# Spark3.0新特性介绍与尝鲜

# 学习目标

1. 了解Spark发展史
2. 了解Spark3.0新特性
3. 掌握Spark3.0动态分区裁剪原理
4. 了解Spark未来发展方向

# Spark发展历史与现状

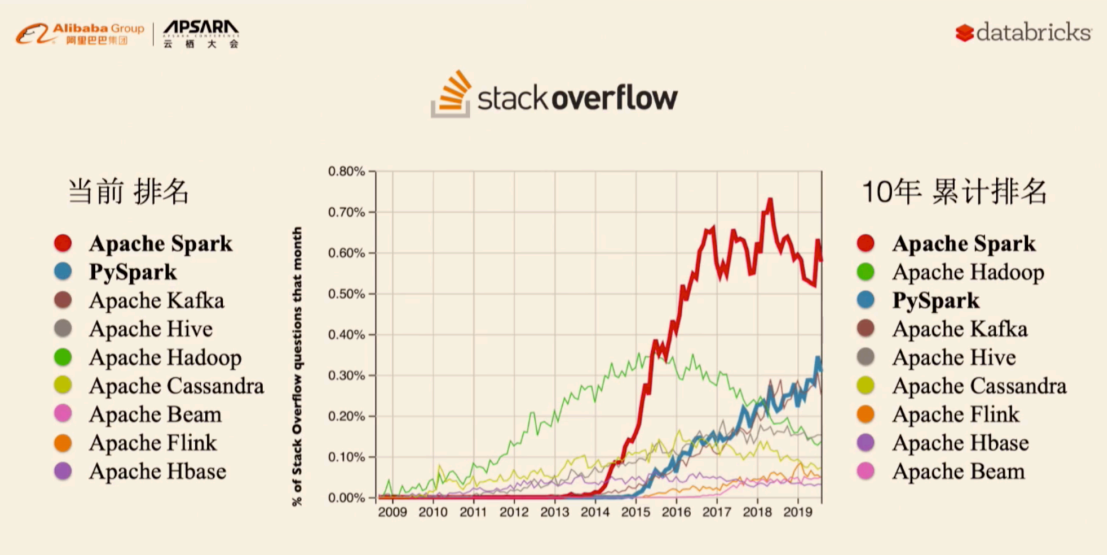
## Spark风雨十年

自2009年伯克利的AMP实验室将 Spark开源以来，Spark在大数据处理领域获得了巨大的成功。Spark的定位是统一的通用的分布式大数据处理分析引擎，基于这个分布式计算引擎, Spark在不同领域方向提供了高层的DSL，比如针对关系型数据的Spark SQL，针对实时数据的Spark streaming，针对机器学习的MLlib, 以及针对图计算的GraphX。

Spark的发展相当迅速，2013年其原创团队建立了Databricks，将Spark捐献给了Apache Foundation；2014年Spark成为了Apache顶级项目；2016年发布了Spark 2.0；2019年10月Spark 3.0 preview版本发布，后续将收集用户反馈后发布Spark 3.0正式版本。



从世界著名的开发者论坛Stack Overflow的数据可以看出，2015年开始Spark每月的问题提交数量已经超越Hadoop，而2018年Spark Python版本的API PySpark每月的问题提交数量也已超过Hadoop。当前排名Spark第一，PySpark第二；而十年的累计排名是Spark第一，PySpark第三。按照这个趋势发展下去，Spark和PySpark在未来很长一段时间内应该还会处于垄断地位。



## Spark流行原因

Spark获得大数据领域的认可的原因概括起来有如下三方面:

第一， 它提供了易用的API，支持Java、Scala、Python, R、SQL等多种语言，这使得数据工程师和数据科学家都能够使用他们最熟悉的语言访问Spark。此外，Spark也提供了声明式的DSL，在API的灵活性和应用性之间达到了平衡。

第二， Spark是目前大数据领域生态系统最丰富的组件，支持ORC、Parquet等数据存储格式，Kafka消息队列以及多种资源调度的框架，几乎所有与数据处理上下游相关的组件都可以在Spark上找到官方或者非官方的集成支持。

第三， Spark有非常高效的处理引擎，它的Catalyst和Tungsten项目支持了对于数据分析性能至关重要的两大特性。其一，内存管理，Spark通过自定义的内存管理，把所有在Spark上处理的数据转化为二进制数据再进行管理，Spark计算直接操作二进制数据而不是对象，这使得内存的使用高效和可控，也很大程度上避免了JVM进程的OOM或者GC问题。其二，代码生成。通过Runtime的CodeGen，Spark消除了传统数据分析中火山模型由于调用链过长而导致的调用代价问题。

# 初识Spark3.0

## 从2.0到3.0



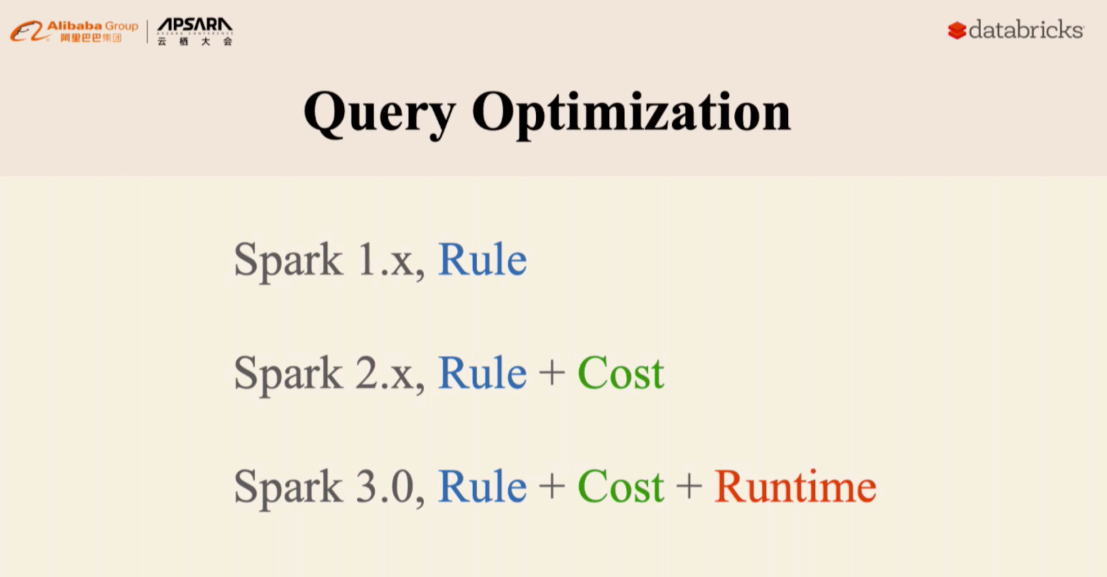
Spark 2.X中增加了基于计算的Cost模型，但是在Spark应用环境中其表现并不如意，主要原因是：

1.一次性计算：Spark 2.X基本的应用场景是ETL，一次性计算，收集数据成本昂贵，统计信息的缺失导致基于Cost的优化基本不可能完成；

2.存储于计算分离：Spark最初设计的原则就是不拥有数据，这意味着用户可以用不同的方式增加、删除和修改，而如果统计信息错误，就无法保证基于cost的优化的正确性，甚至优化后的结果可能更差；

3.多样部署环境：基于Cost的优化依赖Cost模型，而Cost模型在Spark的运行环境是多样的，如云原生、分布式、大型机等，因此无法提出一套普遍适用于多种生态环境的Cost模型。此外，Spark中比较重要的功能UDFs支持用户根据自己需要任意加载复杂或简单的UDF实现其业务逻辑。这种情况下是完全无法计算其Cost的，相应地，优化器也就无法实现一个正确有效的优化计划。

针对上述问题，Spark 3.0版本在cost基础上收集应用在运行期间的统计信息，来帮助动态优化运行的计划。

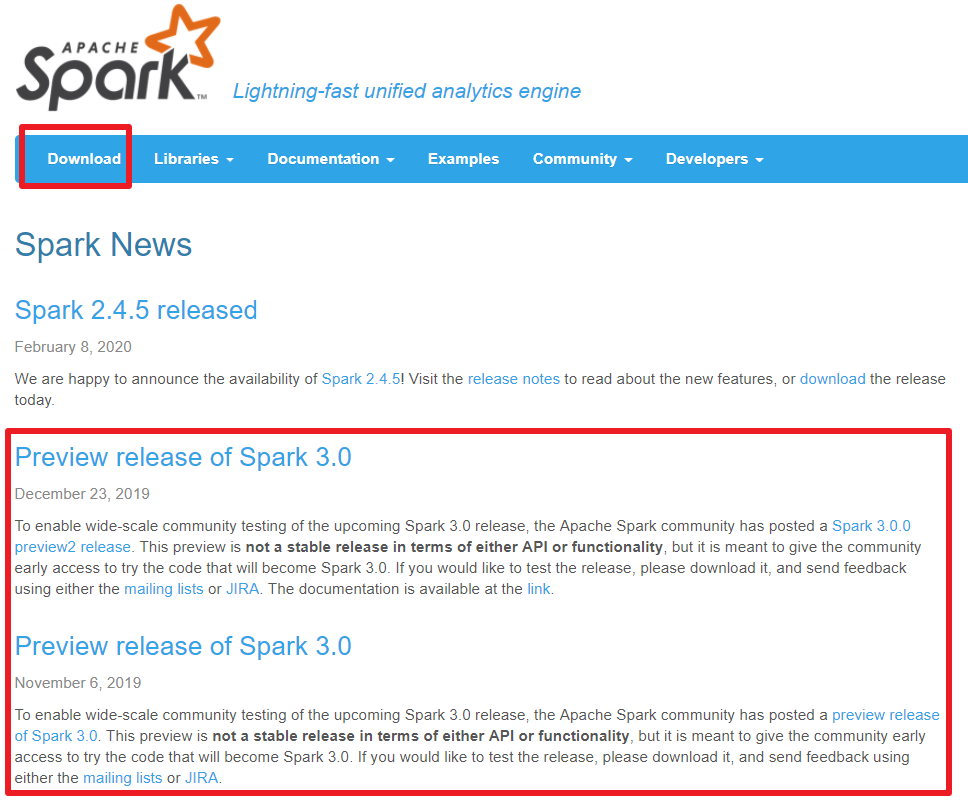


## Spark3.0诞生

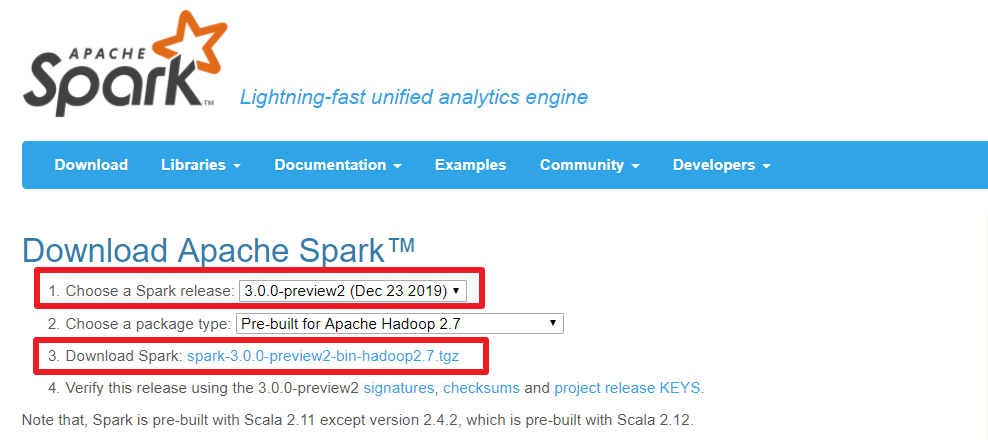
2019年11月08日 Databricks数砖的 Xingbo Jiang 大佬给社区发了一封邮件，宣布 Apache Spark 3.0 预览版正式发布，这个版本主要是为了对即将发布的 Apache Spark 3.0 版本进行大规模社区测试。

注意:无论是从 API 还是从功能上来说，这个预览版都不是一个稳定的版本，它的主要目的是为了让社区提前尝试 Apache Spark 3.0 的新特性。如果想测试这个版本，可以到这里下载。

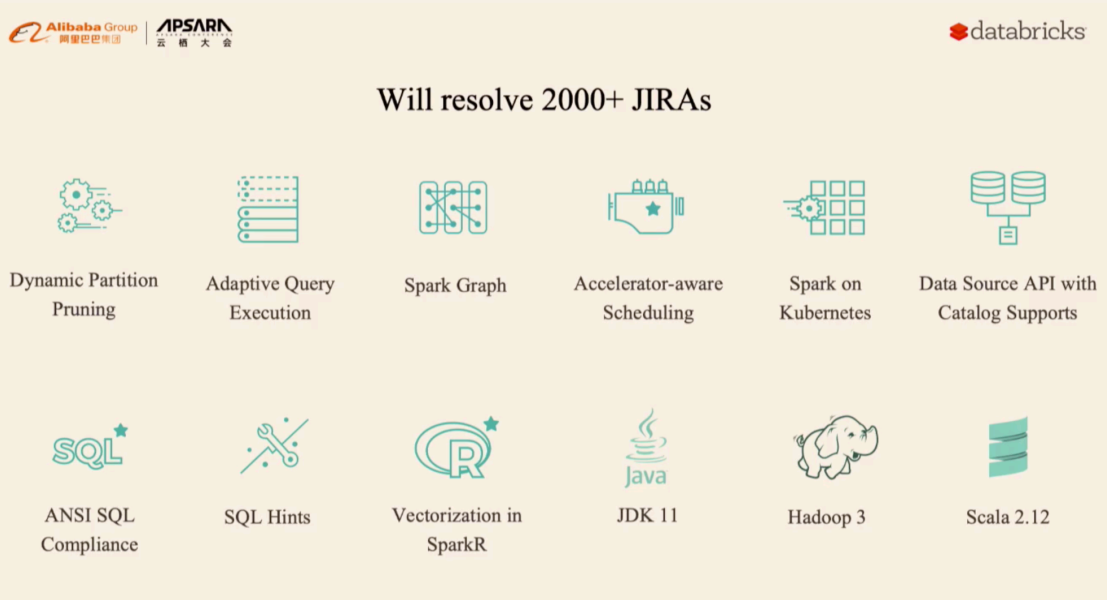
<http://spark.apache.org/news/>



<http://spark.apache.org/downloads.html>

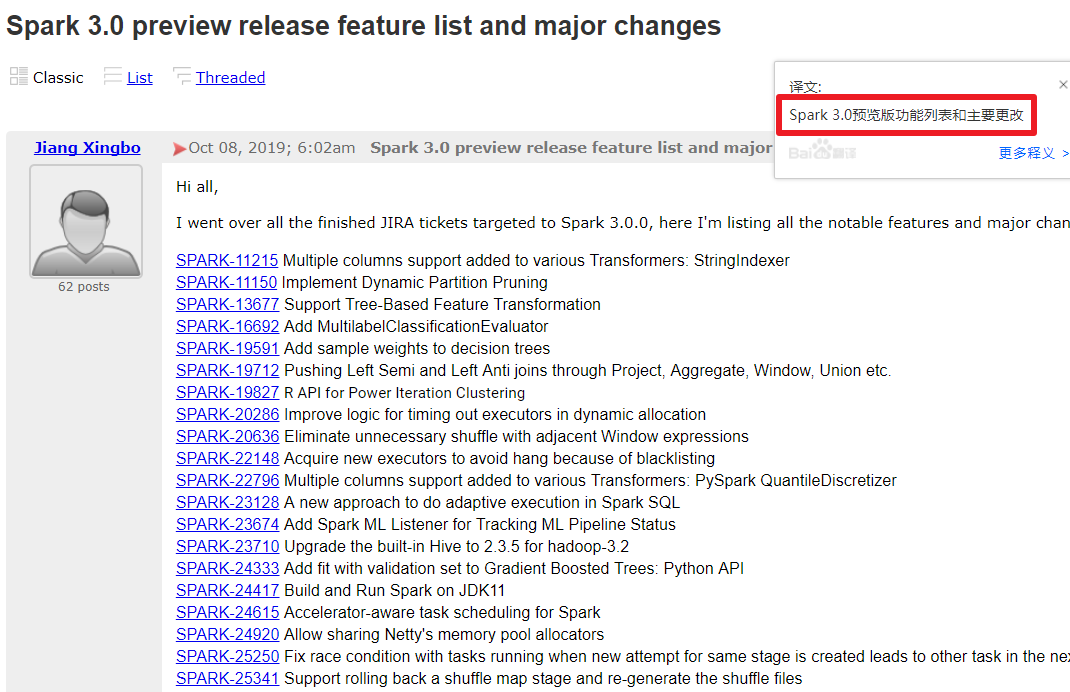


## Spark3.0新特性简介



Spark 3.0 增加了很多令人兴奋的新特性，包括动态分区修剪（Dynamic Partition Pruning）、自适应查询执行（Adaptive Query Execution）、加速器感知调度（Accelerator-aware Scheduling）、支持 Catalog 的数据源API（Data Source API with Catalog Supports）、SparkR 中的向量化（Vectorization in SparkR）、支持 Hadoop 3/JDK 11/Scala 2.12 等等。Spark 3.0.0-preview 中主要特性和变化的完整列表请参阅链接。下面一起了解一下比较重要的新特性。

<http://apache-spark-developers-list.1001551.n3.nabble.com/Spark-3-0-preview-release-feature-list-and-major-changes-td28050.html>



PS：仔细观察的同学可以看出，Spark 3.0 好像没多少 Streaming/Structed Streaming 相关的 ISSUE，这可能有几个原因：

1.目前基于 Batch 模式的 Spark Streaming/Structed Streaming 能够满足企业大部分的需求，真正需要非常实时计算的应用还是很少的，所以 Continuous Processing 模块还处于试验阶段，还不急着毕业；

2.Databricks应该在大量投人开发 Delta Lake 相关的东西，这个能够企业带来收入，目前这个才是他们的重点，所以自然开发 Streaming 的投入少了。

### 动态分区裁剪（Dynamic Partition Pruning）

在3.0以前，spark是不支持动态分区的，所谓动态分区就是针对分区表中多个表进行join的时候基于运行时（runtime）推断出来的信息，在on后面的条件语句满足一定的要求后就会进行自动动态分区裁减优化，比如：

SELECT t1.id,t2.pKey

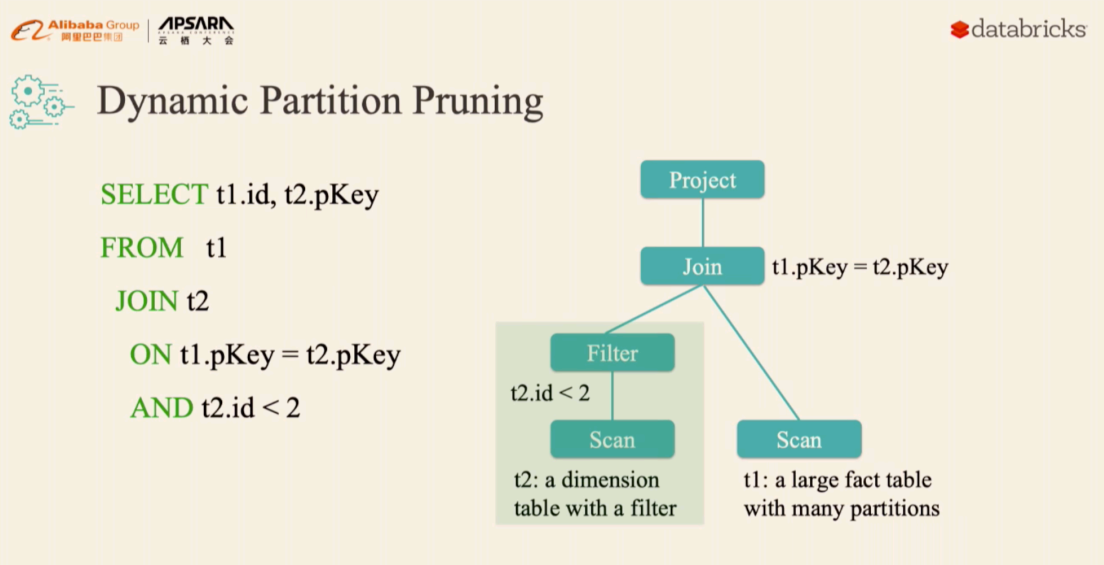
FROM t1

JOIN t2

ON t1.pKey = t2.pKey

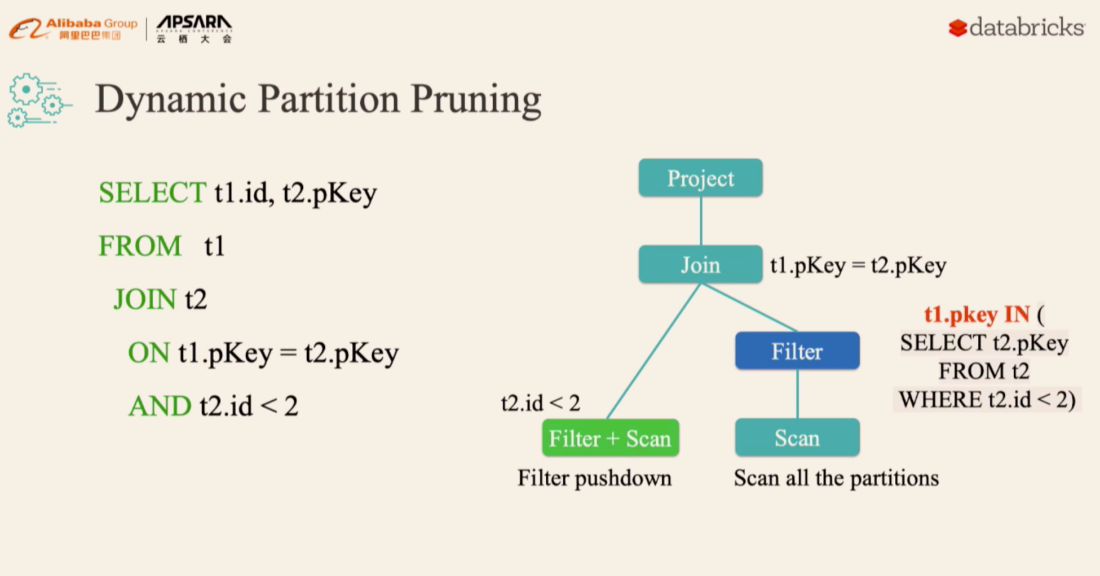
AND t2.id < 2;

上面这条SQL语句在没有使用动态分区的情况下执行过程如下图所示：

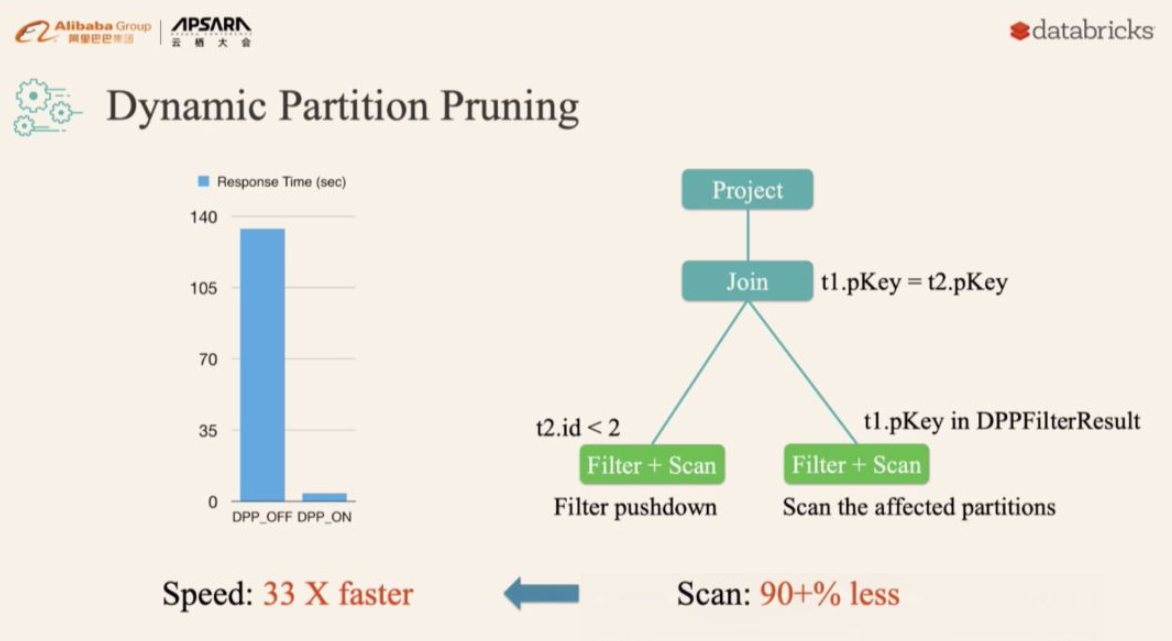


由于之前版本的 Spark 无法进行动态计算代价，所以可能会导致t1 表扫描出大量无效的数据。

在使用了动态分区以后执行过程变成了下图：



可以看到之前进行join的时候对t1表中满足on条件的所有数据进行扫描，然后再把两张表进行join，在加入动态分区之后，我们在运行的时候过滤掉了t1表中的无用数据，经过这个优化，查询扫描的数据大大减少，也减少了join时的内存计算开销，在实际环境中同样的代码，性能提升了有33倍！



### 自适应查询执行（Adaptive Query Execution）

自适应查询是指对执行计划按照实际数据分布和组织情况，评估其执行所消耗的时间和资源，从而选择代价最小的计划去执行。一般数据库的优化器有两种，一种是基于规则的优化器(RBO)，一种是基于代价的优化器(CBO),自适应查询指的就是对CBO的优化，SparkSQL的运行流程主要有：

1.将SQL语句通过词法和语法解析生成未绑定的逻辑执行计划；

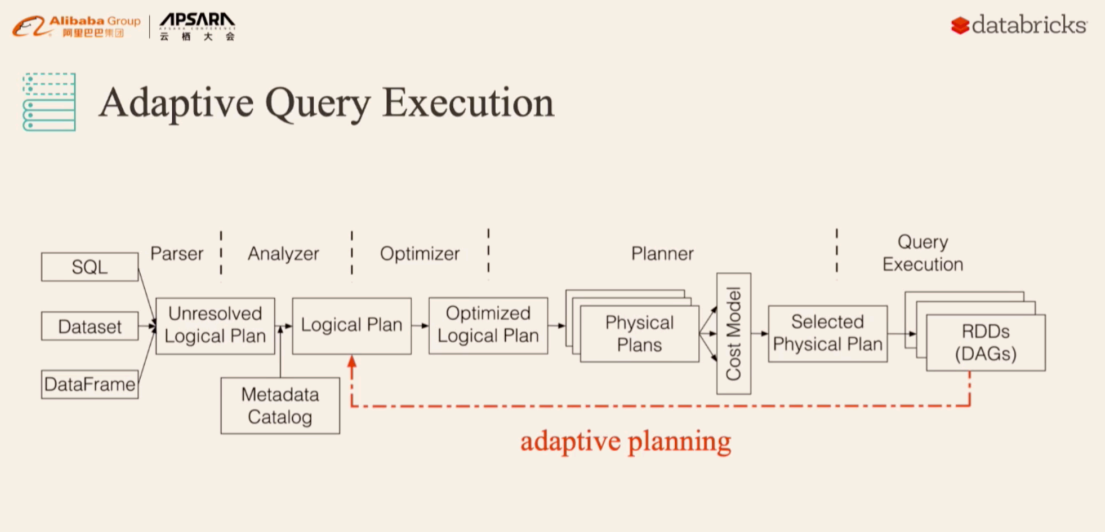
2.分析器配合元数据使用分析规则，完善未绑定的逻辑计划属性转换成绑定的逻辑计划；

3.优化器使用优化规则将绑定的逻辑计划进行合并、裁减、下推生成优化的逻辑计划；

4.规划器使用规划策略把优化的逻辑计划转换成可执行的物理计划；

5.进行preparations规则处理，构建RDD的DAG图执行查询计划。

Spark以前的调度规则是执行计划一旦确定，即使发现后续执行计划可以优化，也不可更改，而自适应查询功能则是在执行查询计划的同时，基于表和列的统计信息，对各个算子产生的中间结果集大小进行估算，根据估算结果来动态地选择最优执行计划，如下图：



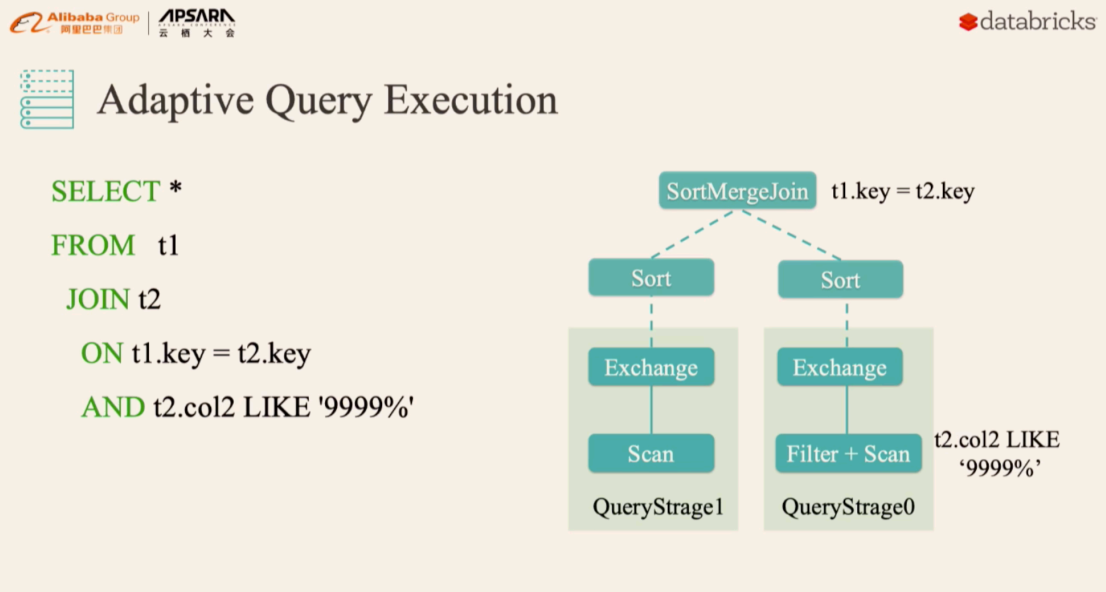
上图是一个经典的Spark的流程，从Parser、Analyzer、Optimizer、Planner到Query的执行。该版本中，AQE指的是图中的红线部分。当某种condition满足的情况下可以进行动态自适应规划。

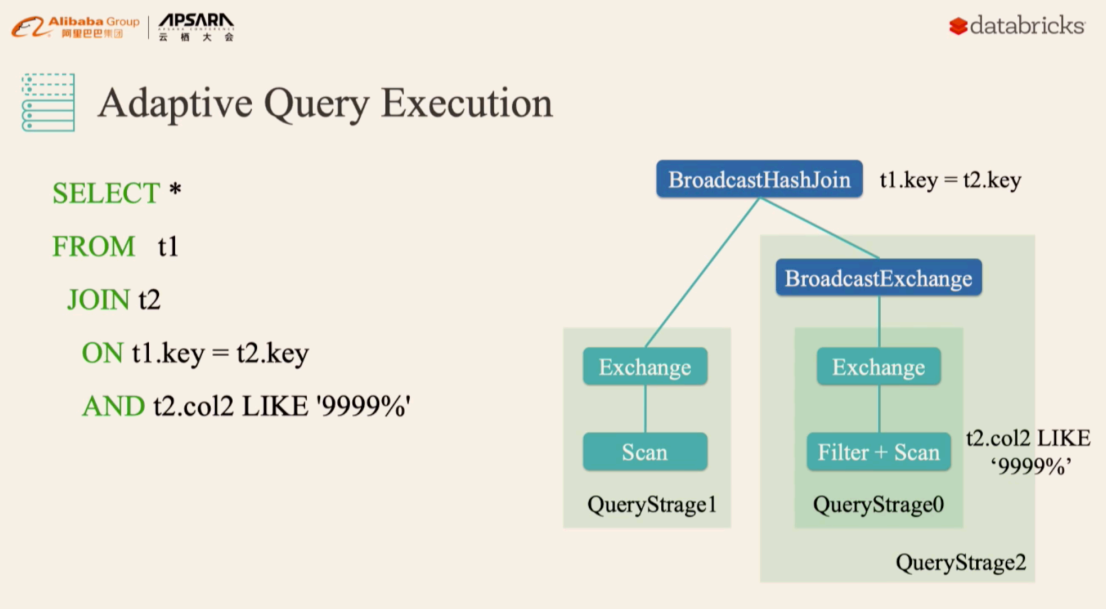
下面举一个简单的例子，执行的是两个表之间的join查询。包含一个key和Filter，如t2.col2 LIKE '9999%'。

在基于cost的模型中是不可能准确的知道Filter能排除多少行的，这种情况下Spark通过谓词下推，将各个条件先应用到对应的数据上，而不是根据写入的顺序执行，就可以先过滤掉部分数据，降低join等一系列操作的数据量级。

在没有动态实时运行信息的时候，保守估计判断只能用SortMergeJoin。

当收集到运行时信息后会发现某个Filter事实上已经去掉了表中绝大多数行，完全可以采用BroadcastHashJoin，如果上层parent也有这种情况，就可以大大提升查询效率。





### 加速器感知调度（Accelerator-aware Scheduling）



如今大数据和机器学习已经有了很大的结合，在机器学习里面，因为计算迭代的时间可能会很长，开发人员一般会选择使用 GPU、FPGA 或 TPU 来加速计算。

在 Apache Hadoop 3.1 版本里面已经开始内置原生支持 GPU 和 FPGA 了。作为通用计算引擎的 Spark 肯定也不甘落后，来自 Databricks、NVIDIA、Google 以及阿里巴巴的工程师们正在为 Apache Spark 添加原生的 GPU 调度支持，该方案填补了 Spark 在 GPU 资源的任务调度方面的空白，有机地融合了大数据处理和 AI 应用，扩展了 Spark 在深度学习、信号处理和各大数据应用的应用场景。

为了让 Spark 也支持 GPUs，在技术层面上需要做出两个主要改变：

1.在 cluster manager 层面上，需要升级 cluster managers 来支持 GPU。并且给用户提供相关 API，使得用户可以控制 GPU 资源的使用和分配。

2.在 Spark 内部，需要在 scheduler 层面做出修改，使得 scheduler 可以在用户 task 请求中识别 GPU 的需求，然后根据 executor 上的 GPU 供给来完成分配。

因为让 Apache Spark 支持 GPU 是一个比较大的特性，所以项目分为了几个阶段。

1.在 Apache Spark 3.0 版本，将支持在 standalone、 YARN 以及 Kubernetes 资源管理器下支持 GPU，并且对现有正常的作业基本没影响。

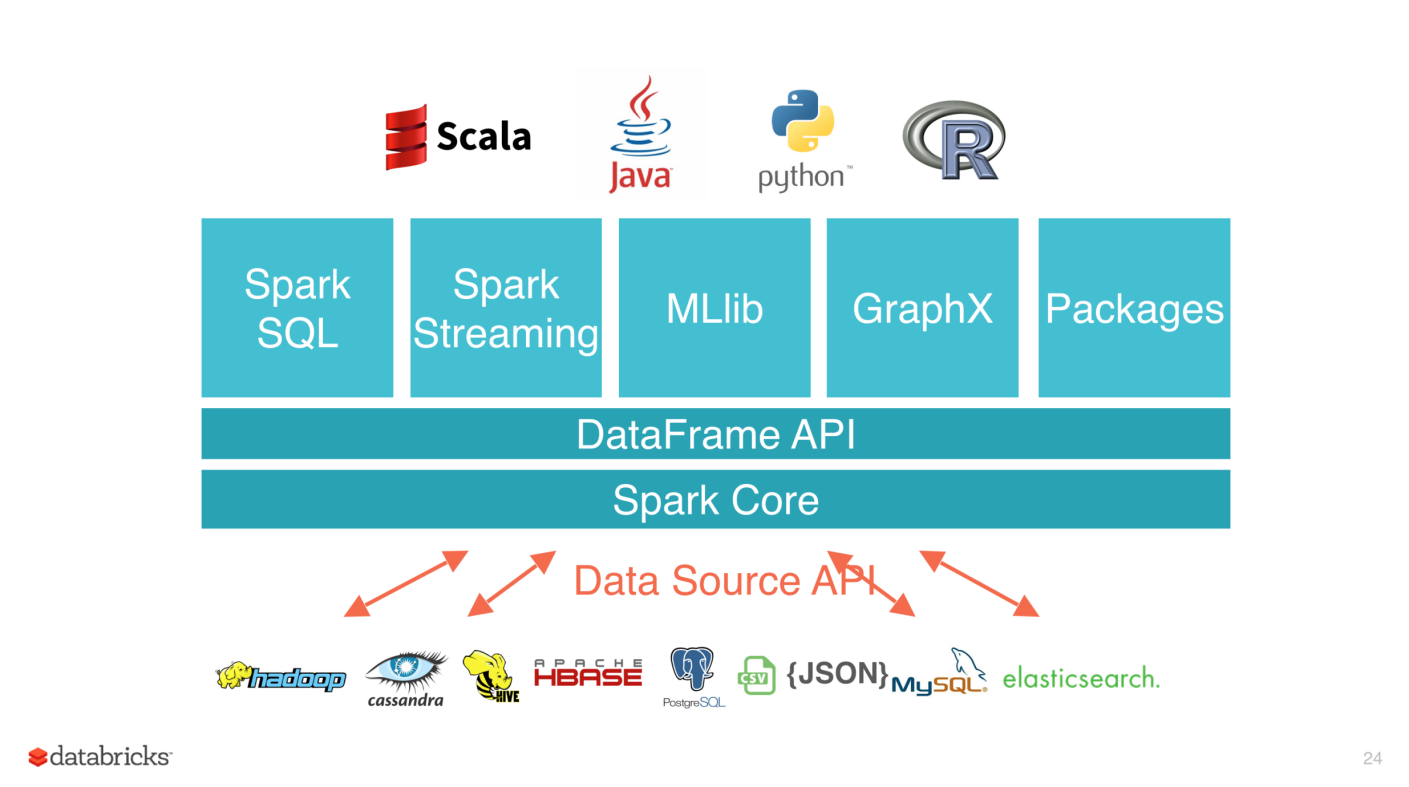
2.对于 TPU 的支持、Mesos 资源管理器中 GPU 的支持、以及 Windows 平台的 GPU 支持将不是这个版本的目标。

3.而且对于一张 GPU 卡内的细粒度调度也不会在这个版本支持；Spark 3.0 版本将把一张 GPU 卡和其内存作为不可分割的单元。

### 更好的API扩展(BetterAPI-Extensions-DataSourceV2)

Data Source API 定义如何从存储系统进行读写的相关 API 接口，比如 Hadoop 的 InputFormat/OutputFormat，Hive 的 Serde 等。这些 API 非常适合用户在 Spark 中使用 RDD 编程的时候使用。使用这些 API 进行编程虽然能够解决我们的问题，但是对用户来说使用成本还是挺高的，而且 Spark 也不能对其进行优化。

为了解决这些问题，Spark 1.3 版本开始引入了 Data Source API V1，通过这个 API 我们可以很方便的读取各种来源的数据，而且 Spark 使用 SQL 组件的一些优化引擎对数据源的读取进行优化，比如列裁剪、过滤下推等等。



Data Source API V1 为我们抽象了一系列的接口，使用这些接口可以实现大部分的场景。但是随着使用的用户增多，逐渐显现出一些问题：



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。虽然这个功能在 Apache Spark 2.x 版本就出现了，但是不是很稳定，所以社区对 Spark DataSource API V2 的稳定性工作以及新功能分别开了两个 ISSUE：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Spark 3.0版本的一大新功能。

### 更好的ANSI-SQL兼容（ANSI SQL Compatible）

因为Spark原来的SQL语法和函数跟ANSI标准还是存在一些差异，因此这次版本更新将缩小和ANSI标准之间的差异，包括增加一些ANSI SQL的函数、区分SQL保留关键字以及内置函数等，如果这部分ISSUE解决，那么Spark一定会得到更广泛的应用。

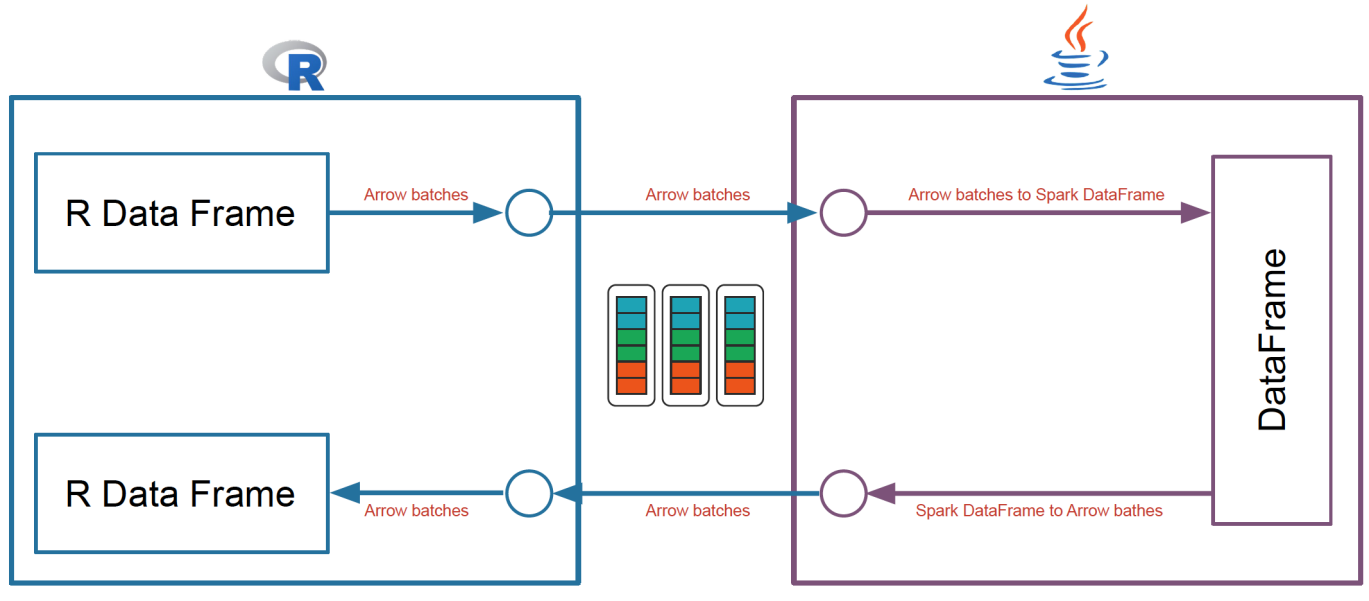
### SparkR向量化读写（Vectorization）

Spark 是从 1.4 版本开始支持 R 语言的，但是那时候 Spark 和 R 进行交互的架构图如下：



每当我们使用 R 语言和 Spark 集群进行交互，需要经过 JVM ，这也就无法避免数据的序列化和反序列化操作，这在数据量很大的情况下性能是十分低下的！而Spark 已经在许多操作中进行了向量化优化（vectorization optimization），例如，内部列式格式（columnar format）、Parquet/ORC 向量化读取、Pandas UDFs 等。向量化可以大大提高性能。

SparkR 向量化允许用户按原样使用现有代码，但是当他们执行 R 本地函数或将 Spark DataFrame 与 R DataFrame 互相转换时，可以将性能提高大约数千倍。这项工作可以看下 SPARK-26759。新的架构如下:



可以看出，SparkR 向量化是利用 Apache Arrow，其使得系统之间数据的交互变得很高效，而且避免了数据的序列化和反序列化的消耗，所以采用了这个之后，SparkR 和 Spark 交互的性能得到极大提升。

# Spark3.0初体验

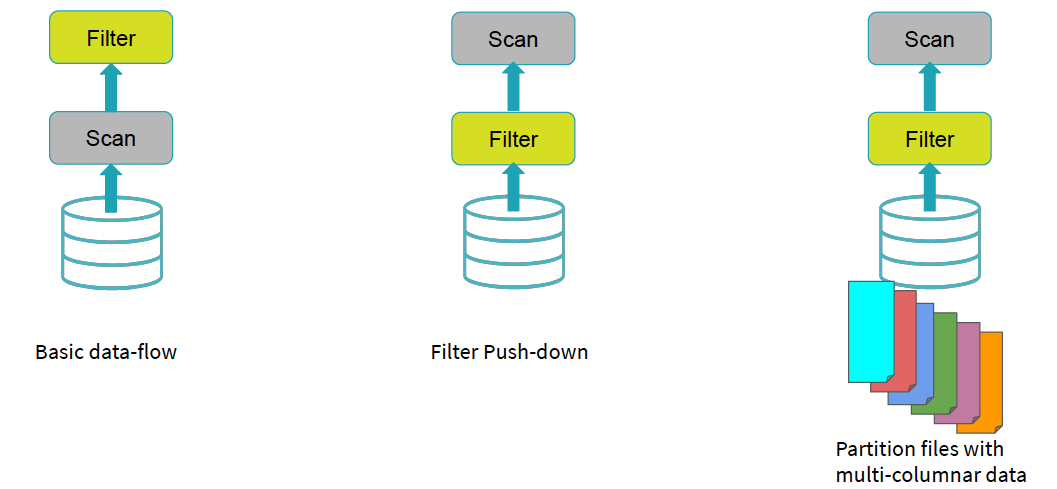
Spark3.0的新特性在前面已经做过简单介绍，接下来给大家详细讲解并体验一下其中比较具有代表性的新特性：动态分区裁剪

## 静态分区裁剪

用过 [Spark](https://www.iteblog.com/archives/tag/spark/" \o "" \t "https://www.iteblog.com/archives/_blank) 的同学都知道，[Spark](https://www.iteblog.com/archives/tag/spark/" \o "" \t "https://www.iteblog.com/archives/_blank) SQL 在查询的时候支持分区裁剪，比如我们如果有以下的查询：

SELECT \* FROM t\_sales WHERE day\_of\_week = 'Mon'

[Spark](https://www.iteblog.com/archives/tag/spark/" \o "" \t "https://www.iteblog.com/archives/_blank) 会自动进行以下的优化：

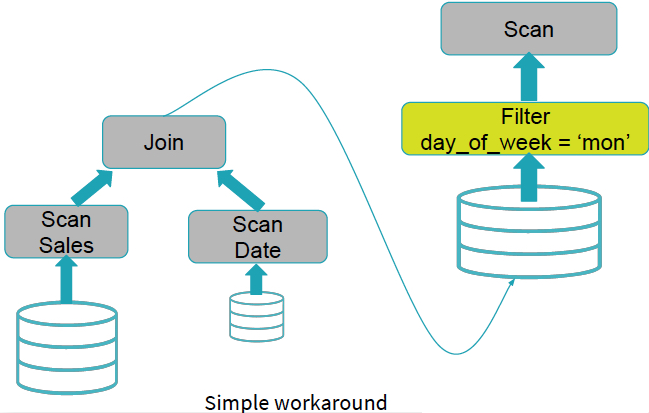
[](https://www.iteblog.com/pic/spark/Spark_static_partition_pruning_iteblog.jpg)

从上图可以看到，Spark 在编译 SQL 的时候自动将 Filter 算子下推到数据源，也就是在 Scan 前进行了 Filter 操作，将 day\_of\_week = 'Mon' 的数据全部拿出来，其他数据不需要的拿出，这样 Spark SQL 中处理的数据就变少了，整个 SQL 的查询数据就会变快，这一切都是编译的时候（compile time）进行的，所以这个叫做静态分区裁剪（Static Partition Pruning）。

上面的 SQL 查询在 Spark 进行了算子下推，已经能够满足我们的查询性能的提升。但是现实世界数据的查询可不会都这么简单，我们来看看下面的查询语句：

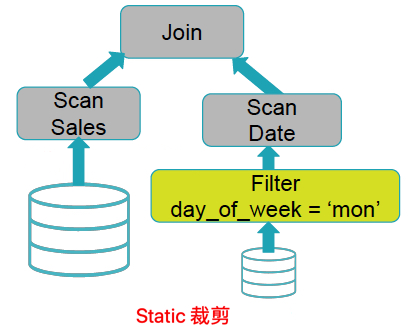
SELECT \* FROM t\_sales事实表 JOIN Date维度表 WHERE Date.day\_of\_week = 'Mon'

1.比较差的查询引擎是这么做的：



从上图可以看出，查询引擎直接忽略了 Date.day\_of\_week = 'Mon' 这个过滤条件，上来就是两表 join，然后 join 的结果再进行过滤，这个可能导致很多无效的计算，如果 Date.day\_of\_week = 'Mon' 可以过滤掉大量的无用数据，肯定可以提升查询性能。

2.在 Spark SQL 里面能够很好的处理这种情况，它会把 Date.day\_of\_week = 'Mon' 过滤条件下推到 Date 表的 Scan 之前进行：



## 动态分区裁剪

上面 Spark SQL 进行的算子下推还能不能再提升查询性能呢？有没有一种更好的方法进一步过滤掉一些无用的数据？这就是接下来要介绍的动态分区裁剪。

什么是动态分区裁剪？

所谓的动态分区裁剪就是基于运行时（runtime）推断出来的信息来进一步进行分区裁剪。

举个例子，我们有如下的查询：

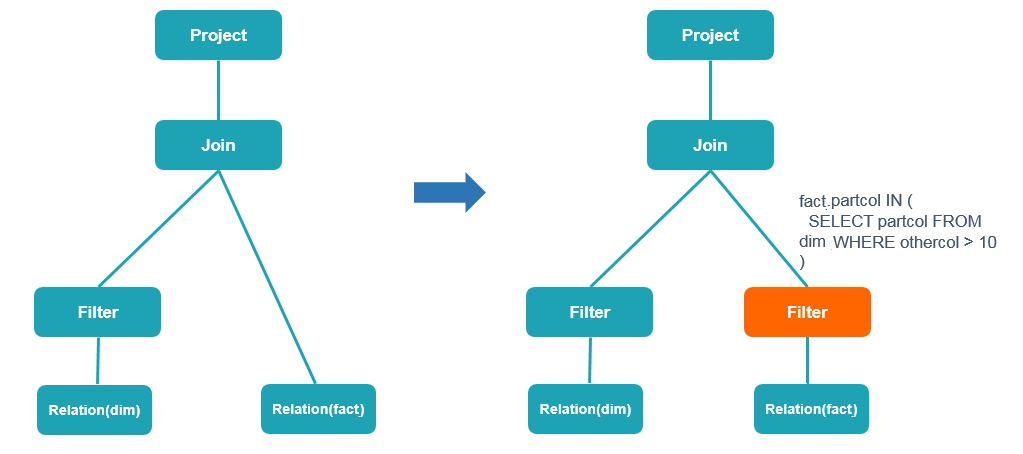
SELECT \* FROM t\_dim

JOIN t\_fact

ON (t\_dim.partcol = t\_fact.partcol)

WHERE t\_dim.othercol > 10

动态查询的执行计划如下：



从上面可以看出，拥有动态分区裁剪，Spark 能够在运行的时候先对 t\_fact表的 partcol 进行了一次过滤，然后再和 t\_dim 表进行 Join，可想而知这个性能一般都会有提升的，特别是在 t\_fact表有很多无用的数据时性能提升会非常大的。

Databricks 公司在10台配置为 i3.xlarge 的集群上进行 TPC-DS 测试，得到的结论是:

在 102 查询中相比 Spark 2.4 有 60 个查询的查询性能提升了 2 - 18 倍的提升。在 Query 98 查询中，性能提升了 100 倍！

## 如何开启

### 相关配置

要启用动态分区裁剪需要将

spark.sql.optimizer.dynamicPartitionPruning.enabled 参数设置为 true（默认为 true）

其他相关参数：

1.spark.sql.optimizer.dynamicPartitionPruning.useStats：默认为 true

When true, distinct count statistics will be used for computing the data size of the partitioned table after dynamic partition pruning, in order to evaluate if it is worth adding an extra subquery as the pruning filter if broadcast reuse is not applicable.

2.spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio：默认0.5

When statistics are not available or configured not to be used, this config will be used as the fallback filter ratio for computing the data size of the partitioned table after dynamic partition pruning, in order to evaluate if it is worth adding an extra subquery as the pruning filter if broadcast reuse is not applicable.

3.spark.sql.optimizer.dynamicPartitionPruning.reuseBroadcast：默认为 true

When true, dynamic partition pruning will seek to reuse the broadcast results from a broadcast hash join operation.

### 注意事项

需要注意并不是什么查询都会启用动态裁剪优化的，必须满足以下几个条件：

1.spark.sql.optimizer.dynamicPartitionPruning.enabled 参数必须设置为 true，不过这个值默认就是启用的；

2.需要裁减的表必须是分区表，而且分区字段必须在 join 的 on 条件里面；

3.Join 类型必须是 INNER, LEFT SEMI （左表是分区表）, LEFT OUTER （右表是分区表）, or RIGHT OUTER （左表是分区表）。

4.满足上面的条件也不一定会触发动态分区裁减，还必须满足 spark.sql.optimizer.dynamicPartitionPruning.useStats 和 spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio 两个参数综合评估出一个进行动态分区裁减是否有益的值，满足了才会进行动态分区裁减。

## 如何使用

### pom.xml

|  |
| --- |
| *<!-- 指定仓库位置，依次为aliyun、cloudera和jboss仓库 -->* <**repositories**>  <**repository**>  <**id**>aliyun</**id**>  <**url**>http://maven.aliyun.com/nexus/content/groups/public/</**url**>  </**repository**>  <**repository**>  <**id**>cloudera</**id**>  <**url**>https://repository.cloudera.com/artifactory/cloudera-repos/</**url**>  </**repository**>  <**repository**>  <**id**>jboss</**id**>  <**url**>http://repository.jboss.com/nexus/content/groups/public</**url**>  </**repository**> </**repositories**> <**properties**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  <**encoding**>UTF-8</**encoding**>  <**scala.version**>2.12.11</**scala.version**>  *<!-- <spark.version>2.2.0</spark.version>-->* </**properties**> <**dependencies**>  <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-library</**artifactId**>  <**version**>${scala.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.12</**artifactId**>  <**version**>3.0.0-preview2</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.12</**artifactId**>  <**version**>3.0.0-preview2</**version**>  </**dependency**> </**dependencies**> <**build**>  <**sourceDirectory**>src/main/scala</**sourceDirectory**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.5.1</**version**>  </**plugin**>  <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  <**version**>3.2.2</**version**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>compile</**goal**>  <**goal**>testCompile</**goal**>  </**goals**>  <**configuration**>  <**args**>  <**arg**>-dependencyfile</**arg**>  <**arg**>${project.build.directory}/.scala\_dependencies</**arg**>  </**args**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-surefire-plugin</**artifactId**>  <**version**>2.18.1</**version**>  <**configuration**>  <**useFile**>false</**useFile**>  <**disableXmlReport**>true</**disableXmlReport**>  <**includes**>  <**include**>\*\*/\*Test.\*</**include**>  <**include**>\*\*/\*Suite.\*</**include**>  </**includes**>  </**configuration**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-shade-plugin</**artifactId**>  <**version**>2.3</**version**>  <**executions**>  <**execution**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>shade</**goal**>  </**goals**>  <**configuration**>  <**filters**>  <**filter**>  <**artifact**>\*:\*</**artifact**>  <**excludes**>  <**exclude**>META-INF/\*.SF</**exclude**>  <**exclude**>META-INF/\*.DSA</**exclude**>  <**exclude**>META-INF/\*.RSA</**exclude**>  </**excludes**>  </**filter**>  </**filters**>  <**transformers**>  <**transformer** implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  <**mainClass**></**mainClass**>  </**transformer**>  </**transformers**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  </**plugins**> </**build**> |

### 测试代码

我们先使用 [Spark](https://www.iteblog.com/archives/tag/spark/" \o "" \t "https://www.iteblog.com/archives/_blank) 创建两张表：

tab1 表产生 10000 个分区，tab2 表产生 100 个分区。

并执行查询语句：

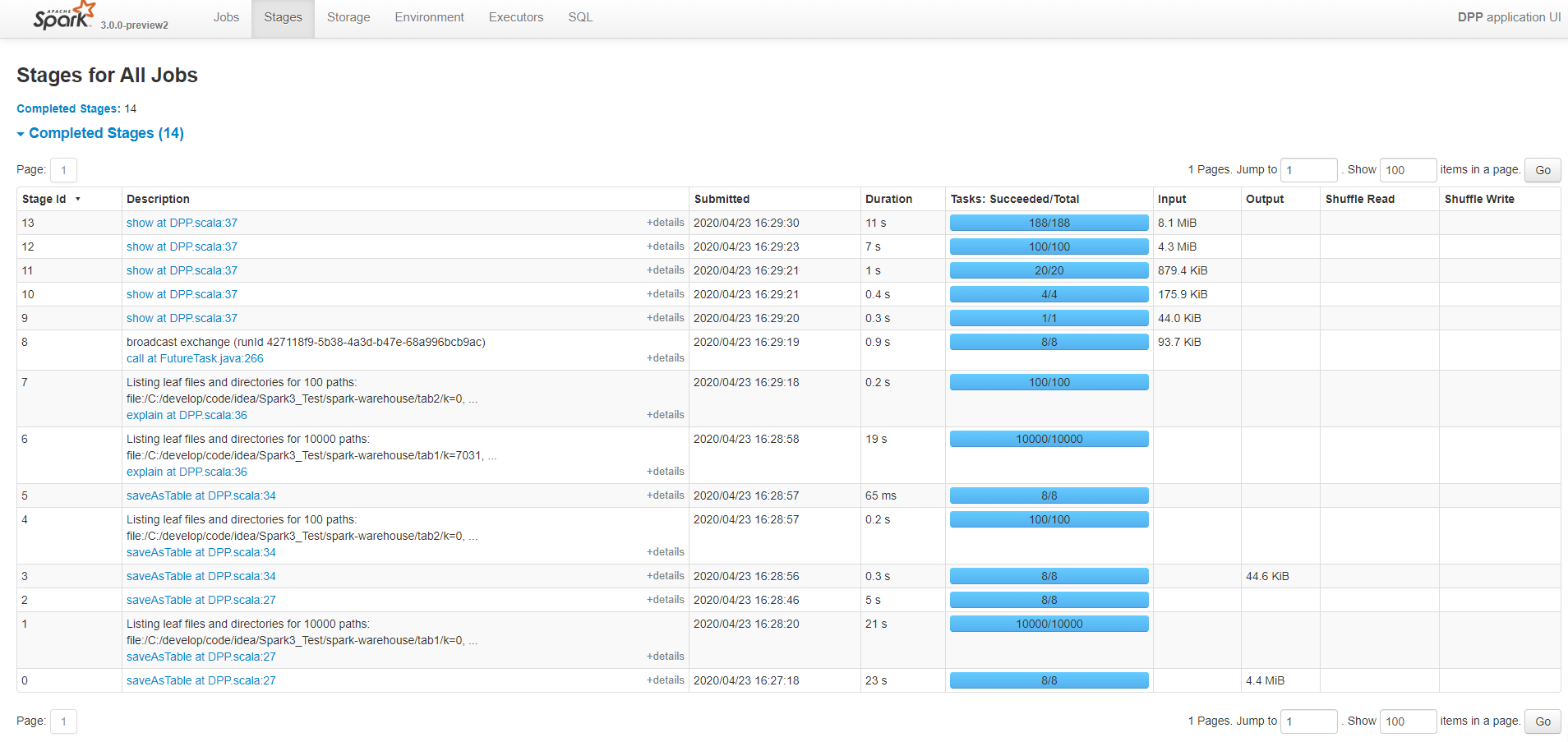
|  |
| --- |
| **package** cn.itcast.hello  **import** org.apache.spark.SparkContext **import** org.apache.spark.sql.SparkSession  */\*\*  \* Author itcast  \* 动态分区裁剪（Dynamic Partition Pruning）  \*/* **object DPP** {  **def** main(args: **Array**[String]): Unit = {  **val** spark: **SparkSession** = **SparkSession**.*builder*()  .master("local[\*]")  .appName("DPP")  .config("spark.sql.optimizer.dynamicPartitionPruning.enabled", "false")*//默认为true* .getOrCreate()  **val** sc: **SparkContext** = spark.sparkContext  sc.setLogLevel("WARN")   **import** spark.**implicits**.\_   spark.range(10000)  .select($"id", $"id".as("k"))  .write.partitionBy("k")  .format("parquet")  .mode("overwrite")  .saveAsTable("tab1")   spark.range(100)  .select($"id", $"id".as("k"))  .write.partitionBy("k")  .format("parquet")  .mode("overwrite")  .saveAsTable("tab2")   spark.sql("SELECT \* FROM tab1 t1 JOIN tab2 t2 ON t1.k = t2.k AND t2.id < 2").explain()  spark.sql("SELECT \* FROM tab1 t1 JOIN tab2 t2 ON t1.k = t2.k AND t2.id < 2").show()   **Thread**.*sleep*(Long.*MaxValue*)   sc.stop()  spark.stop()  } } |

### 没有启用动态分区裁剪

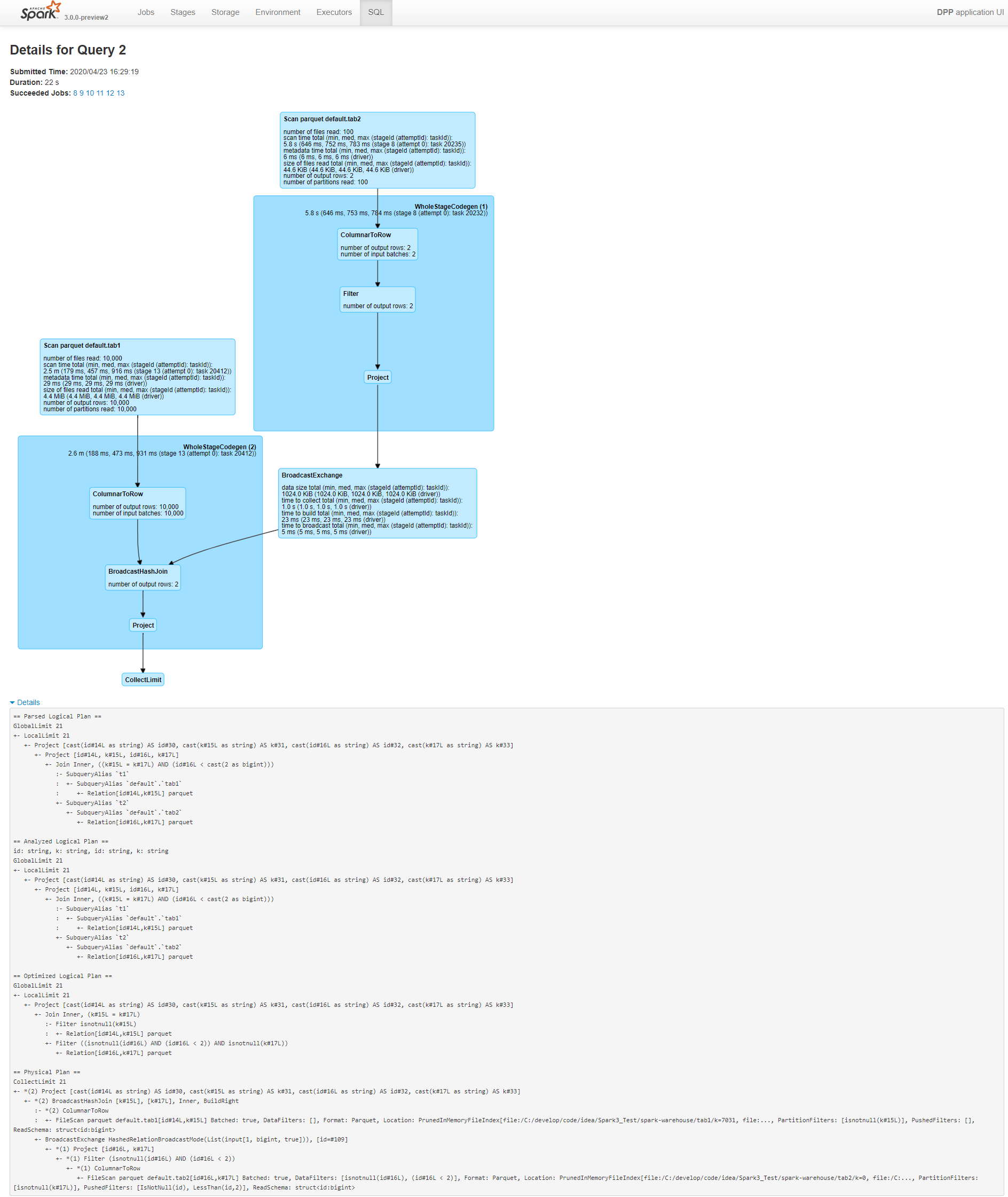
运行完上面的代码之后，

在没有启用动态分区裁剪的情况下执行结果如下：

|  |
| --- |
| == Physical Plan ==  \*(2) BroadcastHashJoin [k#15L], [k#17L], Inner, BuildRight  :- \*(2) ColumnarToRow  : +- FileScan parquet default.tab1[id#14L,k#15L] Batched: true, DataFilters: [], Format: Parquet, Location: PrunedInMemoryFileIndex[file:/C:/develop/code/idea/Spark3\_Test/spark-warehouse/tab1/k=7031, file:..., PartitionFilters: [isnotnull(k#15L)], PushedFilters: [], ReadSchema: struct<id:bigint>  +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, true])), [id=#64]  +- \*(1) Project [id#16L, k#17L]  +- \*(1) Filter (isnotnull(id#16L) AND (id#16L < 2))  +- \*(1) ColumnarToRow  +- FileScan parquet default.tab2[id#16L,k#17L] Batched: true, DataFilters: [isnotnull(id#16L), (id#16L < 2)], Format: Parquet, Location: PrunedInMemoryFileIndex[file:/C:/develop/code/idea/Spark3\_Test/spark-warehouse/tab2/k=0, file:/C:..., PartitionFilters: [isnotnull(k#17L)], PushedFilters: [IsNotNull(id), LessThan(id,2)], ReadSchema: struct<id:bigint>  +---+---+---+---+  | id| k| id| k|  +---+---+---+---+  | 0| 0| 0| 0|  | 1| 1| 1| 1|  +---+---+---+---+ |



DAG 图如下：

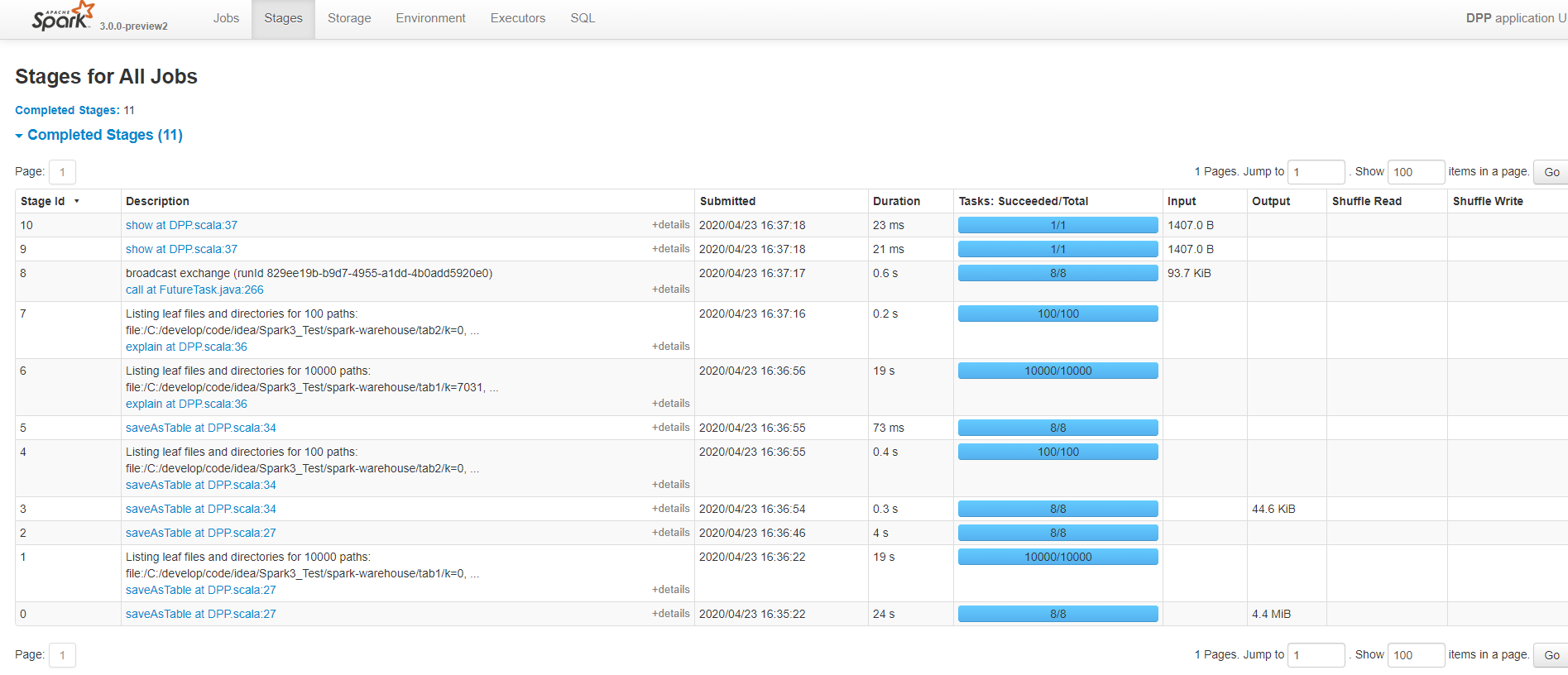


从上面的物理执行计划和 DAG 执行图可以看出，这个就是正常的算子下推的结果。

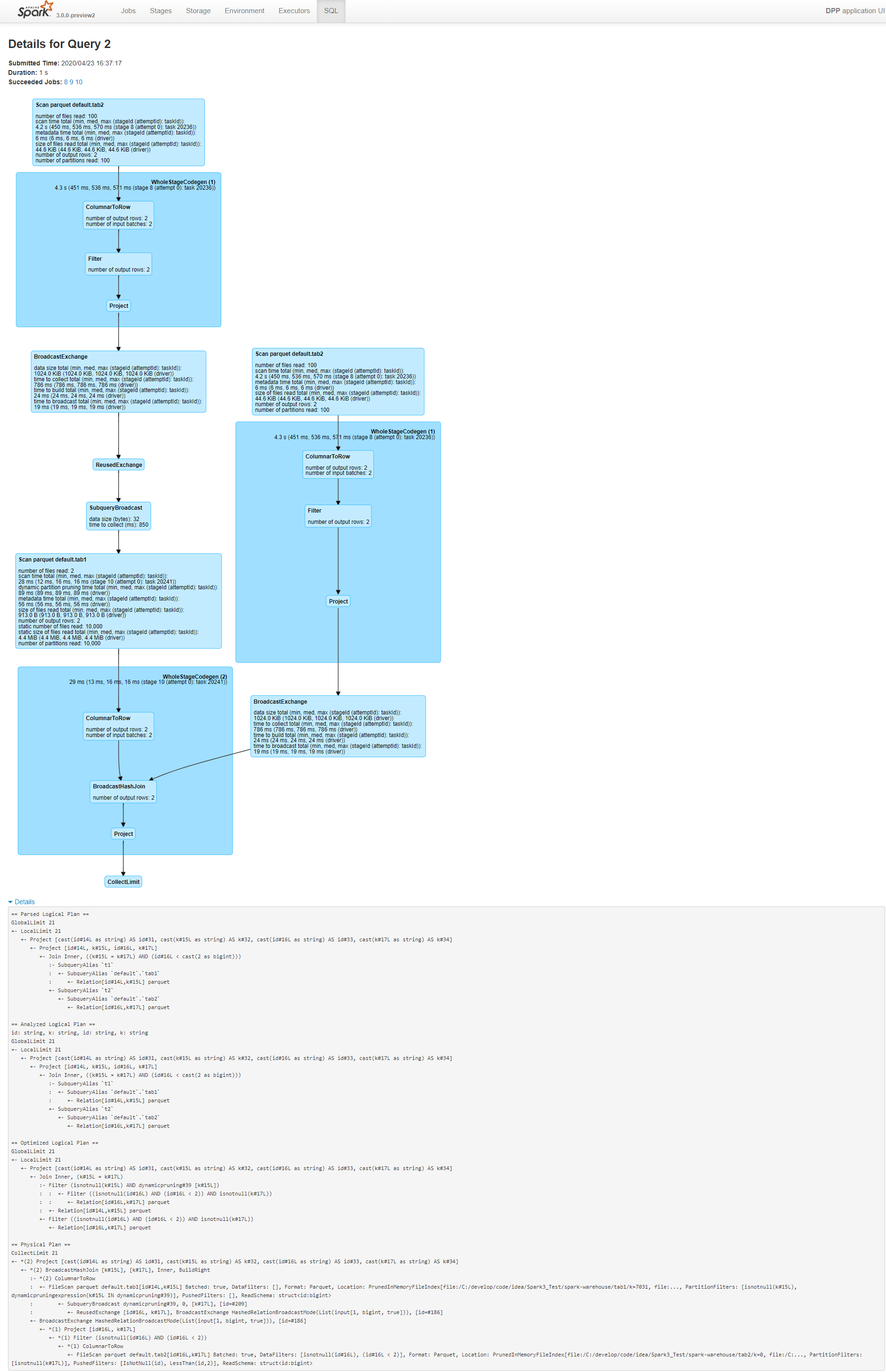
### 启动动态分区裁剪

如果我们使用 Apache [Spark 3.0](https://www.iteblog.com/archives/tag/spark-3-0/" \o "" \t "https://www.iteblog.com/archives/_blank) 并启用动态分区裁减功能，执行结果如下：

|  |
| --- |
| == Physical Plan ==  \*(2) BroadcastHashJoin [k#15L], [k#17L], Inner, BuildRight  :- \*(2) ColumnarToRow  : +- FileScan parquet default.tab1[id#14L,k#15L] Batched: true, DataFilters: [], Format: Parquet, Location: PrunedInMemoryFileIndex[file:/C:/develop/code/idea/Spark3\_Test/spark-warehouse/tab1/k=7031, file:..., PartitionFilters: [isnotnull(k#15L), dynamicpruningexpression(k#15L IN dynamicpruning#22)], PushedFilters: [], ReadSchema: struct<id:bigint>  : +- SubqueryBroadcast dynamicpruning#22, 0, [k#17L], [id=#111]  : +- ReusedExchange [id#16L, k#17L], BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, true])), [id=#90]  +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, true])), [id=#90]  +- \*(1) Project [id#16L, k#17L]  +- \*(1) Filter (isnotnull(id#16L) AND (id#16L < 2))  +- \*(1) ColumnarToRow  +- FileScan parquet default.tab2[id#16L,k#17L] Batched: true, DataFilters: [isnotnull(id#16L), (id#16L < 2)], Format: Parquet, Location: PrunedInMemoryFileIndex[file:/C:/develop/code/idea/Spark3\_Test/spark-warehouse/tab2/k=0, file:/C:..., PartitionFilters: [isnotnull(k#17L)], PushedFilters: [IsNotNull(id), LessThan(id,2)], ReadSchema: struct<id:bigint>  +---+---+---+---+  | id| k| id| k|  +---+---+---+---+  | 0| 0| 0| 0|  | 1| 1| 1| 1|  +---+---+---+---+ |



DAG 图如下：



可以看出，tab1 表的扫描相比上面那个多了一个分区过滤（PartitionFilters），这样在某些情况下性能就可以提升多倍。

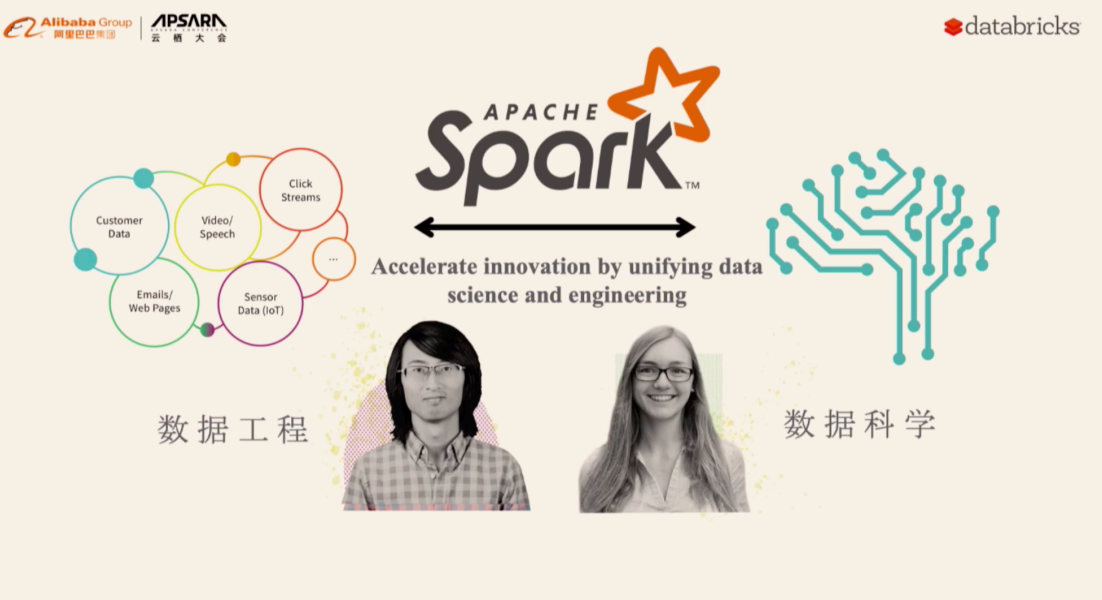
# Spark生态与未来

Spark设计之初的目标就是实现一套统一的系统，来解决与大数据相关的各式问题。就像是苹果公司在最初推出苹果手机的时候，其功能丰富性的目标引来了很多人的质疑，但是后来经过发展印证了其目标的商业价值，很多相机公司、游戏公司因为苹果手机的竞争而最终销声匿迹。我们相信Spark未来也可以在大数据领域实现能够统一解决各种问题的目标，但是目前Spark在流处理方面做得还不够好。

人工智能和大数据处理在近几年发展迅速，从最开始的数据仓库对已有数据进行响应式查询，来获得已有数据的统计信息，到现在基于已有数据进行的预测和分析，再到之后会用到人工智能和机器学习技术来辅助决策。Spark作为一个统一的大数据处理、分析引擎，将发挥很重要的作用。



Spark主要针对两个群体，即数据工程师和数据科学家。它们之间的关系应该是相辅相成的，如下图所示：

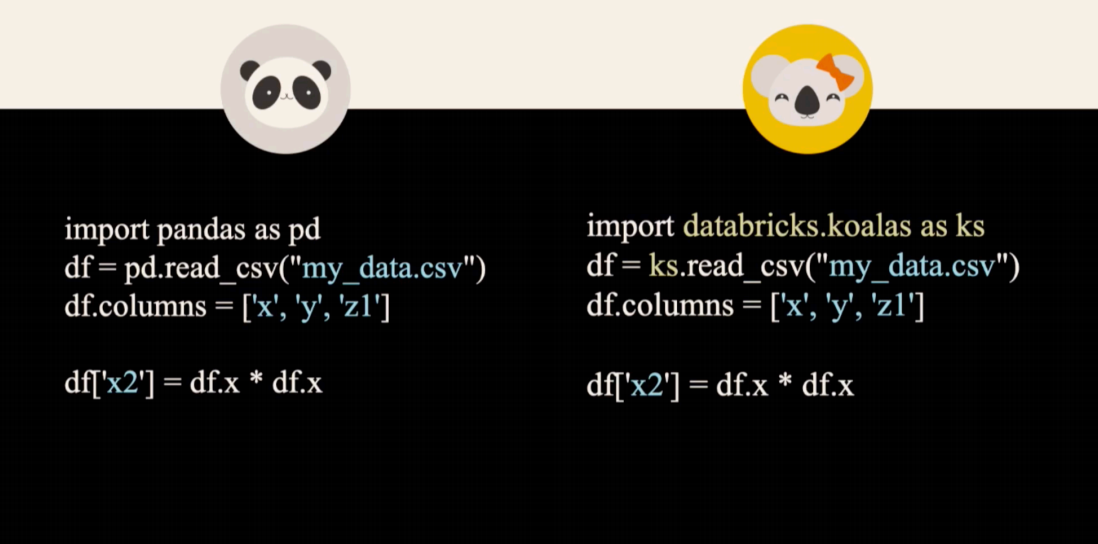


Spark通过提供一套统一的引擎，来打破数据工程和数据科学之间的障碍，使其更加容易地进行交流和合作。

## 数据科学

在数据科学方面Spark采用和当前比较主流的数据分析语言python进行衔接来完成，在python中使用pandas可以进行数据分析，但是适用范围只能在数据量比较小的时候，为了解决这个问题，Spark开源了koalas，其和pandas无缝兼容，为广大的数据科学家带来在大数据场景下能轻松的进行数据分析的福音。

下面是单独使用koalas 和 pandas的案例:

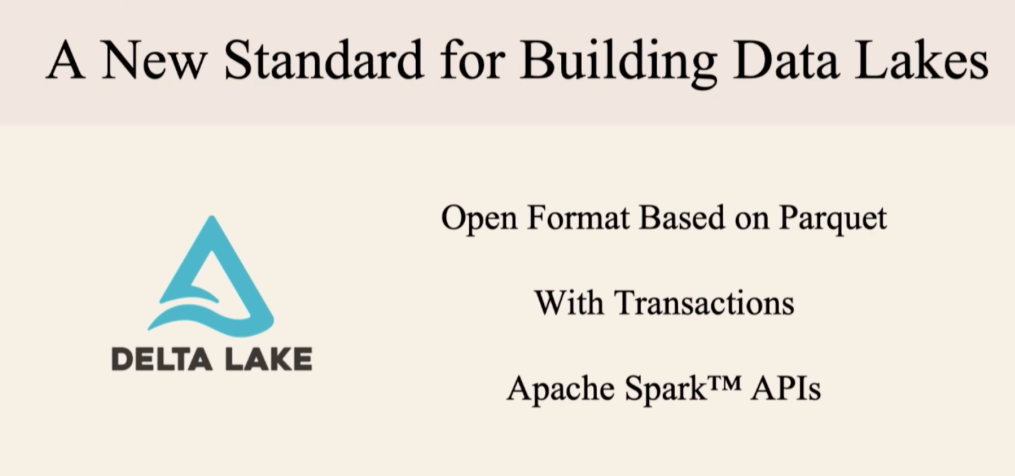


可以看到二者在使用语法方面完全没有差异，极大的方便了数据科学家的使用和操作。

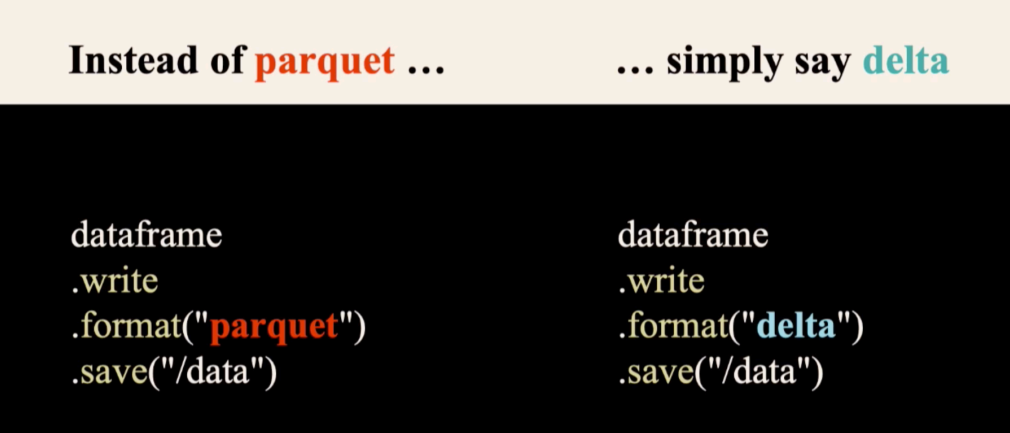
## 数据工程

在数据工程方面，Databricks开源了Delta Lake

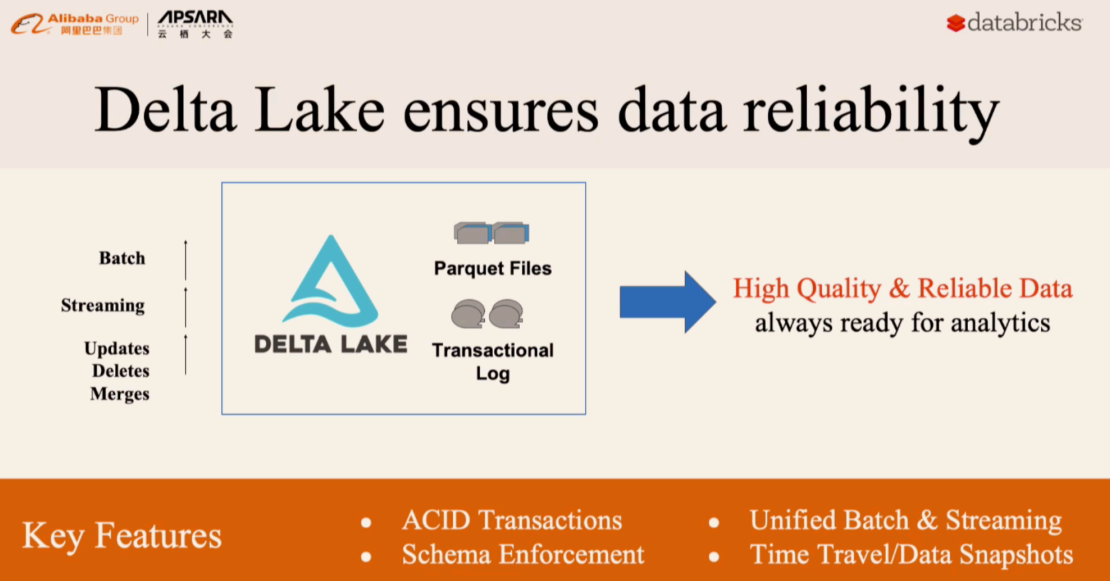
Delta是基于Spark上千用户在使用过程中的痛点问题而设计开发的一款产品。它是构建data lake的开放标准，基于已有的Parquet，支持transaction来保证数据的质量，同时完全兼容Spark的API。Delta于2019年4月宣布开源，令很多用户激动不已



下图中的例子展示了如何从Parquet转换到Delta。用户只需要在代码中用delta替换Parquet，仅此而已。

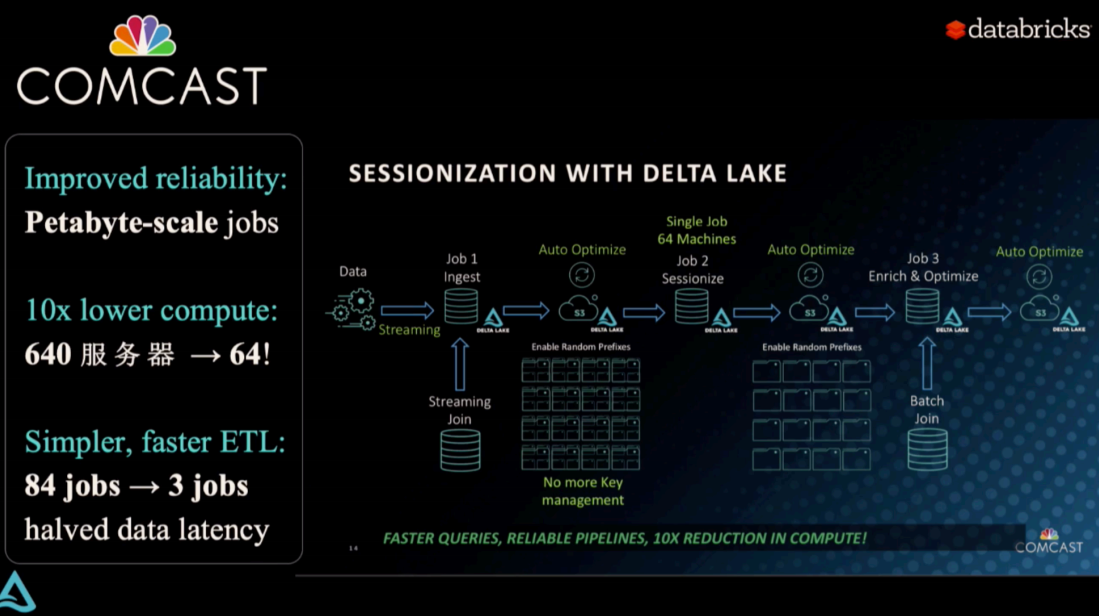


Delta主要包括四个核心的特性，其中最重要的是ACID transactions，它通过乐观并发控制来实现数据的同时读、写，从而保证数据的一致性，提高了数据的质量，让批处理和流处理可以使用一套统一的数据，支持update、delete和merge等典型的数据仓库操作。还有一点不得不提的是，Spark中元数据处理已经从小数据变成了大数据，当一个表有成百上千万的partition的时候，元数据获取和处理将会耗费大量的时间，有时甚至多大几小时，而使用Delta使用Spark做元数据处理，可以大大提高效率。

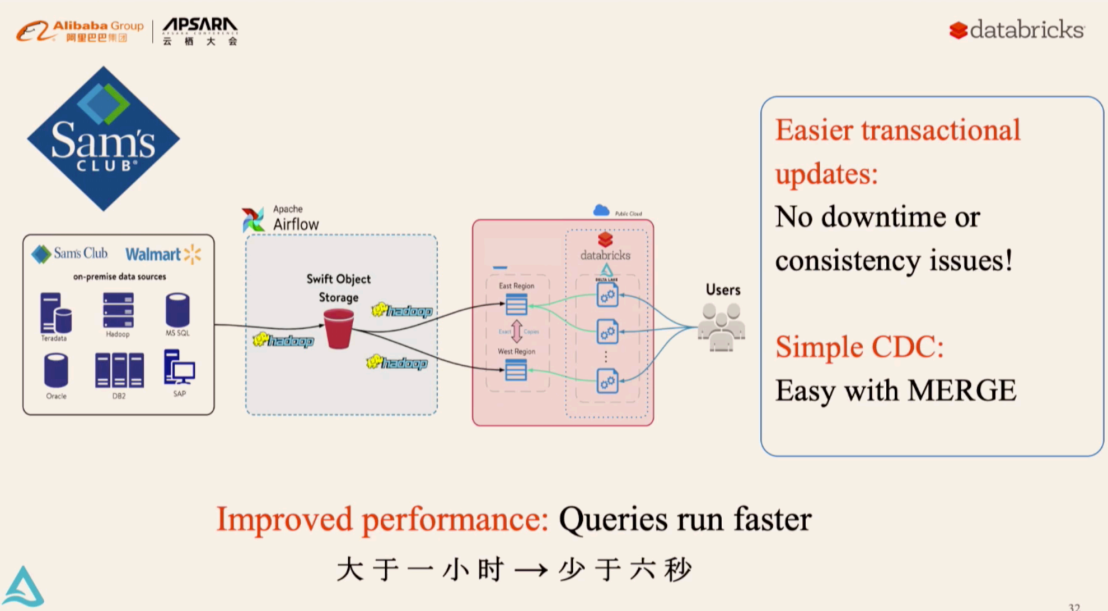


下面分享三个使用Delta的社区用户场景：

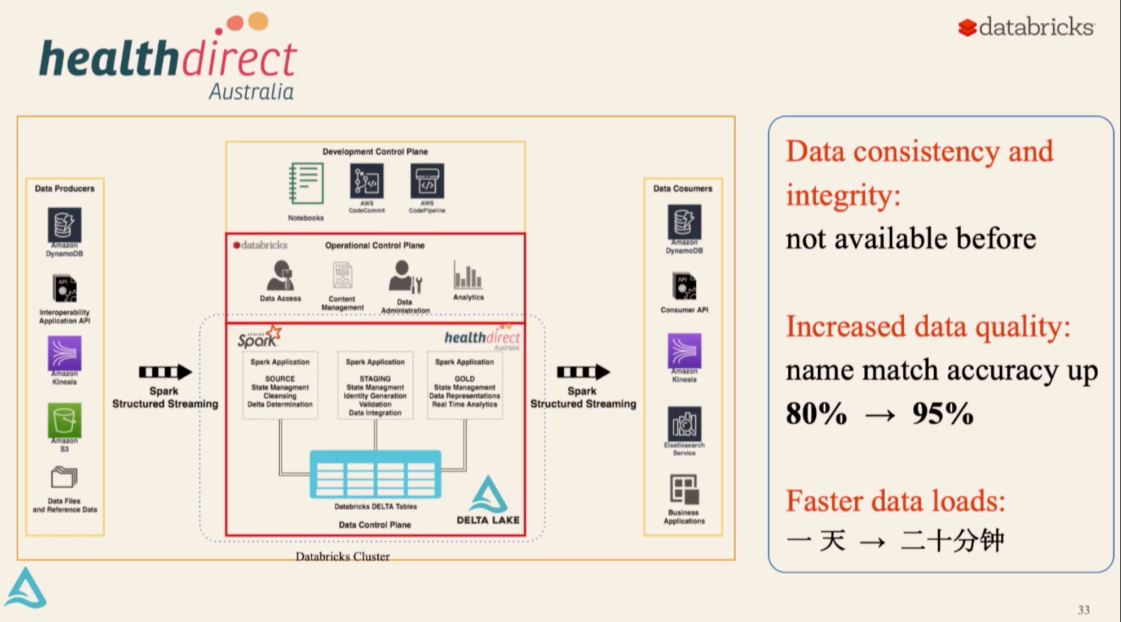
• Comcast：它是一个媒体内容提供商，它使用Spark来作为数据处理引擎已经很长时间了，因为其数据量级比较大，可以达到PB级别。在用Delta代替Parquet后，由原来一条pipeline需要的640个服务器降低到64个，原来需要84个job完成的数据处理降低到现在的3个，数据延迟加速了一倍。



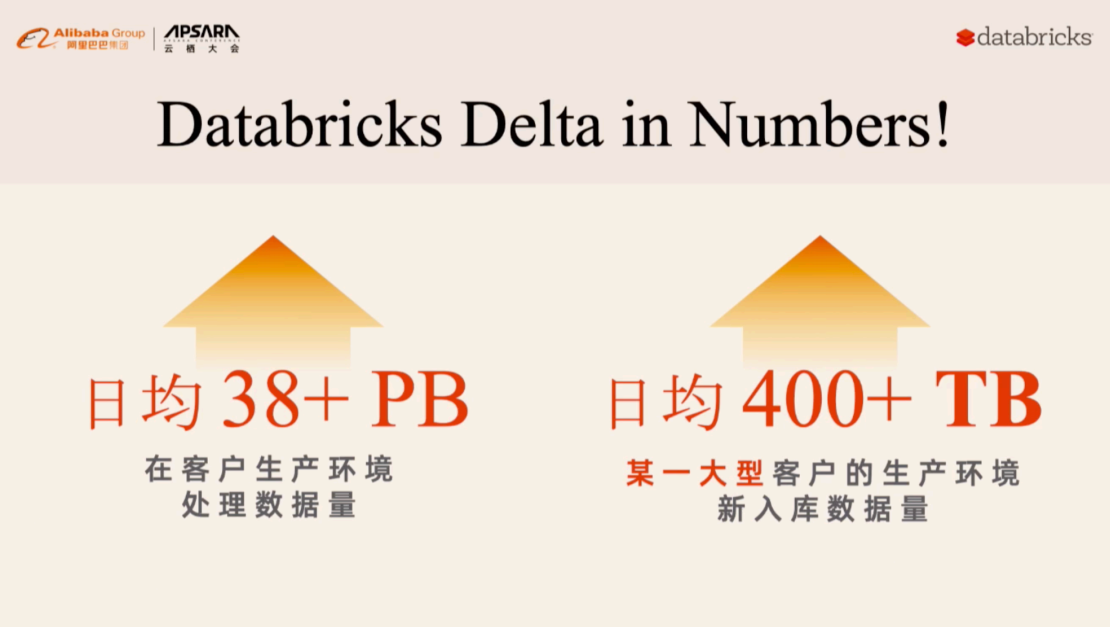
• Sam’s：它是沃尔玛的一个子品牌，也是Spark的重度用户。其原来没有transaction的update，因此经常会由于数据的质量问题导致pipeline停止。自从使用了Delta，transaction的问题得到了完美解决。此外，Delta提供jobs自动merge的API，帮助该用户merge部分的操作时间由原来的超过一个小时缩减到现在的小于6秒。



• Healthdirect Australia：该用户原来的数据一致性无法保证，因为Delta的使用，数据质量大幅度提升，名称匹配的准确性由原来的80%提升到了95%，数据加载速度由原来的一天缩减到二十分钟。



在Databricks使用Delta每天在客户生成环境中处理的数据量达到38PB以上，某一大型客户的生产环境每天新入库的数据量高达400TB以上。



总体来说，Apache Spark 3.0将会是Spark较大的一次版本升级，包含了许多重要特性，非常值得大家期待。