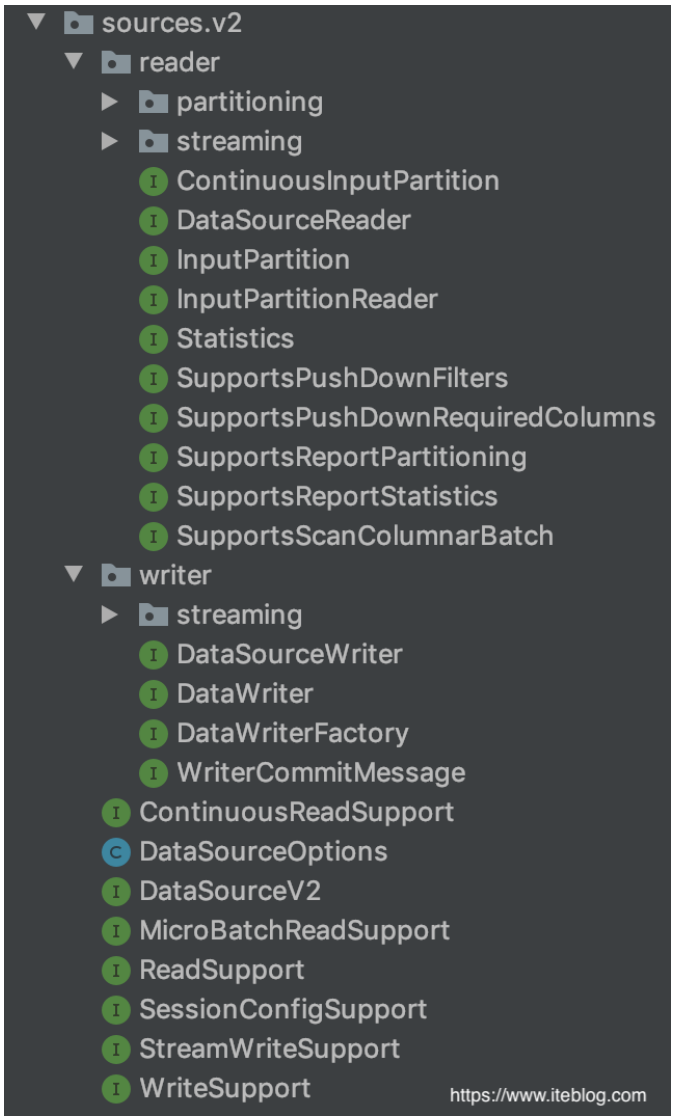
Apache Spark DataSource V2 介绍及入门编程指南（下）

我们在 Apache Spark DataSource V2 介绍及入门编程指南（上） 文章中介绍了 Apache Spark DataSource V1 的不足，所以才有了 Data Source API V2 的诞生。

Data Source API V2

为了解决 Data Source V1 的一些问题，从 Apache Spark 2.3.0 版本开始，社区引入了 Data Source API V2，在保留原有的功能之外，还解决了 Data Source API V1 存在的一些问题，比如不再依赖上层 API，扩展能力增强。Data Source API V2 对应的 ISSUE 可以参见 SPARK-15689。

本文以最新的 Apache Spark 2.4.3 版本进行介绍，这个版本的 Data Source API V2 主要抽象出以下几个接口：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

这些抽象出来的类全部存放在 sql 模块中 core 的 org.apache.spark.sql.sources.v2 包里面，咋一看好像类的数目比之前要多了，但是功能、扩展性却比之前要好很多的。从上面的包目录组织结构可以看出，Data Source API V2 支持读写、流数据写、微批处理读（比如 KafkaSource 就用到这个了）以及 ContinuousRead（continuous stream processing）等多种方式读。

在 reader 包里面有 SupportsPushDownFilters、SupportsPushDownRequiredColumns、SupportsReportPartitioning、SupportsReportStatistics 以及 SupportsScanColumnarBatch，分别对应的含义是算子下推、列裁剪、数据分区、统计信息以及批量列扫描等。

为了加深大家对 Data Source API V2 的印象，本文将介绍使用 Data Source API V2 编写一个读取 MySQL 数据的程序。

实现 ReadSupport 接口

为了使用 Data Source API V2，我们肯定是需要使用到 Data Source API V2 包里面相关的类库，对于读取程序，我们只需要实现 ReadSupport 相关接口就行，如下：

```
package com.iteblog.mysql

import org.apache.spark.sql.sources.v2.reader.DataSourceReader
import org.apache.spark.sql.sources.v2.{DataSourceOptions, DataSourceV2, ReadSupport}
import scala.collection.JavaConverters._

class DefaultSource extends DataSourceV2 with ReadSupport{
  override def createReader(options: DataSourceOptions): DataSourceReader = {
    MySQLSourceReader(options.asMap().asScala.toMap)
  }
}
```

我们定义了一个 DefaultSource 的类，实现了 ReadSupport 接口，并使用 DataSourceV2 标记这是一个 Data Source API V2 的程序。注意，Data Source API V2 的程序必须实现 ReadSupport 或 WriteSupport 接口中的一个或两个，分别代表读和写的逻辑。这里为了简便起见，我们只实现了 ReadSupport 接口。

实现读 MySQL 相关操作

前面我们实现了 ReadSupport 接口，并重写了 createReader 方法。这里我们需要实现 DataSourceReader 接口相关的操作，如下：

```
package com.iteblog.mysql

import java.util

import org.apache.spark.sql.catalyst.InternalRow
import org.apache.spark.sql.execution.datasources.jdbc.{JDBCOptions, JDBCRDD}
import org.apache.spark.sql.sources.{EqualTo, Filter, GreaterThan, IsNotNull}
import org.apache.spark.sql.sources.v2.reader.{DataSourceReader, InputPartition, SupportsPushDownFilters, SupportsPushDownRequiredColumns}
import org.apache.spark.sql.types.StructType

import scala.collection.mutable.ArrayBuffer
import scala.collection.JavaConverters._

case class MySQLSourceReader(options: Map[String, String]) extends DataSourceReader {

  var requiredSchema: StructType = {
    val jdbcOptions = new JDBCOptions(options)
    JDBCRDD.resolveTable(jdbcOptions)
  }

  override def readSchema(): StructType = {
    requiredSchema
  }

  override def planInputPartitions(): util.List[InputPartition[InternalRow]] = {
    List[InputPartition[InternalRow]](MySQLInputPartition(requiredSchema, options)).asJava
  }
}
```

DataSourceReader 接口我们需要分别实现 readSchema 和 planInputPartitions 方法，分别代表我们程序需要读取的列相关信息，以及每个分区拆分及读取逻辑等。细心的同学肯定可以想到，读取操作不是可以弄一些算子下推，列裁剪相关的优化吗？没错，由于 DataSource V2 的优化，我们可以在这里加上 SupportsPushDownFilters、SupportsPushDownRequiredColumns、SupportsReportPartitioning 等相关的优化，完整的程序如下：

```
package com.iteblog.mysql

import java.util

import org.apache.spark.sql.catalyst.InternalRow
import org.apache.spark.sql.execution.datasources.jdbc.{JDBCOptions, JDBCRDD}
import org.apache.spark.sql.sources.{EqualTo, Filter, GreaterThan, IsNotNull}
```

```
import org.apache.spark.sql.sources.v2.reader._
import org.apache.spark.sql.types.StructType

import scala.collection.mutable.ArrayBuffer
import scala.collection.JavaConverters._

case class MySQLSourceReader(options: Map[String, String])
  extends DataSourceReader with SupportsPushDownRequiredColumns with SupportsPushDo
wnFilters {

  val supportedFilters: ArrayBuffer[Filter] = ArrayBuffer[Filter]()

  var requiredSchema: StructType = {
    val jdbcOptions = new JDBCOptions(options)
    JDBCRDD.resolveTable(jdbcOptions)
  }

  override def readSchema(): StructType = {
    requiredSchema
  }

  override def planInputPartitions(): util.List[InputPartition[InternalRow]] = {
    List[InputPartition[InternalRow]](MySQLInputPartition(requiredSchema, supportedFilters.to
Array, options)).asJava
  }

  override def pushFilters(filters: Array[Filter]): Array[Filter] = {
    if (filters.isEmpty) {
      return filters
    }

    val unsupportedFilters = ArrayBuffer[Filter]()

    filters.foreach {
      case f: EqualTo => supportedFilters += f
      case f: GreaterThan => supportedFilters += f
      case f: IsNotNull => supportedFilters += f
      case f@_ => unsupportedFilters += f
    }

    unsupportedFilters.toArray
  }

  override def pushedFilters(): Array[Filter] = supportedFilters.toArray

  override def pruneColumns(requiredSchema: StructType): Unit = {
```

```
    this.requiredSchema = requiredSchema
  }
}
```

上面程序我们加上了列裁剪和算子下推。其中 pushedFilters 和 pushFilters 方法分别代码可以推下去的过滤以及不可以推下去的过滤。具体那些可以推下去，哪些不可以推下去是根据我们自己实现的。比如本例中只支持下推等于(EqualTo)、大于（GreaterThan）以及不为空（IsNotNull）的过滤条件，其他不支持。pruneColumns 这个方法就是列裁剪，就是我们 Spark SQL 中需要使用到的列，比如 select id, name from iteblog where age > 10 and state != 1 这条 SQL 列裁剪需要的列为 id、name 以及 state，其他的列不需要读取到 Spark 层面上来。

大家再仔细思路可以看出，DataSource V2 把每种优化都写到单独的一个接口里面，这样我们需要哪个优化就可以加哪个，这样就可以排列组合出很多种用法，这明显比 DataSource V1 版本的 PrunedFilteredScan 要灵活很多。假如我们需要将 limit 下推，我们只需要定义一个类似于 SupportsPushDownLimit 接口即可，非常的灵活。

最后一个需要我们实现的就是分片读取，在 DataSource V1 里面缺乏分区的支持，而 DataSource V2 支持完整的分区处理，也就是上面的 planInputPartitions 方法。在那里我们可以定义使用几个分区读取数据源的数据。比如如果是 TextInputFormat，我们可以读取到对应文件的 splits 个数，然后每个 split 构成这里的一个分区，使用一个 Task 读取。为了简便起见，我这里使用了只使用了一个分区，也就是 List[InputPartition[InternalRow]](MySQLInputPartition(requiredSchema, supportedFilters.toArray, options)).asJava。

分区读取实现

到这里，我们需要定义每个分区具体是如何读取的，这里就是真实的数据读取实现逻辑，比如本文例子的实现如下：

```
package com.iteblog.mysql

import java.sql.{DriverManager, ResultSet}

import org.apache.spark.sql.catalyst.InternalRow
import org.apache.spark.sql.catalyst.util.DateTimeUtils
import org.apache.spark.sql.jdbc.JdbcDialects
import org.apache.spark.sql.sources.{EqualTo, Filter, GreaterThan, IsNotNull}
import org.apache.spark.sql.sources.v2.reader.{InputPartition, InputPartitionReader}
import org.apache.spark.sql.types._
import org.apache.spark.unsafe.types.UTF8String

case class MySQLInputPartition(requiredSchema: StructType, pushed: Array[Filter], options: M
```

```
ap[String, String])
  extends InputPartition[InternalRow] {

  override def createPartitionReader(): InputPartitionReader[InternalRow] =
    MySQLInputPartitionReader(requiredSchema, pushed, options)
}

case class MySQLInputPartitionReader(requiredSchema: StructType, pushed: Array[Filter], opti
ons: Map[String, String])
  extends InputPartitionReader[InternalRow] {

  val tableName: String = options("dbtable")
  val driver: String = options("driver")
  val url: String = options("url")

  def initSQL: String = {
    val selected = if (requiredSchema.isEmpty) "1" else requiredSchema.fieldNames.mkString(","
)

    if (pushed.nonEmpty) {
      val dialect = JdbcDialects.get(url)
      val filter = pushed.map {
        case EqualTo(attr, value) => s"${dialect.quoteIdentifier(attr)} = ${dialect.compileValue(valu
e)}"
        case GreaterThan(attr, value) => s"${dialect.quoteIdentifier(attr)} > ${dialect.compileValue
(value)}"
        case IsNotNull(attr) => s"${dialect.quoteIdentifier(attr)} IS NOT NULL"
      }.mkString(" AND ")

      s"SELECT $selected FROM $tableName WHERE $filter"
    } else {
      s"SELECT $selected FROM $tableName"
    }
  }

  val rs: ResultSet = {
    Class.forName(driver)
    val conn = DriverManager.getConnection(url)
    println(initSQL)
    val stmt = conn.prepareStatement(initSQL, ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONC
UR_READ_ONLY)
    stmt.setFetchSize(1000)
    stmt.executeQuery()
  }

  override def next(): Boolean = rs.next()
```

```
  override def get(): InternalRow = {
    InternalRow(requiredSchema.fields.zipWithIndex.map { element =>
      element._1.dataType match {
        case IntegerType => rs.getInt(element._2 + 1)
        case LongType => rs.getLong(element._2 + 1)
        case StringType => UTF8String.fromString(rs.getString(element._2 + 1))
        case e: DecimalType => val d = rs.getBigDecimal(element._2 + 1)
          Decimal(d, d.precision, d.scale)
        case TimestampType => val t = rs.getTimestamp(element._2 + 1)
          DateTimeUtils.fromJavaTimestamp(t)
      }
    }: _*)
  }

  override def close(): Unit = rs.close()
}
```

具体分区读取是需要实现 InputPartitionReader 接口的，大家可以看到，这里面就是真正的 MySQL 查询 SQL 的拼接，以及我们平时参见的 MySQL
数据查询方法。仔细的同学可以看出拼接的 SQL 中 where
条件里面的就是我们的算子下推逻辑；而 select 部分就是我们的列裁剪部分。

使用 DataSource V2

到这里，我们已经使用 DataSource V2 API 定义了一个读取 MySQL 的类库，我们可以像正常 Spark 类库一样使用这个类库，如下：

```
val df = spark.read
    .format("com.iteblog.mysql")
    .option("url", url)
    .option("dbtable", "search_info")
    .option("driver", "com.mysql.jdbc.Driver")
    .load().filter("id > 10")

df.explain(true)
```

这条 SQL 没有使用到
select，所以会使用到表中所有的列，并且以为我们已经支持大于等算子下推，所以 id > 10
这个应该是会下推到 MySQL 端执行的，具体的执行计划如下：

```
== Parsed Logical Plan ==
'Filter ('id > 10)
+- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=search_
info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Analyzed Logical Plan ==
ID: decimal(20,0), ip: string, count: int, times: timestamp, total: decimal(20,0)
Filter (cast(id#0 as decimal(20,0)) > cast(cast(10 as decimal(2,0)) as decimal(20,0)))
+- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=search_
info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Optimized Logical Plan ==
Filter (isnotnull(id#0) && (id#0 > 10))
+- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=search_
info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Physical Plan ==
*(1) Project [ID#0, ip#1, count#2, times#3, total#4]
+- *(1) ScanV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Filters: [isnotnull(id#0), (i
d#0 > 10)], Options: [dbtable=search_info,driver=com.mysql.jdbc.Driver,url=*********(redact
ed),paths=[]])
```

从上面可以清晰看到 id > 10 已经下推了，见 Filters: [isnotnull(id#0), (id#0 > 10)]。对应拼接出来的 SQL 为

```
SELECT ID,ip,count,times,total FROM search_info WHERE `id` IS NOT NULL AND `id` > 10
```

在看下下面的测试：

```
val df = spark.read
    .format("com.iteblog.mysql")
    .option("url", url)
    .option("dbtable", "search_info")
    .option("driver", "com.mysql.jdbc.Driver")
    .load().filter("id > 10 and count >= 10").select("id", "ip")
df.explain(true)
```

对应的执行计划如下：

```
== Parsed Logical Plan ==
'Project [unresolvedalias('id, None), unresolvedalias('ip, None)]
+- Filter ((cast(id#0 as decimal(20,0)) > cast(cast(10 as decimal(2,0)) as decimal(20,0))) && (coun
t#2 >= 10))
   +- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=searc
h_info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Analyzed Logical Plan ==
id: decimal(20,0), ip: string
Project [id#0, ip#1]
+- Filter ((cast(id#0 as decimal(20,0)) > cast(cast(10 as decimal(2,0)) as decimal(20,0))) && (coun
t#2 >= 10))
   +- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=searc
h_info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Optimized Logical Plan ==
Project [id#0, ip#1]
+- Filter (((isnotnull(id#0) && isnotnull(count#2)) && (id#0 > 10)) && (count#2 >= 10))
   +- RelationV2 DefaultSource[ID#0, ip#1, count#2, times#3, total#4] (Options: [dbtable=searc
h_info,driver=com.mysql.jdbc.Driver,url=*********(redacted),paths=[]])

== Physical Plan ==
*(1) Project [id#0, ip#1]
+- *(1) Filter (count#2 >= 10)
   +- *(1) ScanV2 DefaultSource[ID#0, ip#1, count#2] (Filters: [isnotnull(count#2), isnotnull(id#
0), (id#0 > 10)], Options: [dbtable=search_info,driver=com.mysql.jdbc.Driver,url=*********(re
dacted),paths=[]])
```

从上面的 Physical Plan 可以看出，count#2 >= 10 这个并没有推到数据源执行，以为我们这个例子里面没有实现大于等于算子的下推。本例我们使用了 select，并且指定了 id、ip 列，再加上没有推到 MySQL 端的列，所以这次执行只需要获取 id、ip 以及 count 三列即可，最后拼接后的 SQL 如下：

```
SELECT ID,ip,count FROM search_info WHERE `count` IS NOT NULL AND `id` IS NOT NULL AN
D `id` > 10
```

好了，DataSource API V2 的 demo 到这里就介绍的差不多了。目前 DataSource API V2 还在不断演化中，不同版本的 API 可能和这里介绍的不一样，比如 Spark 2.3.x 支持分区的 API

是 createDataReaderFactories，而 Spark 2.4.x 是 planInputPartitions，详见 SPARK-24073 。同时，Apache Spark DataSource API V2 是一个比较大的 Feature ，虽然早在 Spark 2.3 版本中已经引入了，但是其实还有很多功能未发布，内置的各种数据源实现基本上都是基于 DataSource API V1 实现的；而且在 Apache Spark 2.x 版本中也不是很稳定，关于 Spark DataSource API V2 版本的稳定性工作以及新功能可以分别参见 SPARK-25186 以及 SPARK-22386 。Spark DataSource API V2 最终稳定版以及新功能将会随着年底和 Apache Spark 3.0.0 版本一起发布，其也算是 Apache Spark 3.0.0 版本的一大新功能。

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）