; however, you get an RDD of Row objects, as shown in Listing 16. Since a row can contain anything, you need to specify the type (or cast the result) as you fetch the values for each column in the row. With `Datasets` you can directly get back an RDD templated on the same type, which can make the conversion back to a useful RDD much simpler.

```
def toRDD(input: DataFrame): RDD[RawPanda] = {
    val rdd: RDD[Row] = input.rdd
    rdd.map(row => RawPanda(row.getAs[Long](0), row.getAs[String](1),
      row.getAs[String](2), row.getAs[Boolean](3), row.getAs[Array[Double]](4)))
}
```

Listing 16: Convert a `DataFrames`.

If you know that the schema of your `DataFrames` matches that of another, you can use the existing schema when constructing your new `DataFrames`. One common place where this occurs is when an input `DataFrames` has been converted to an RDD for functional filtering and then back.

Local collections

Much like with RDDs, you can also create `DataFramess` from local collections and bring them back as collections, as illustrated in Listing 17. The same memory requirements apply; namely, the entire contents

of the `DataFrames` will be in-memory in the driver program. As such, distributing local collections is normally limited to unit tests, or joining small datasets with larger distributed datasets.

```
def createFromLocal(input: Seq[PandaPlace]) = {
    session.createDataFrame(input)
}
```

Listing 17: Creating from a local collection.

Additional formats

As with core Spark, the data formats that ship directly with Spark only begin to scratch the surface of the types of systems with which you can interact. Some vendors publish their own implementations, and many are published on Spark Packages. As of this writing there are over twenty formats listed on the Data Source's page with the most popular being Avro, Redshift, CSV,6 and a unified wrapper around 6+ databases called deep-spark.

Save Modes

In core Spark, saving RDDs always requires that the target directory does not exist, which can make appending to existing tables challenging. With Spark SQL, you can specify the desired behavior when writing out to a path that may already have data. The default behavior is SaveMode.ErrorIfExists; matching the behavior of RDDs, Spark will throw an exception if the target already exists. The different save modes and their behaviors are listed in Table 4.

Save Mode	Behavior
ErrorIfExists	Throws an exception if the target already exists. If target doesn't exist write the data out.
Append	If target already exists, append the data to it. If the data doesn't exist write the data out.
Overwrite	If the target already exists, delete the target. Write the data out.
Ignore	If the target already exists, silently skip writing out. Otherwise write out the data.

表 4: Save modes

Partitions (Discovery and Writing)

Partition data is an important part of Spark SQL since it powers one of the key optimizations to allow reading only the required data, discussed more in "Logical and Physical Plans". If you know how your downstream consumers may access your data (e.g., reading data based on zip code), when you write your data it is beneficial to use that information to partition your output. When reading the data, it's useful to understand how partition discovery functions, so you can have a better understanding of whether your filter can be pushed down.

When reading partitioned data, you point Spark to the root path of your data, and it will automatically discover the different partitions. Not all data types can be used as partition keys; currently only strings and numeric data are the supported types.

If your data is all in a single `DataFrames`, the `DataFrames`Writer API makes it easy to specify the partition information while you are writing the data out. The partitionBy function takes a list of columns to partition

the output on, as shown in Listing 18. You can also manually save out separate DataFramess (say if you are writing from different jobs) with individual save calls.

```
1  def writeOutByZip(input: DataFrame): Unit = {
2      nput.write.partitionBy("zipcode").format("json").save("output/")
3  }
```

Listing 18: Save partitioned by zip code.

In addition to splitting the data by a partition key, it can be useful to make sure the resulting file sizes are reasonable, especially if the results will be used downstream by another Spark job.

Datasets

Datasets are an exciting extension of Spark SQL that provide additional compile-time type checking. Starting in Spark 2.0, DataFramess are now a specialized version of Datasets that operate on generic Row objects and therefore lack the normal compile-time type checking of Datasets. Datasets can be used when your data can be encoded for Spark SQL and you know the type information at compile time. The Dataset API is a strongly typed collection with a mixture of relational (DataFrames) and functional (RDD) transformations. Like DataFramess, Datasets are represented by a logical plan the Catalyst optimizer (see "Query Optimizer") can work with, and when cached the data is stored in Spark SQL's internal encoding format.

Interoperability with RDDs, DataFrames, and Local Collections

Datasets can be easily converted to/from DataFrames and RDDs, but in the initial version they do not directly extend either. Converting to/from RDDs involves encoding/decoding the data into a different form. Converting to/from DataFrames is almost "free" in that the underlying data does not need to be changed; only extra compile-time type information is added/removed.

To convert a DataFrame to a Dataset you can use the as[ElementType] function on the DataFrame to get a Dataset[ElementType] back as shown in Listing 18. The ElementType must be a case class, or similar such as tuple, consisting of types Spark SQL can represent (see "Basics of Schemas"). To create Datasets from local collections, createDataSet(...) on the SQLContext and the toDS() implicit function are provided on Seqs in the same manner as createDataFrame(...) and toDF(). For converting from RDD to Dataset you can first convert from RDD to DataFrame and then convert it to a Dataset.

```
1  def fromDF(df: DataFrame): Dataset[RawPanda] = {
2      df.as[RawPanda]
3  }
```

Listing 19: Create a Dataset from a DataFrame.

Converting from a Dataset back to an RDD or DataFrame can be done in similar ways as when converting DataFrames, and both are shown in Listing 19. The toDF simply copies the logical plan used in the Dataset into a DataFrame —so you don't need to do any schema inference or conversion as you do when converting from RDDs. Converting a Dataset of type T to an RDD of type T can be done by calling .rdd, which unlike calling toDF, does involve converting the data from the internal SQL format to the regular types.

```
/**
* Illustrate converting a Dataset to an RDD
*/
def toRDD(ds: Dataset[RawPanda]): RDD[RawPanda] = {
    ds.rdd
}
/**
* Illustrate converting a Dataset to a DataFrame
*/
def toDF(ds: Dataset[RawPanda]): DataFrame = {
    ds.toDF()
}
```

Listing 20: Convert Dataset to DataFrame and RDD.

Compile-Time Strong Typing

One of the reasons to use Datasets over traditional DataFrames is their compile-time strong typing. DataFrames have runtime schema information but lack compile-time information about the schema. This strong typing is especially useful when making libraries, because you can more clearly specify the requirements of your inputs and your return types.

Easier Functional (RDD "like") Transformations

One of the key advantages of the Dataset API is easier integration with custom Scala and Java code. Datasets expose filter, map, mapPartitions, and flatMap with similar function signatures as RDDs, with the notable requirement that your return ElementType also be understandable by Spark SQL (such as tuple or case class of types discussed in "Basics of Schemas"). Listing 20 illustrates this using a simple map function.

```
def funMap(ds: Dataset[RawPanda]): Dataset[Double] = {
    ds.map{rp => rp.attributes.filter(_ > 0).sum}
}
```

Listing 21: Functional query on Dataset.

Beyond functional transformations, such as map and filter, you can also intermix relational and grouped/aggregate operations.

Relational Transformations

Datasets introduce a typed version of select for relational-style transformations. When specifying an expression for this you need to include the type information, as shown in Listing 21 . You can add this information by calling as[ReturnType] on the expression/column.

```
def squishyPandas(ds: Dataset[RawPanda]): Dataset[(Long, Boolean)] = {
    ds.select($"id".as[Long], ($"attributes"(0) > 0.5).as[Boolean])
}
```

Listing 22: Simple relational select on Dataset.

Multi-Dataset Relational Transformations In addition to single Dataset transformations, there are also transformations for working with multiple Datasets. The standard set operations, namely intersect, union, and subtract, are all available with the same standard caveats as discussed in Table 3. Joining Datasets is also supported, but to make the type information easier to work with, the return structure is a bit different than traditional SQL joins.

Multi-Dataset Relational Transformations

Grouped Operations on Dataset

Similar to grouped operations on DataFrames (described in "Aggregates and groupBy"), groupBy on Datasets prior to Spark 2.0 returns a GroupedDataset or a KeyValueGroupedDataset when grouped with an arbitrary function, and a RelationalGroupedDataset when grouped with a relational/Dataset DSL expression. You can specify your aggregate functions on all of these, along with a functional mapGroups API. As with the expression in "Relational Transformations", you need to use typed expressions so the result can also be a Dataset.

Taking our previous example of computing the maximum panda size by zip in Listing 22, you would rewrite it to be as shown in Listing 22.

```
1  def maxPandaSizePerZip(ds: Dataset[RawPanda]): Dataset[(String, Double)] = {
2      ds.map(rp => MiniPandaInfo(rp.zip, rp.attributes(2)))
3      .groupByKey(mp => mp.zip).agg(max("size").as[Double])
4  }
```

Listing 23: Compute the max panda size per zip code typed.

Extending with User-Defined Functions and Aggregate Functions (UDFs, UDAFs)

User-defined functions and user-defined aggregate functions provide you with ways to extend the DataFrame and SQL APIs with your own custom code while keeping the Catalyst optimizer. The Dataset API (see "Datasets") is another performant option for much of what you can do with UDFs and UDAFs. This is quite useful for performance, since otherwise you would need to convert the data to an RDD (and potentially back again) to perform arbitrary functions, which is quite expensive. UDFs and UDAFs can also be accessed from inside of regular SQL expressions, making them accessible to analysts or others more comfortable with SQL.

Writing nonaggregate UDFs for Spark SQL is incredibly simple: you simply write a regular function and register it using sqlContext.udf().register. A simple string length UDF is shown in Listing 24. If you are registering a Java or Python UDF you also need to specify your return type.

```
1  def setupUDFs(sqlCtx: SQLContext) = {
2    sqlCtx.udf.register("strLen", (s: String) => s.length())
3  }
```

Listing 24: String length UDF.

Aggregate functions (or UDAFs) are somewhat trickier to write. Instead of writing a regular Scala function, you extend the UserDefinedAggregateFunction and implement a number of different functions, similar to the

functions one might write for aggregateByKey on an RDD, except working with different data structures. While they can be complex to write, UDAFs can be quite performant compared with options like mapGroups on Datasets or even simply written aggregateByKey on RDDs. You can then either use the UDAF directly on columns or add it to the function registry as you did for the nonaggregate UDF.

Listing 25 is a simple UDAF for computing the average, although you will likely want to use Spark's built in avg in real life.

```
def setupUDAFs(sqlCtx: SQLContext) = {
    class Avg extends UserDefinedAggregateFunction {
      // Input type
      def inputSchema: org.apache.spark.sql.types.StructType =
        StructType(StructField("value", DoubleType) :: Nil)
      def bufferSchema: StructType = StructType(
        StructField("count", LongType) ::
        StructField("sum", DoubleType) :: Nil
      )
      // Return type
      def dataType: DataType = DoubleType
      def deterministic: Boolean = true
      def initialize(buffer: MutableAggregationBuffer): Unit = {
        buffer(0) = 0L
        buffer(1) = 0.0
      }
      def update(buffer: MutableAggregationBuffer,input: Row): Unit = {
        buffer(0) = buffer.getAs[Long](0) + 1
        buffer(1) = buffer.getAs[Double](1) + input.getAs[Double](0)
      }
      def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
        buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
        buffer1(1) = buffer1.getAs[Double](1) + buffer2.getAs[Double](1)
      }
      def evaluate(buffer: Row): Any = {
        buffer.getDouble(1) / buffer.getLong(0)
      }
    }
    // Optionally register
    val avg = new Avg
    sqlCtx.udf.register("ourAvg", avg)
  }
```

Listing 25: UDAF for computing the average.

This is a little more complicated than our regular UDF, so let's take a look at what the different parts do. You start by specifying what the input type is, then you specify the schema of the buffer you will use for storing the in-progress work. These schemas are specified in the same way as DataFrame and Dataset schemas, discussed in "Basics of Schemas".

From there the rest of the functions are implementing the same functions you use when writing aggregateByKey on an RDD, but instead of taking arbitrary Scala objects you work with Row and MutableAggre-

gationBuffer. The final evaluate function takes the Row representing the aggregation data and returns the final result.

UDFs, UDAFs, and Datasets all provide ways to intermix arbitrary code with Spark SQL.