Under-the-hood: Creating Your Own Spark Data Sources

Speaker: Jayesh Thakrar @ Conversant

Why Build Your Own Data Source?

Don't need

- text, csv, ORC, Parquet, JSON files
- JDBC sources
- When available from project / vendor, e.g.
  - Cassandra, Kafka, MongoDB, etc
  - Teradata, Greenplum (2018), etc

Need to

- Use special features, e.g.
  - Kafka transactions
  - Exploit RDBMS specific partition features
- Because you can, and want to ☺

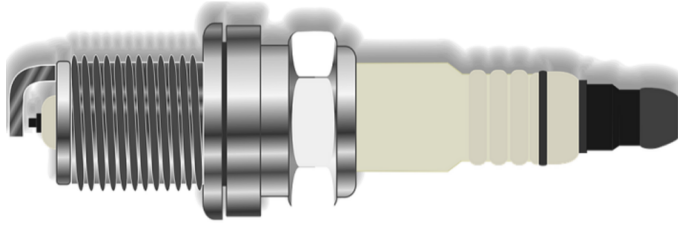
Conversant Use Cases (mid-2017)

- Greenplum as data source (read/write)
- Incompatibility between Spark, Kafka and Kafka connector versions

Agenda

1. Introduction to Spark data sources
2. Walk-through sample code
3. Practical considerations

Introduction To Spark Data Source



Using Data Sources

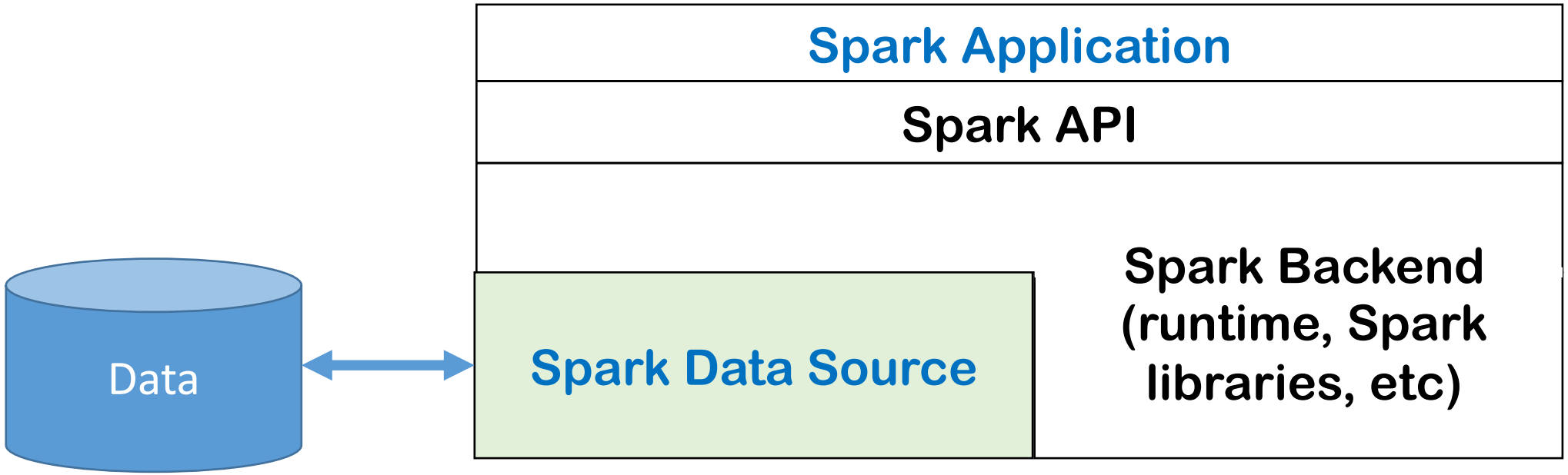
- **Built-in**

  spark.read.csv("path")
  spark.read.orc("path")
  spark.read.parquet("path")

- **Third-party/custom**

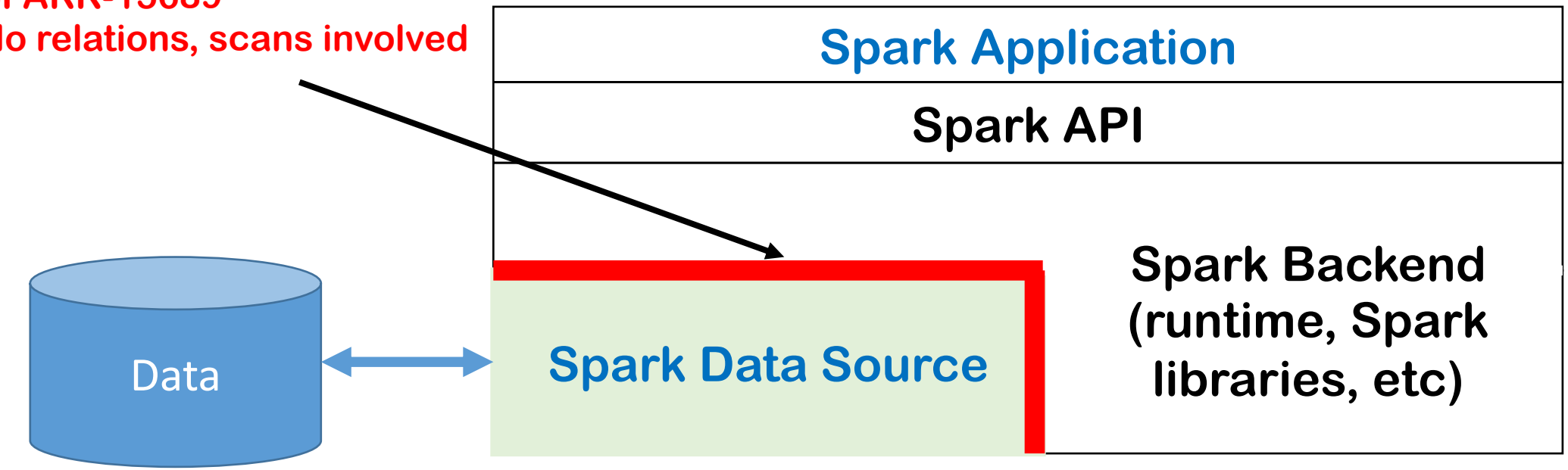
  spark.read.format("class-name").load() // custom data source
  spark.read.format("...").option("...", "...").load() // with options
  spark.read.format("...").schema("..").load() // with schema

Spark Data Source



Data Source V2 API

Shiny, New V2 API
since Spark 2.3 (Feb 2018)
SPARK-15689
No relations, scans involved



V2 API: Data Source Types

Data Source Type	Output
Batch	Dataset
Microbatch (successor to Dstreams)	Structured Stream = stream of bounded dataset
Continuous	Continuous Stream = continuous stream of Row(s)

V2 DATA SOURCE API

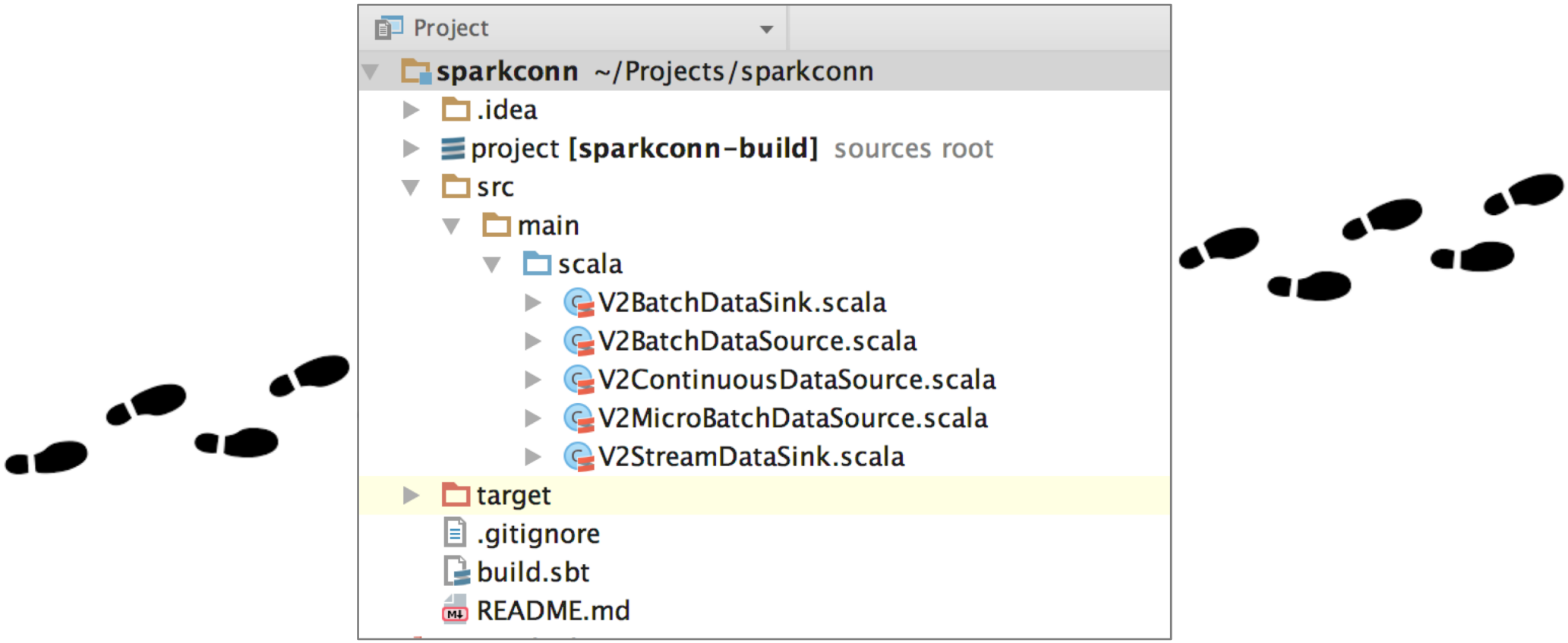
- **Well documented design, but still evolving**
  V2 API : Design Doc , Feature SPARK-15689
  Continuous Streaming Design Doc and Feature - SPARK-20928

- **Implemented as Java Interfaces (not classes)**

- **Similar interfaces across all data source types**

- **Microbatch and Continuous not hardened yet…**
  e.g. SPARK-23886, SPARK-23887, SPARK-22911

Code Walkthrough



Details

- **Project: https://github.com/JThakrar/sparkconn**

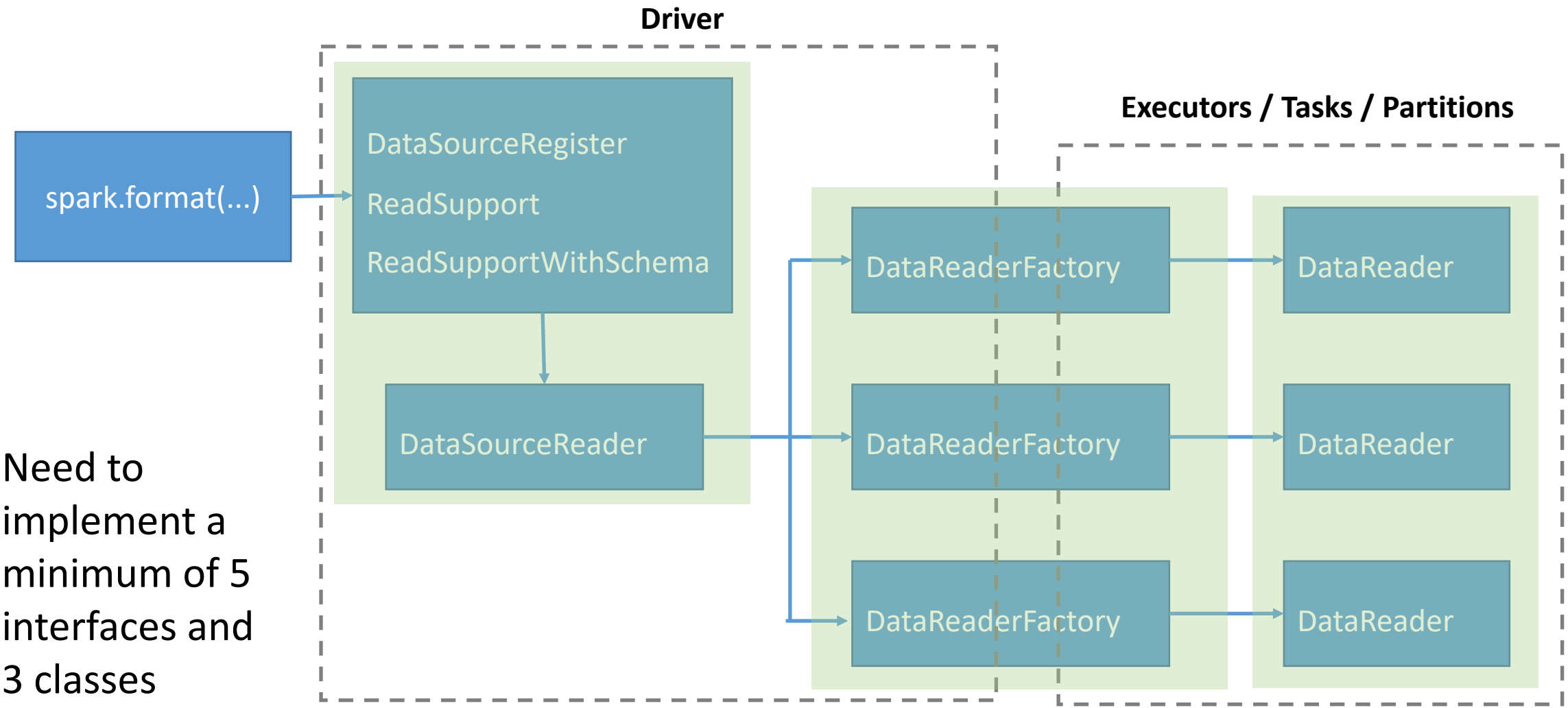
- **Requirements:**
  - **Scala 2.11**
  - **SBT**

Reading From Data Source

```
val data = spark.read.format("DataSource").option("key", "value").schema("..").load()
```

Step	Action
spark	= SparkSession
read	Returns a DataFrameReader for data source orchestration
format	Lazy lookup of data source class - by shortname or by full-qualified class name
option	Zero, one or more key-value pairs of options for data source
schema	Optional, user-provided schema
load	Loads the data as a dataframe. Remember dataframes are lazy-evaluated.

Reading: Interfaces to Implement



Need to implement a minimum of 5 interfaces and 3 classes

Interface Definitions And Dependency

libraryDependencies +=
"org.apache.spark" %%
"spark-sql" %
"2.3.1" %
"provided"

Example V2 API Based Datasource

- Very simple data source that generates a row with a fixed schema of a single string column. Column name = "string_value"
- Completely self-contained (i.e. no external connection)
- Number of partitions in dataset is user-configurable (default = 5)
- All partitions contains same number of rows (strings)
- Number of rows per partition is user-configurable (default = 5)

Read Interface: DataSourceRegister

Interface	Purpose
org.apache.spark.sql.sources.v2.**DataSourceRegister** org.apache.spark.sql.sources.v2.**ReadSupport** and / or org.apache.spark.sql.sources.v2.**ReadSupportWithSchema**	**DataSourceRegister** is the entry point for your data source. **ReadSupport** is then responsible to instantiate the object implementing DataSourceReader (discussed later). It accepts the options/parameters and optional schema from Spark application.

```
class V2BatchDataSource
  extends DataSourceRegister
    with ReadSupport { // use ReadSupportWithSchema to support user-provided Schema

  val DEFAULT_PARTITION_COUNT: Int = 5

  val DEFAULT_ROWS_PER_PARTITION: Int = 5

  override def shortName(): String = "v2batchdatasource" // to implement DataSourceRegister

  override def createReader(options: DataSourceOptions): DataSourceReader = { // to implement ReadSupport
    val optionsKV = options.asMap().asScala // converts options to lower-cased keyed map
    val partitions = optionsKV.getOrElse("partitions", DEFAULT_PARTITION_COUNT.toString).toInt
    val rows = optionsKV.getOrElse("rowsperpartition", DEFAULT_ROWS_PER_PARTITION.toString).toInt
    assert(partitions > 0, s"Partitions should be > 0 (specified value = $partitions)")
    assert(rows > 0, s"Rows should be > 0 (specified value = $rows)")
    println(s"\n\nCreating ${this.getClass.getName}: with $partitions partitions, each with $rows rowsPerPartition\n")
    new V2BatchDataSourceReader(partitions, rows)
  }
}
```

Read Interface: DataSourceReader

Interface	Purpose
org.apache.spark.sql.sources.v2.reader.**DataSourceReader**	This interface requires implementations for: • determining the schema of data • determining the number of partitions and creating that many reader factories below.

```
class V2BatchDataSourceReader(partitions: Int,
                              rows: Int)
  extends DataSourceReader {

  println(s"\n\nCreating ${this.getClass.getName}: $partitions partitions and $rows rowsPerPartition each\n")

  override def readSchema(): StructType = {
    StructType(StructField("string_value", StringType, false)::Nil)
  }

  override def createDataReaderFactories(): java.util.List[DataReaderFactory[Row]] = {
    println("Creating DataReaderFactories.........")
    (1 to partitions).map(partition =>
      new V2BatchDataReaderFactory(partition, rows, partitions).asInstanceOf[DataReaderFactory[Row]]).asJava
  }
}
```

Read Interface: DataReaderFactory

Interface	Purpose
org.apache.spark.sql.sources.v2.reader.**DataReaderFactory**	This is the "handle" passed by the driver to each executor. It instantiates the readers below and controls data fetch.

```
class V2BatchDataReaderFactory(partition: Int,
                               rows: Int,
                               totalPartitions: Int)
  extends DataReaderFactory[Row] {

  println(s"\n\nCreating ${this.getClass.getName}: $partition of $totalPartitions\n")

  override def createDataReader(): DataReader[Row] = {
    new V2BatchDataReader(partition, rows)
  }

}
```

Read Interface: DataReader

Interface	Purpose
org.apache.spark.sql.sources.v2.reader.**DataReader**	This does the actual work of fetching data from source (at task-level)

```
class V2BatchDataReader(partition: Int,
                        rows: Int)
  extends DataReader[Row] {

  private var rowsRemaining = rows

  println(s"\n\nCreating ${this.getClass.getName}: $partition\n")

  override def next(): Boolean = rowsRemaining > 0

  override def get(): Row = {
    var resultRow: Row = Row("")

    if (next) {
      rowsRemaining = rowsRemaining - 1
      resultRow = Row(s"Partition: $partition || Row ${rows - rowsRemaining} of $rows")
    } else {
      new IllegalArgumentException
    }
    resultRow
  }

  override def close(): Unit = {}
}
```

Summary

Because you are implementing interfaces, **YOU** can determine the "class" parameters and initialization

Interface	Purpose
org.apache.spark.sql.sources.v2.**DataSourceRegister** org.apache.spark.sql.sources.v2.**ReadSupport**	DataSourceRegister is the entry point for your connector. ReadSupport is then responsible to instantiate the object implementing DataSourceReader below. It accepts the options/parameters and optional schema from Spark application.
org.apache.spark.sql.sources.v2.reader.**DataSourceReader**	This interface requires implementations for: • determining the schema of data • determining the number of partitions and creating that many reader factories below. The **DataSourceRegister**, **DataSourceReader** and **DataReaderFactory** are instantiated at the driver. The driver then serializes **DataReaderFactory** and sends it to each of the executors.
org.apache.spark.sql.sources.v2.reader.**DataReaderFactory**	This is the "handle" passed by the driver to each executor. It instantiates the readers below and controls data fetch.
org.apache.spark.sql.sources.v2.reader.**DataReader**	This does the actual work of fetching data

Write Interfaces to Implement

Interface	Purpose
org.apache.spark.sql.sources.v2.**DataSourceRegister** org.apache.spark.sql.sources.v2.**WriteSupport**	DataSourceRegister is the entry point for your connector. WriteSupport is then responsible to instantiate the object implementing DataSourceWriter below. It accepts the options/parameters and the schema.
org.apache.spark.sql.sources.v2.writer.**DataSourceWriter**	This interface requires implementations for: • committing data write • aborting data write • creating writer factories The **DataSourceRegister**, **DataSourceWriter** and **DataWriterFactory** are instantiated at the driver. The driver then serializes **DataWriterFactory** and sends it to each of the executors.
org.apache.spark.sql.sources.v2.writer.**DataWriterFactory**	This is the "handle" passed by the driver to each executor. It instantiates the readers below and controls data fetch.
org.apache.spark.sql.sources.v2.writer.**DataWriter**	This does the actual work of writing and committing/aborting the data
org.apache.spark.sql.sources.v2.writer.**WriterCommitMessage**	This is a "commit" message that is passed from the **DataWriter** to **DataSourceWriter.**

Practical Considerations

Know Your Data Source

- Configuration
- Partitions
- Data schema
- Parallelism approach
- Batch and/or streaming
- Restart / recovery

V2 API IS STILL EVOLVING

- **SPARK-22386 -** Data Source V2 Improvements
- **SPARK-23507 -** Migrate existing data sources
- **SPARK-24073 -** DataReaderFactory Renamed in 2.4
- **SPARK-24252, SPARK-25006 -** DataSourceV2: Add catalog support
- **So Why use V2?**
  Future-ready and alternative to V2 needs significantly more time and effort!
  See https://www.youtube.com/watch?v=O9kpduk5D48

About…

Conversant

- Digital marketing unit of Epsilon under Alliance Data Systems (ADS)
- (Significant) player in internet advertising.

  We see about 80% of internet ad bids in the US
- Secret sauce = anonymous cross-device profiles driving personalized messaging

Me (Jayesh Thakrar)

- Sr. Software Engineer (jthakrar@conversantmedia.com)
-  https://www.linkedin.com/in/jayeshthakrar/

Questions?