Spark Data Source API V1与V2简介

历史背景与演进动因

V1 API的诞生与局限性

Spark早期版本（1.x）的**V1 API**基于Hadoop生态构建，核心设计目标是兼容HDFS存储系统和传统MapReduce作业。其核心抽象`HadoopFsRelation`和`RDD`为文件型数据源提供了统一的访问接口，但存在以下问题：

- 接口冗余
  ：开发者需要同时实现`RelationProvider`、`FileFormat`、`HadoopFsRelation`等多个接口。
- 优化黑盒：Catalyst优化器无法穿透数据源内部逻辑，谓词下推（Predicate Pushdown）、列剪枝（Column Pruning）等优化需手动实现。
- 流批割裂
  ：流处理（`StreamingRelation`）和批处理（`HadoopFsRelation`）使用不同的API，导致数据源需要重复开发两套逻辑。

V2 API的设计目标

为解决V1的痛点，Spark社区在2.3版本引入**V2 API**（孵化阶段），并在3.x版本逐步成熟，其核心目标包括：

- 统一接口：通过`DataSourceV2`标准化流批处理，减少代码重复。
- 深度优化：允许Catalyst优化器与数据源直接交互，支持动态剪枝（Dynamic Pruning）、自动下推（Automatic Pushdown）等高级优化。
- 模块化扩展
  ：解耦元数据管理（`Catalog`）、扫描逻辑（`Scan`）、写入逻辑（`Write`），降低扩展成本。

V1 API架构深度解析

核心组件与执行流程

V1 API的架构围绕`HadoopFsRelation`展开，其执行流程如下：
1. 数据源注册：
实现`RelationProvider`接口，返回`BaseRelation`子类（如`HadoopFsRelation`）。
2. 逻辑计划生成：
通过`DataSource`类解析文件路径和格式，生成`LogicalRelation`节点。
3. 物理计划转换：
优化器将`LogicalRelation`转换为`FileSourceScanExec`节点，触发`FileScanRDD`执行。

关键类与接口

HadoopFsRelation

```
case class HadoopFsRelation(
    location: FileIndex,
    partitionSchema: StructType,
    // The top-
level columns in `dataSchema` should match the actual physical file schema, otherwise
    // the ORC data source may not work with the by-ordinal mode.
    dataSchema: StructType,
    bucketSpec: Option[BucketSpec],
    fileFormat: FileFormat,
    options: Map[String, String])(val sparkSession: SparkSession)
extends BaseRelation with FileRelation
```

FileFormat
定义文件读写逻辑，关键方法包括：
- `inferSchema()`：推断模式（如CSV需采样数据）。
- `buildReader()`：生成`Row`读取器（行式）或`ColumnarBatch`读取器（列式，仅Parquet支持）。

优化限制案例

假设一个查询过滤时间戳字段ts > '2023-01-01'：

- V1实现：
  `FileFormat`需主动实现`supportPushDownFilters()`并返回`true`，并在`buildReader()`中应用过滤条件。
- 问题：
  若数据源未显式实现谓词下推，Catalyst无法强制优化，导致全量数据读取。

V2 API架构与核心创新

模块化接口设计

V2 API通过角色分离实现模块化：

- **`Table`**：描述数据源的元数据（表名、模式、分区等）。
- **`ScanBuilder`**：构造`Scan`实例，支持优化器交互（如下推过滤条件）。
- **`Scan`**：定义数据扫描逻辑，支持流批两种模式。
- **`WriteBuilder`**：构造`Write`实例，处理数据写入。

```
// V2 API核心接口示例
trait Table {
  def name(): String
  def schema(): StructType
  def properties(): util.Map[String, String]
  def newScanBuilder(options: CaseInsensitiveStringMap): ScanBuilder
}

trait ScanBuilder {
  def build(): Scan
  def pushFilters(filters: Array[Filter]): Array[Filter]  // 下推过滤条件
  def pruneColumns(requiredSchema: StructType): Unit      // 列剪枝
}
```

物理计划优化示例

V2 API允许Catalyst优化器直接操作`ScanBuilder`：
1. **初始计划**：
用户提交查询`SELECT * FROM t WHERE id > 100`。
2. **优化阶段**：
Catalyst调用`pushFilters()`将`id > 100`下推至数据源。
3. **最终执行**：
`Scan`仅读取满足条件的数据，减少I/O开销。

列式读取与向量化

V2 API原生支持列式处理：
- **`SupportsColumnar`标记接口**：
数据源实现此接口后，Spark自动使用`ColumnarBatch`读取数据。
- **性能收益**：
Parquet V2连接器相比V1的TPC-DS查询性能提升30%（数据来源：Databricks基准测试）。

执行计划树深度对比

V1物理计划详解

```
== Physical Plan ==
*(1) FileScan parquet [id#0L, name#1, ts#2]
   Batched: true,
   DataFilters: [isnotnull(id), (id#0L > 100)],
   Format: Parquet,
   Location: InMemoryFileIndex[s3://bucket/path],
```

```
   PartitionFilters: [],
   PushedFilters: [IsNotNull(id), GreaterThan(id,100)],
   ReadSchema: struct<id:bigint,name:string,ts:timestamp>
```

- **关键节点**：`FileSourceScanExec`
- **优化特征**：
- `PushedFilters`显示成功下推的过滤条件。
- `Batched: true`表示使用批量读取（仅Parquet/ORC支持）。

4.2 V2物理计划详解

```
== Physical Plan ==
*(1) BatchScan parquet [id#0L, name#1, ts#2]
   Filters: [isnotnull(id#0L), (id#0L > 100)],
   ParquetScan:
      Location: InMemoryFileIndex[s3://bucket/path],
      ReadSchema: struct<id:bigint,name:string,ts:timestamp>,
      PushedFilters: [IsNotNull(id), GreaterThan(id,100)],
      DataFilters: []
```

- **关键节点**：`BatchScanExec`
- **优化特征**：
- 支持动态分区剪枝（Dynamic Partition Pruning，需`SupportsRuntimeFiltering`接口）。
- `DataFilters`为空，说明过滤条件完全下推至存储层。

最佳实践

默认情况下，spark 将为 avro,csv,json,kafka,orc,parquet,text 等数据源选择 datasource V1 ，如下：

```
scala> import spark.implicits._
import spark.implicits._

scala> val iteblogTbl= spark.read.parquet("/home/iteblog/data/hive/warehouse/iteblog_tbl/part-00000-583432a1-5272-456c-aa51-945016437b0f-c000.snappy.parquet")
iteblogTbl: org.apache.spark.sql.DataFrame = [n_nationkey: bigint, n_name: string ... 2 more fields]

scala> iteblogTbl.createOrReplaceTempView("iteblogTbl")
```

```
scala> spark.sql("select * from iteblogTbl").explain
== Physical Plan ==
*(1) ColumnarToRow
+- FileScan parquet [n_nationkey#0L,n_name#1,n_regionkey#2L,n_comment#3] Batched: true,
DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[file:/home/iteblog/data/
hive/warehouse/iteblog_tbl..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<n_natio
nkey:bigint,n_name:string,n_regionkey:bigint,n_comment:string>
```

从上可以看到使用的是 FileScan 来读取数据。如果我们要使用 datasource V2，那么在启动 spark-shell 的时候加上 --conf spark.sql.sources.useV1SourceList=avro,csv,json,kafka,orc,text，去掉了 parquet，同样运行上面代码，得到的 执行计划树如下：

```
scala> import spark.implicits._
import spark.implicits._

scala> val iteblogTbl= spark.read.parquet("/home/iteblog/data/hive/warehouse/iteblog_tbl/par
t-00000-583432a1-5272-456c-aa51-945016437b0f-c000.snappy.parquet")
iteblogTbl: org.apache.spark.sql.DataFrame = [n_nationkey: bigint, n_name: string ... 2 more fie
lds]

scala> iteblogTbl.createOrReplaceTempView("iteblogTbl")

scala> spark.sql("select * from iteblogTbl").explain
== Physical Plan ==
*(1) ColumnarToRow
+- BatchScan parquet file:/home/iteblog/data/hive/warehouse/iteblog_tbl/part-00000-583432a
1-5272-456c-aa51-945016437b0f-c000.snappy.parquet[n_nationkey#0L, n_name#1, n_regionk
ey#2L, n_comment#3] ParquetScan DataFilters: [], Format: parquet, Location: InMemoryFileIn
dex(1 paths)[file:/home/iteblog/data/hive/warehouse/iteblog_tbl..., PartitionFilters: [], PushedA
ggregation: [], PushedFilters: [], PushedGroupBy: [], ReadSchema: struct<n_nationkey:bigint,n_
name:string,n_regionkey:bigint,n_comment:string> RuntimeFilters: []
```