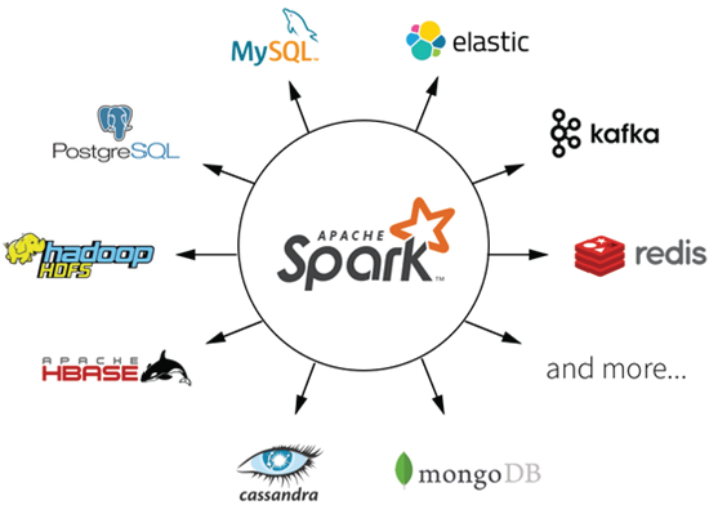
1. What is Data Source v2 API?

Data Source API is not a new feature for Apache Spark. The previous implementation is known as Data Source v1 which is introduced in Apache Spark 1.3 version. The main purpose of Data Source API in Apache Spark is to load data into Spark from third-party or custom sources through spark-packages. And since it is designed to let a developer intervene, own or necessary logics to read or write data to external data sources can easily be implemented.



Source: databricks

Some features that Spark can do with built-in sources are not available in Data Source v1. It means it is hard to make the external data source implementations as fast as the built-in ones.

On the other hand, Spark has flourished since version 1.3 in terms of custom memory management, abstraction of dataset and structured streaming capabilities whereas Data Source API remained the same during this evolution. Due to this problem, data source developers were experiencing huge problems in the development process and sometimes they were forced to fork Spark to make extensive changes. The problems and benefits mentioned above led to development of Data Source v2 aiming to reach the goals like a more Java-friendly API, better integration with Apache Spark optimizer, facilitating development of high performant, easy-to-maintain and easy-to-extend external data sources.

2. Limitations of Data Source v1 API

The main limitations that led to the development of new Data Source V2 are:
• Data Source v1 API was highly coupled with high level abstractions like SQLContext, RDD and DataFrame which were got deprecated or replaced by the evolution of the Apache Spark,
• Partitioning and Sorting is not propagated from the data sources, and thus, not used in the Spark optimizer,

• No transaction supports in write interface,
• Extensibility is not good and operator push-down capabilities are limited,
• Difficulty to add different data encoding and to implement writing,
• Lack of columnar read interface for high performance,
• Lack of Structured Streaming Support.

3. Limitations of Data Source v2 API

Major updates to the V2 API provide more control to the developers of data sources and improved integration capabilities with Spark Optimizer. Migrating to this API makes third-party sources perform better. Interfaces that need to be implemented for reading and writing operations are listed below. In this article, we will discuss only batch read and batch write operations. All implementations will be written in Scala Programming Language.

The below are the basic interfaces to read the data in Data Source V2 API:
- TableProvider
- Table
- ScanBuilder
- Scan
- Batch
- PartitionReaderFactory
- PartitionReader

Where some of them are common, the following interfaces are used to implement write operations in Data Source V2:
- TableProvider
- Table
- WriteBuilder
- Write
- BatchWrite
- DataWriterFactory
- Writer

3.1. Common Interfaces for Both Read & Write Operations

3.1.1. TableProvider

```
class CustomSourceProvider extends TableProvider {
  override def inferSchema(options: CaseInsensitiveStringMap): StructType = ???

  override def getTable(schema: StructType,
                        partitioning: Array[Transform], properties: util.Map[String, String]): Table = ???
}
```

TableProvider represents an external source that can be read from or written to.

Methods

• **inferSchema:** This method gets input parameter and tries to infer the schema.
• **inferPartitioning:** This method is used to infer the partitioning of the table.
• **getTable:** Returns a Table instance with specified schema and properties to read and write operations.
• **supportsExternalMetaData:** This method is used to get whether an external source accepts external metadata or not.

3.1.2. Table

```
class CustomSourceTable extends SupportsRead with SupportsWrite {
  override def name(): String = ???

  override def schema(): StructType = ???

  override def capabilities(): util.Set[TableCapability] = ???

  override def newScanBuilder(options: CaseInsensitiveStringMap): ScanBuilder = ???

  override def newWriteBuilder(info: LogicalWriteInfo): WriteBuilder = ???

}
```

Table interface represents a logical entity of external data source. It can be a JDBC table etc.

Methods

• **capabilities:** Specifies operations that external source will support (batch read). Those are the capabilities exposed by the table. This avails Spark to verify those endeavoring to run the operations.
• **name:** Name of external source.
• **schema:** Schema of external source.
• **newScanBuilder:** This method should be overridden if an external source supports reading operations.
• **newWriteBuilder:** This method should be overridden if an external source supports writing operations.

3.2. Interfaces for Read Operations

3.2.1. ScanBuilder

```
class CustomSourceScanBuilder extends ScanBuilder {
  override def build(): Scan = ???
}
```

This is an interface to be implemented for building the *Scan* object. Pushdown filters and push down required column operations should be also dealt within this interface.

Methods

• **build:** This method is used to gather a Scan instance.

3.2.2. Scan

```
class CustomSourceScan extends Scan {
  override def readSchema(): StructType = ???

  override def toBatch: Batch = ???
}
```

A logical representation of an external data source scan.

Methods:

• **readSchema:** The actual schema of an external data source. This method is similar to the one in the Table interface, except the schema may change as a result of column reduction or another optimization. On the other hand, we may require inference of the schema. Unlike the Table interface, which returns the initial schema of the data source, this method returns the actual data structure.

• **toBatch:** If an external source supports batch read, this method should be overridden to indicate that this scan configuration should be used for batch reading.

3.2.3. Batch

```
class CustomSourceBatch extends Batch {
  override def planInputPartitions(): Array[InputPartition] = ???

  override def createReaderFactory(): PartitionReaderFactory = ???
}
```

This interface is a physical representation of an external data source for batch queries. The physical details, such as how many partitions the scanned data has and how to retrieve the partitions' records, are provided through this interface.

Methods:

• **planInputPartitions:** Array of input partition. Input partition count can be static or it can be given by the user. Input partitions are serializable and will be sent to Executors at runtime.

• **createReaderFactory:** A factory to create readers as many as the partition number.

3.2.4. PartitionReaderFactory

```
class CustomSourcePartitionReaderFactory extends PartitionReaderFactory {
  override def createReader(partition: InputPartition): PartitionReader[InternalRow] = ???
}
```

PartitionReaderFactory interface creates a partition reader. *PartitionReaderFactory* creates partition readers as many as the number of partitions and sends them to Spark executors.

Methods:

• **createReader:** Returns a row-based partition reader to read data from the given InputPartition.

3.2.5. PartitionReader

```
class CustomSourcePartitionReader extends PartitionReader[InternalRow] {
  override def next(): Boolean = ???

  override def get(): InternalRow = ???

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

PartitionReader interface is responsible for outputting data for an RDD partition. This is the interface where we actually read the data.

Methods:

• **next:** Proceeds to next record, if there is no more record it returns false.
• **get:** Returns the current record.
• **close:** Internally closeable objects should be dealt within this method.

3.3. Interfaces for Write Operations

3.3.1.WriteBuilder

```
class CustomSourceWriteBuilder extends WriteBuilder {

  override def build(): Write = ???

}
```

An interface for building the Write.

Methods:

• **build:** This method is used to get a Write instance.

3.3.2. Write

```
class CustomSourceWrite extends Write {

  override def toBatch: BatchWrite = ???

}
```

A logical representation of an external data source write.

Methods:

• **toBatch:** Returns a BatchWrite to write data to the target.

3.3.3. BatchWrite

An interface that defines how to write the data to data source for batch processing. Runs on Spark Driver. For each partition, it creates a data writer.

```
class CustomSourceBatchWrite extends BatchWrite {
  override def createBatchWriterFactory(info: PhysicalWriteInfo): DataWriterFactory = ???

  override def commit(messages: Array[WriterCommitMessage]): Unit = ???

  override def abort(messages: Array[WriterCommitMessage]): Unit = ???
}
```

Methods:

• **createBatchWriterFactory:** Creates a writer factory that will be serialized and sent to executors.
• **commit:** If all data writers completed successfully this method is called to finish this writing job.
• **abort:** If some data writers fail or keep failing when retried, this method is called to abort the writing job.

3.3.4. DataWriterFactory

```
class CustomSourceDataWriterFactory extends DataWriterFactory {
  override def createWriter(partitionId: Int, taskId: Long): DataWriter[InternalRow] = ???
}
```

A factory that is responsible for creating and initializing the actual data writer at executor side. The writer factory will be serialized and sent to executors then the data writer will be created on executors and do the actual writing. So, this interface must be serialized but DataWriter doesn't need to be.

Methods:

• **createWriter:** Creates a data writer to do the actual writing work.

3.3.5. DataWriter

```
class CustomSourceDataWriter extends DataWriter[InternalRow] {

  override def write(record: InternalRow): Unit = ???

  override def commit(): WriterCommitMessage = ???

  override def abort(): Unit = ???

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

A *DataWriter* interface is responsible for writing data to the target. Each partition is related to one *DataWriter*. Each *DataWriter* runs on Spark Executors.

Methods:

• **abort:** Aborts this writer if it is failed.
• **commit:** If all records of this writer are written successfully, returns a commit message to driver side to BatchWrite.
• **write:** Writes one record at a time.
• **close:** Internally closeable objects should be dealt within this method.

There are additional methods and interfaces such as *SupportsPushDownFilters* and *preferredLocations*. These methods and interfaces can be found in API docs.

4. Conclusion

Businesses rely extensively on a wide range of data sources for their analytical goods. These data processing workflows involve tasks like cleaning, converting, and combining unstructured external data with internal data sources. Spark is proving to be quite useful in this manner. Some businesses have also developed straightforward user interfaces that make batch data processing operations accessible to non-programmers.

The ability of Spark to accommodate numerous data sources and programming languages was well known. Spark ETLs provide clean data for all types of data, including relational and semi-structured data (like JSON). Spark data pipelines are built to process massive volumes of data.

When using dispersed data from many sources, such as relational databases or data warehouses, Spark applications frequently need to directly access external data sources. To help with this, Spark offers Data Source APIs, a pluggable method of using Spark SQL to retrieve structured data. The Spark Optimizer is intimately connected with Data Source APIs. They offer improvements like column trimming and filter push down to the external data source. Depending on the data source, these optimizations can greatly speed up Spark query execution, but they only offer a portion of the capability that can be sent down to and used at the data source.

Data Source v2 API is one of the core features of Spark. Data Source V2 is developed due to the limitations of Data Source V1. With new features, it offers many new capabilities to developers. And Data Source API will continue to improve in the future as it is one of the core features of Spark.

REFERENCES

1. https://docs.google.com/document/d/1n_vUVbF4KD3gxTmkNE-on5qdQ-Z8qU5Frf6WMQZ6jJVM/edit?usp=sharing

2. https://www.databricks.com/session/apache-spark-data-source-v2

3. https://spark.apache.org/docs/3.2.0/api/java/org/apache/spark/sql/connector/read/

4. https://spark.apache.org/docs/3.2.0/api/java/org/apache/spark/sql/connector/write/

5. https://blog.madhukaraphatak.com/categories/datasource-v2-spark-three/

6. https://docs.google.com/document/d/1DDXCTCrup4bKWByTalkXW-gavcPdvur8a4eEu8x1BzPM

Contact Us

interprobe

InterProbe_

InterProbe Bilgi Teknolojileri

https://interprobe.com.tr