

# 让你的数据处理 更简单

Apache Spark  
中文实战攻略（上册）

Spark+AI Summit 2020 中文精华版峰会全新收录  
Apache Spark 3.0 性能优化与基础实战一书看遍





更多精彩内容扫码关注  
Spark 中文社区微信公众号



钉钉扫码加入  
Spark 中文社区钉钉群



阿里云开发者“藏经阁”  
海量电子书免费下载

# I 目录

<b>Apache Spark 的前世今生</b>	<b>4</b>
Apache Spark 3.0: 全新功能知多少	5
Apache Spark 3.0: 十年回顾, 展望未来	15
<b>Delta Lake 深度解析</b>	<b>30</b>
数据工程师眼中的 Delta lake	31
Data Lake 三剑客 —— Delta、Hudi、Iceberg对比分析	37
核桃编程 Delta Lake 实时数仓应用实践	43
“脏数据” 走开: Schema 约束和 Schema 演变	54
如何用事务日志优雅地解决并发读写	60
使用 Jupyter Notebook 运行 Delta Lake 入门教程	67
<b>Spark SQL 性能优化</b>	<b>78</b>
Apache Spark 3.0中的SQL性能改进概览	79
Structured Streaming生产化实践及调优	91
使用Spark Streaming SQL进行PV/UV统计	102
自适应查询执行AQE: 在运行时加速SparkSQL	107
浅析Hive/Spark SQL读文件时的输入任务划分	114

# Apache Spark 的前世今生

# Apache Spark 3.0: 全新功能知多少

简介：分享嘉宾 Apache Spark PMC李潇，就职于 Databricks，Spark 研发部主管，领导 Spark, Koalas, Databricks runtime, OEM 的研发团队，在直播中为大家深入讲解了 Apache Spark 3.0的新功能。

Spark3.0解决了超过3400个JIRAs，历时一年多，是整个社区集体智慧的成果。Spark SQL和 Spark Cores是其中的核心模块，其余模块如PySpark等模块均是建立在两者之上。Spark3.0新增了太多的功能，无法一一列举，下图是其中24个相对来说比较重要的新功能，下文将会围绕这些进行简单介绍。

## Performance



Adaptive Query Execution



Dynamic Partition Pruning



Query Compilation Speedup



Join Hints

## Built-in Data Sources



Parquet/ORC Nested Column Pruning



CSV Filter Pushdown



Parquet: Nested Column Filter Pushdown



New Binary Data Source

## Richer APIs



Accelerator-aware Scheduler



Built-in Functions



pandas UDF Enhancements



DELETE/UPDATE/MERGE in Catalyst

## SQL Compatibility



Overflow Checking



ANSI Store Assignment



Proleptic Gregorian Calendar



Reserved Keywords

## Extensibility and Ecosystem



Data Source V2 API + Catalog Support



Hadoop 3 Support



Hive 3.x Metastore Hive 2.3 Execution



Java 11 Support

## Monitoring and Debuggability



Structured Streaming UI



DDL/DML Enhancements



Observable Metrics



Event Log Rollover

## 一、Performance

与性能相关的新功能主要有：

- Adaptive Query Execution
- Dynamic Partition Pruning
- Query Compilation Speedup
- Join Hints

### （一）Adaptive Query Execution

Adaptive Query Execution（AQE）在之前的版本里已经有所实现，但是之前的框架存在一些缺

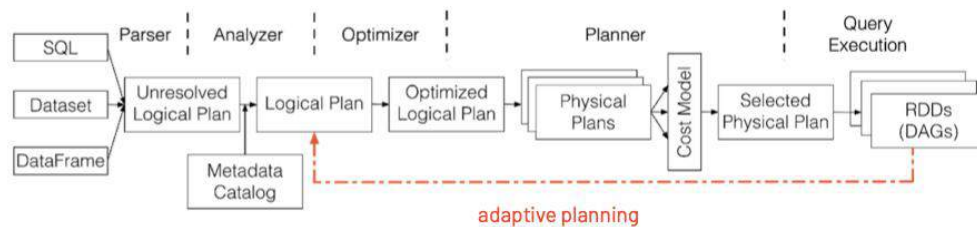
陷，导致使用不是很多，在Spark3.0中Databricks（Spark初创团队创建的大数据与AI智能公司）和Intel的工程师合作，解决了相关的问题。

在Spark1.0中所有的Catalyst Optimizer都是基于规则 (rule) 优化的。为了产生比较好的查询规则，优化器需要理解数据的特性，于是在Spark2.0中引入了基于代价的优化器（cost-based optimizer），也就是所谓的CBO。然而，CBO也无法解决很多问题，比如：

- 数据统计信息普遍缺失，统计信息的收集代价较高；
- 储存计算分离的架构使得收集到的统计信息可能不再准确；
- Spark部署在某一单一的硬件架构上，cost很难被估计；
- Spark的UDF（User-defined Function）简单易用，种类繁多，但是对于CBO来说是个黑盒子，无法估计其cost。

总而言之，由于种种限制，Spark的优化器无法产生最好的Plan。也正是因为上诉原因，运行期的自适应调整就变得相当重要，对于Spark更是如此，于是有了AQE，其基本方法也非常简单易懂。如下图所示，在执行完部分的查询规划后，Spark可以收集到结果的统计信息，然后利用这些信息再对查询规划重新进行优化。这个优化的过程不是一次性的，而是自适应的，也就是说随着查询规划的执行会不断的进行优化，而且尽可能地复用了现有优化器的已有优化规则。让整个查询优化变得更加灵活和自适应。

## Adaptive Query Execution [AQE]



Based on statistics of the finished plan nodes, re-optimize the execution plan of the remaining queries



Spark3.0中AQE包括三个主要的运行期自适应功能：

- 可以基于运行期的统计信息，将Sort Merge Join 转换为Broadcast Hash Join；
- 可以基于数据运行中间结果的统计信息，减少reducer数量，避免数据在shuffle期间的过量分区导致性能损失；
- 可以处理数据分布不均导致的skew join。

更多的信息大家可以通过搜索引擎查询了解。



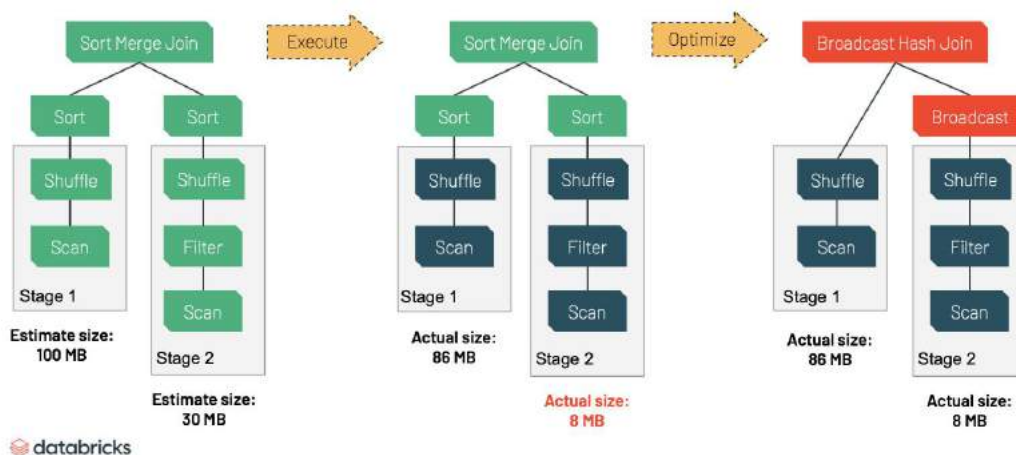
如果你是一个Spark的资深用户，可能你读了很多的调优宝典，其中第一条就是让你的Join变得更快，方法就是尽可能地使用Broadcast Hash Join。比如你可以增加

`spark.sql.autoBroadcastJoinThreshold` 阈值，或者使用 `broadcast HINT`。但是这基本上属于艺高人胆大。首先，这种方法很难调，一不小心就会Out of Memory，甚至性能变得更差，即使现在产生了一定效果，但是随着负载的变化可能调优会完全失败。

也许你会想：Spark为什么不解决这个问题呢？这里有很多挑战，比如：

- 统计信息的缺失，统计信息的不准确，那么就是默认依据文件大小来预估表的大小，但是文件往往是压缩的，尤其是列存储格式，比如parquet 和 ORC，而Spark是基于行处理，如果数据连续重复，file size可能和真实的行存储的真实大小，差别非常之大。这也是为何提高 `autoBroadcastJoinThreshold`，即使不是太大也可能导致out of memory；
- Filter复杂、UDFs的使用都会使Spark无法准确估计Join输入数据量的大小。当你的query plan异常大和复杂的时候，这点尤其明显。
- 其中，Spark3.0中基于运行期的统计信息，将Sort Merge Join 转换为Broadcast Hash Join 的过程如下图所示。

## Convert Sort Merge Join to Broadcast Hash Join

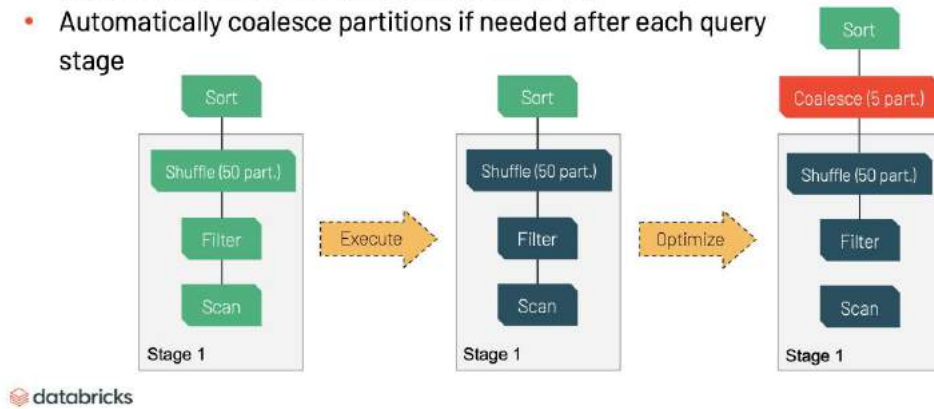


也许你还会看到调优宝典告诉你调整shuffle产生的partitions的数量。而当前默认数量是200，但是这个200为什么就不得而知了。然而，这个值设置为多少都不是最优的。其实在不同shuffle，数据的输入大小和分布绝大多数都是不一样。那么简单地用一个配置，让所有的shuffle来遵循，显然是不好的。要设得太小，每个partition的大小就会太大，那么GC的压力就会很大，aggregation和sort会更有可能的去spill数据到磁盘。但是，要是设太大，partition的大小就会太小，partition的数量会大。这个会导致不必要的IO，也让task调度器的压力剧增。那么调度器会导致所有task都变慢。这一系列问题在query plan复杂的时候变得尤为突出，还可能会影响到其他性能，最后耗时耗力却调优失败。

对于这个问题的解决，AQE就有优势了。如下图所示，AQE可以在运行期动态的调整partition来达到性能最优。

## Dynamically Coalesce Shuffle Partitions

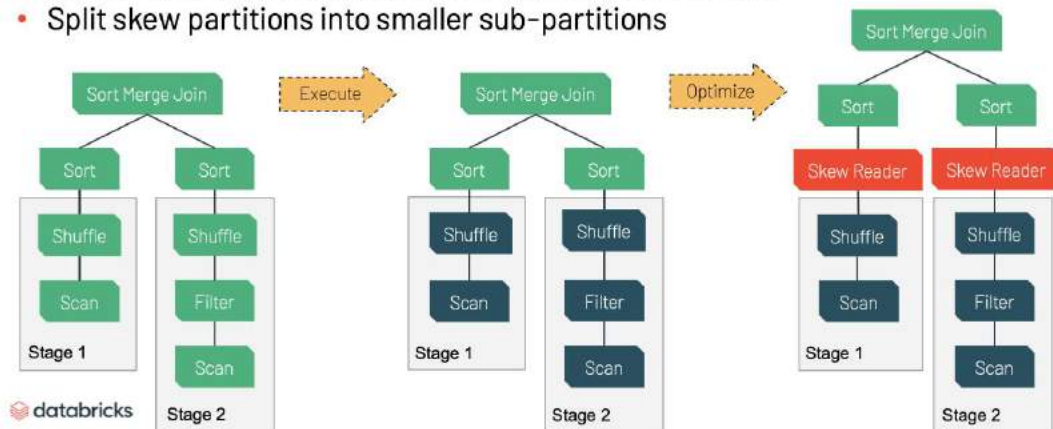
- Set the initial partition number high to accommodate the largest data size of the entire query execution
- Automatically coalesce partitions if needed after each query stage



此外，数据分布不均Spark调优的一个疑难杂症，它的表现有多种，比如若干task停滞不前，像是出现了bugs，又比如大量的disk spilling会导致很多节点都无事可做。此外，你也许会看到out of memory这种异常。其解决方法也很多，比如找到skew values然后重写query，或者在join的情况下增加skew keys来消除数据分布不均，但是无论哪种方法，都非常浪费时间，且后期难以维护。AQE解决问题的方式如下，其通过shuffle落地后的中间数据结果判断哪些partition是skew的，如果partition过大，就将其分成若干较小的partition，通过分而治之，总体性能大幅提升。

## Dynamically Optimize Skew Joins

- Detect skew from partition sizes using runtime statistics
- Split skew partitions into smaller sub-partitions



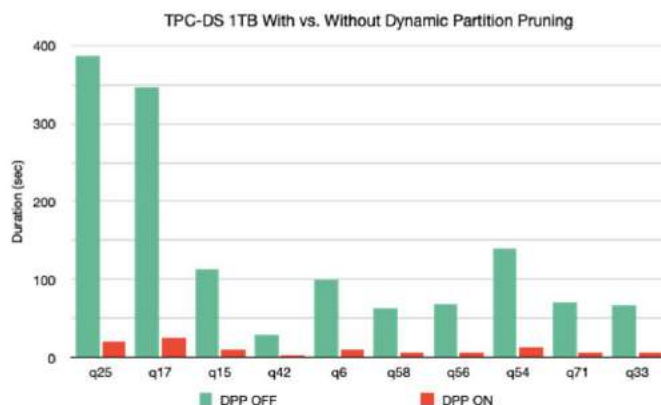


AQE的发布可以说是一个时代的开始，未来将会更进一步发展，引入更多自适应规则，让Spark可以随着数据分布和特性的变化自动改变Query plan，让更多的query编译静态优化变成运行时的动态优化。

## （二）Dynamic Partition Pruning

Dynamic Partition Pruning也是一个运行时的动态优化方法，简单来说就是我们可以通过Query的某些分支的中间结果来避免不必要的partition读取，这种方法是无法通过编译期推测出来的，只能在运行时根据结果来判断，这种方法对数据仓库的star-schema效果非常明显，在TPC-DS获得了非常明显的加速，可以加速2-18倍。

### Dynamic Partition Pruning

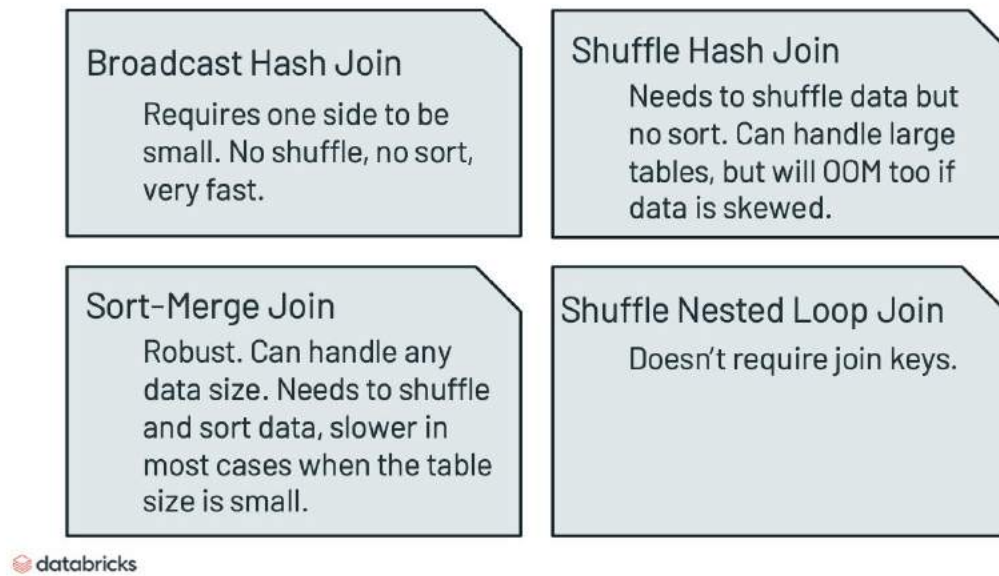


60 / 102 TPC-DS queries: a speedup between 2x and 18x



## （三）Join Hints

Join Hints是一个非常普遍的数据库的优化策略，在3.0之前已经有了Broadcast hash join，3.0之后的版本加了Sort-merge join、Shuffle hash join和 Shuffle nested loop join，但是要注意谨慎使用，因为数据的特性不同，很难保证一直有效，即使有效，也不代表一直有效，随着时间的变化，你的数据变了，可能会让你的query 变慢，变得不稳定。总体来说上面的四种Join的适用条件和特点如下所示，总而言之，使用Join Hints要谨慎。



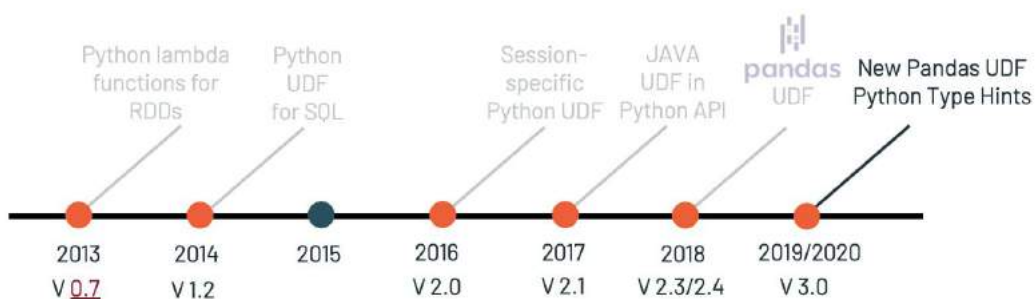
## 二、Richer APIs

Spark3.0简化了开发，不但增加了更多的新功能，也改善了众多现有的功能，让更多的用法成为可能，主要有：

- Accelerator-aware Scheduler
- Built-in Functions
- pandas UDF enhancements
- DELETE/UPDATE/MERGE in Catalyst

### （一）pandas UDF enhancements

pandas UDF应该说是PySpark用户中最喜爱的特性之一，对于其功能和性能的提升应该都是喜闻乐见的，其发展历程如下图所示。



Blog post: <https://databricks.com/blog/2020/05/20/new-pandas-udfs-and-python-type-hints-in-the-upcoming-release-of-apache-spark-3-0.html>

最新的pandas UDF和之前的不同之处在于引入了Python Type Hints，现在用户可以使用pandas中的数据类型比如pandas.Series等来表示pandas UDF的种类，不再需要记住原来的UDF类型，只需要指定正确的输入和输出类型即可。此外，pandas UDF可以分为pandas UDF和pandas API。

## （二）Accelerator-aware Scheduler

Accelerator-aware Scheduler是加速器的调度支持，狭义上也就是指GPU调度支持。加速器经常用来对特定负载做加速，目前，用户还是需要指定什么应用需要加速器资源，但是在将来我们会支持job或者stage级别的调度。Spark3.0中我们已经支持大多调度器，此外，我们还可以通过Web UI来监控GPU的使用，欢迎大家使用，更多详细资料大家可以到社区学习。

## （三）Built-in Functions

### 32 New Built-in Functions



databricks

为了让Spark3.0更方便实用，Spark社区按照其他的主流，比如数据库厂商等，内嵌了如上图所示的32个常用函数，这样用户就无须自己写UDF，并且速度更快。比如针对map类型，Spark3.0新增加了map\_keys和map\_values，更加地方便易用。其他新增加的更多内嵌函数大家可以到社区具体了解。

## 三、Monitoring and Debuggability

Spark3.0也增加了一些对监控和调优的改进，主要有：

- Structured Streaming UI
- DDL/DML Enhancements
- Observable Metrics
- Event Log Rollover

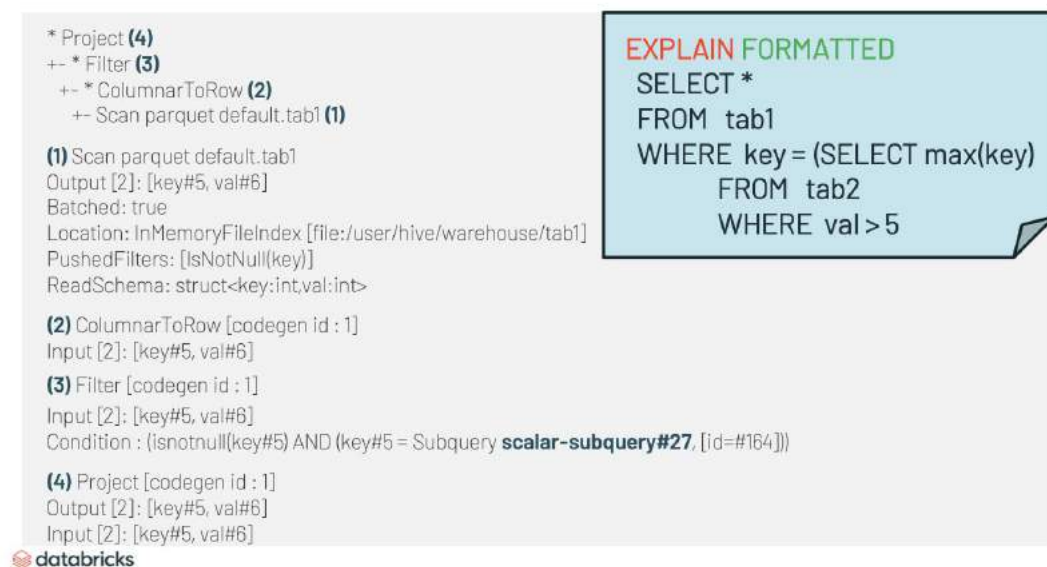
## （一）Structured Streaming UI

Structured Streaming是在Spark2.0中发布的，在Spark3.0中加入了UI的配置。新的UI主要包括了两种统计信息：已完成的Streaming查询聚合信息和未完成的Streaming查询的当前信息，包括Input Rate、Process Rate、Batch Duration和Operate Duration。

## （二）DDL/DML Enhancements

我们还增加了各种DDL/DML命令，比如EXPLAIN和。

EXPLAIN是性能调优的必备工具，读取EXPLAIN是每个用户的基本功，但是随着系统的运行，EXPLAIN的信息越来越多，而且信息多元、多样，在新的版本中我们引入了新的FORMATTED模式，如下所示，在开头处有一个非常精简的树状图，且之后的每个部分都有很详细的解释，更容易加更多的注意，这就从水平扩展变成了垂直扩展，更加的直观。



```
* Project (4)
+- * Filter (3)
   +- * ColumnarToRow (2)
      +- Scan parquet default.tab1 (1)


(1) Scan parquet default.tab1
Output [2]: [key#5, val#6]
Batched: true
Location: InMemoryFileIndex [file:/user/hive/warehouse/tab1]
PushedFilters: [IsNotNull(key)]
ReadSchema: struct<key:int,val:int>

(2) ColumnarToRow [codegen id : 1]
Input [2]: [key#5, val#6]

(3) Filter [codegen id : 1]
Input [2]: [key#5, val#6]
Condition : (isnotnull(key#5) AND (key#5 = Subquery scalar-subquery#27, [id=#164]))

(4) Project [codegen id : 1]
Output [2]: [key#5, val#6]
Input [2]: [key#5, val#6]
```

**EXPLAIN FORMATTED**  
SELECT \*  
FROM tab1  
WHERE key = (SELECT max(key)  
FROM tab2  
WHERE val > 5)



## （三）Observable Metrics

我们还引入了Observable Metrics用以观测数据的质量。要知道数据质量对于很多Spark应用都是相当重要的，通常定义数据质量的Metrics还是非常容易的，比如用一些聚合参数，但是算出这个Metrics的值就非常麻烦，尤其对于流计算来说。

## 四、SQL Compatibility

SQL兼容性也是Spark必不可提的话题，良好的兼容性更方便用户迁移到Spark平台，在Spark3.0中新增的主要功能有：

- ANSI Store Assignment
- Overflow Checking

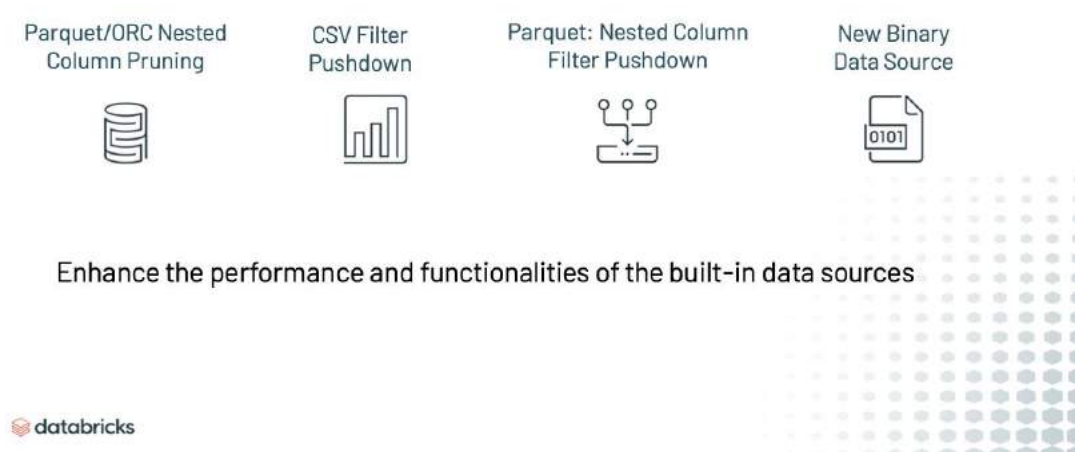
- Reserved Keywords in Parser
- Proleptic Gregorian Calendar

也就是说，这个版本中我们让insert遵守了ANSI Store Assignment，并且增加了运行时的overflow的检查，还提供了一个模式让SQL Parser来准确地遵守ANSI标准的保留字，还切换了Calendar，这样更加符合ANSI的SQL标准。比如说我们想要插入两列数据，类型是int和string，如果将int插入到了string中，还是可以的，不会发生数据精度的损失和数据丢失；但是如果我们尝试将string类型插入到int类型中，就有可能发生数据损失甚至丢失。ANSI Store Assignment+Overflow Checking在输入不合法的时候就会在运行时抛出异常，需要注意的是这个设置默认是关闭的，可以根据个人需要打开。

## 五、Built-in Data Sources

在这个版本中我们提升了预装的数据源，比如Parquet table，我们可以对Nested Column做Column Pruning和Filter Pushdown，此外还支持了对CSV的Filter Pushdown，还引入了Binary Data Source来处理类似于二进制的图片文件。

### Built-in Data Sources



## 六、Extensibility and Ecosystem

Spark3.0继续加强了对生态圈的建设：

- 对Data Source V2 API的持续改善和catalog支持；
- 支持Java 11；
- 支持Hadoop 3；
- 支持Hive 3。

### （一）Data Source V2 API+Catalog Support

Spark3.0加上了对Catalog的支持来扩展Data Source API。Catalog plugin API可以让用户注册自己的catalog来实现对元数据的处理，这样可以让Spark用户更简单方便的使用数据源的表。对于没有实现Catalog plugin的数据源，用户需要先注册每个外部数据源的表才能访问，但是实现了



Catalog plugin API之后我们只需要注册Catalog，然后就可以直接远程访问和操作catalog的表。对于数据源的开发者来说，什么时候支持Data Source V2 API呢？下面是几点建议：

不过这里需要注意，Data Source V2还不是很稳定，开发者可能在未来还需要调整相关API的实现。大数据的发展相当迅速，Spark3.0为了能更方便的部署，我们升级了对各个组件和环境版本的支持，但是要注意以下事项。

关于生态圈，这里要提一下Koalas，它是一个纯的Python库，用Spark实现了绝大部分的pandas API，让pandas用户除了可以处理小数据，也可以处理大数据。Koalas对于pandas用户来说可以将pandas的代码扩展到大数据处理，使得学习PySpark变得更简单；对于现有的PySpark用户来说，多了更多的选择，可以用pandas API来解决生产力问题。过去一年多，Koalas的下载量是惊人的，在pip的下载量单日已经超过了37000，而且还在不断增长，5月的下载量也达到了85万。Koalas的代码其实不多，主要是API的实现，执行还是由Spark来做，所以Spark性能的提升对于Koalas用户来说是直接受益的。Koalas的发布周期相当频密，目前已经有33个发布，欢迎大家下载使用。

## Koalas



- Announced April 24, 2019
- Pure Python library
- Aims at providing the pandas API on top of Spark.
- Seamless transition between small and large data



如何读和理解Spark UI对大多数新用户来说是一个很大的挑战，尤其对SQL用户来说，在Spark3.0中我们增加了自己的UI文档<https://spark.apache.org/docs/latest/web-ui.html>并且增加了SQL Reference，<https://spark.apache.org/docs/latest/sql-ref.html>等，更详细的文档使得用户上手Spark的时候更加容易，欢迎大家去试一试Spark3.0，感受Spark的强大。

# Apache Spark 3.0：十年回顾，展望未来

简介：今年是Spark发布的第十年，回顾Spark如何一步步发展到今天，其发展过程所积累的经验，以及这些经验对Spark未来发展的启发，对Spark大有裨益。在7月4日的Spark+AI SUMMIT 2020中文精华版线上峰会上，Databricks Spark研发部主管李潇带来了《Apache Spark 3.0简介：回顾过去的十年，并展望未来》的全面解析，为大家介绍了Spark的起源、发展过程及最新进展，同时展望了Spark的未来。

演讲嘉宾简介：李潇，Databricks Spark研发部主管，Apache Spark committer,PMC member。

本次分享主要围绕以下四个方面：

5.Spark的起源

6.Spark的今天

7.Spark的最新进展

8.Spark的未来

## 一、Spark的起源

Spark的创始人Matei于2007年开始在Berkeley读博，他对数据中心级别的分布式计算非常感兴趣。



## This is a Special Year for Apache Spark

Spark 3.0, and 10 years since the first release of Spark!

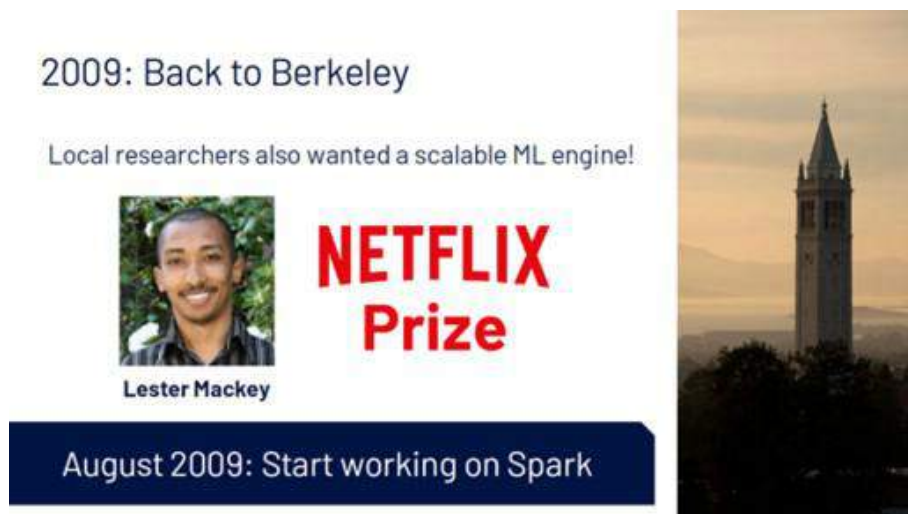


彼时，一些互联网公司已经开始用几千台机器计算并存储数据。这样，Matei开始和Yahoo以及Facebook的团队合作，来解决工业界中的大数据问题。在这一过程中，他意识到了这些大数据技术的巨大潜能，不仅能应用于互联网公司，还能应用于科学数据集和工业数据集等场景。然而，这些技术还没有在这些场景里普及起来，所有的处理流水线都还依赖于专业的工程师实现。同时，大多数技术都仅局限于批处理，还缺少对交互式查询的支持。最后，这些技术还都不支持机器学习。



另外，Matei在Berkeley继续研究时发现，Berkeley的研究团体同样需要可扩展的数据处理器，特别是机器学习研究。

于是，Matei开始抓住这个契机和Lester合作，后者参加了Netflix的一个机器学习大赛。Matei根据具体的应用场景设计了一个编程模型，使得像Lester这样的应用者可以使用这个编程模型开发大数据应用软件。基于此，2009年Matei着手开发Spark，并在最初就收到了许多好评，Berkeley的许多大数据研究者们都使用Spark进行大数据应用开发和研究。



2010年，Matei将Spark开源。Spark的最初版本仅仅关注Map Reduce风格的计算，其网页如下图所示。相对于传统的Map Reduce, Spark具有更干净的API以及更好的性能。令人兴奋的是，Spark在开源两个月后就收到了社区的代码，这说明确实有开源社区成员使用Spark并且利用Spark做出了一些有趣的项目。



在接下来的两年，Matei投入了许多精力来拜访早期Spark用户，组织Meetup。在和一些早期用户接触后，Matei被这些早期案例震惊了：一些用户在用Spark做一些他从未想过的事情。比如，一些用户使用Spark做后端处理，以开发交互式应用。比如，一些神经科学家使用Spark实时监控动物大脑产生的信号，进行神经科学实验。还有一些创业公司使用Spark分析用户使用的社交媒体数据，利用Spark更新内存数据以进行流计算，而这发生在Spark提供流计算模块之前。另外，来自Yahoo的数据仓库团队使用Spark运行SQL query，运行R实现的数据科学业务，他们还将Spark engine共享给团队内部的许多用户，用户体量非常大。这些都坚定了Matei的信心，确定了Spark大有可为。





从2012年到2015年，基于Spark的不同使用场景和需求，Spark团队开始扩展Spark以确保任何数据处理场景都可以使用Spark分担计算工作，用Spark解决数据处理问题。他们主要做了三方面努力：

第一，添加了Python, R以及SQL等编程语言入口，使得用户可以使用其熟悉的语言来使用Spark。

第二，添加了许多模块以完成不同数据处理任务，包括图处理、流处理。

第三，提供了更高层级的API以消除用户手动设置API的困扰，比如大家熟知的DataFrames API。

DataFrames API是Spark SQL中最受欢迎的API，和SQL Language一样。这是因为DataFrames API会被SQL优化集优化，同时又与编程语言融合在一起，因此比SQL语言更加简单好用。

## 2012-15: Expand Access to Spark

**Languages:** Python, R and SQL

**Libraries:** ML, Graphs and Streaming

**New high-level API:** DataFrames,  
built on the Spark SQL engine

SPARK+AI SUMMIT



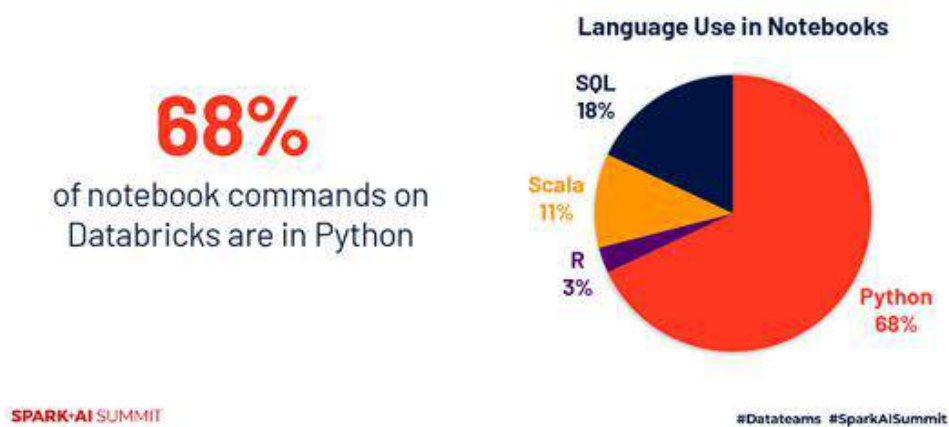
这些曾经的努力深深地影响着现在的Apache Spark。



## 二、Spark的今天

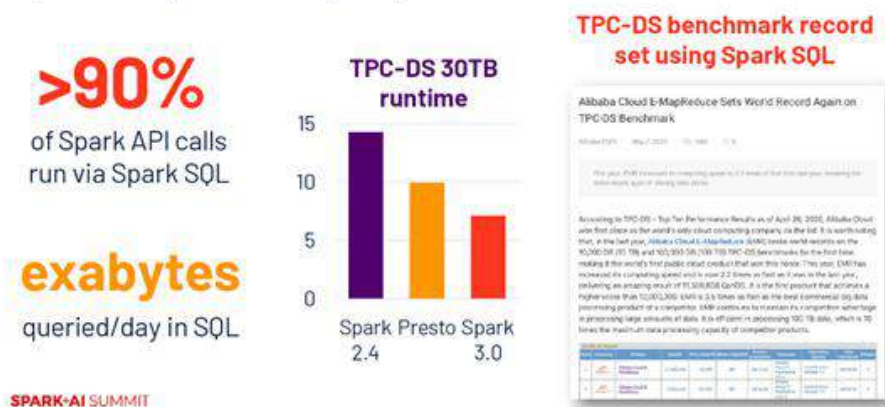
在Python方面，68%的Databricks的交互式notebook 命令是用Python写的，是Scala的6倍，同时远远高于SQL这种广泛使用的数据库语言。这意味着，更多的程序员可以使用Spark进行数据分析和处理，而不仅仅局限于使用SQL的程序员。

### Apache Spark Today: Python

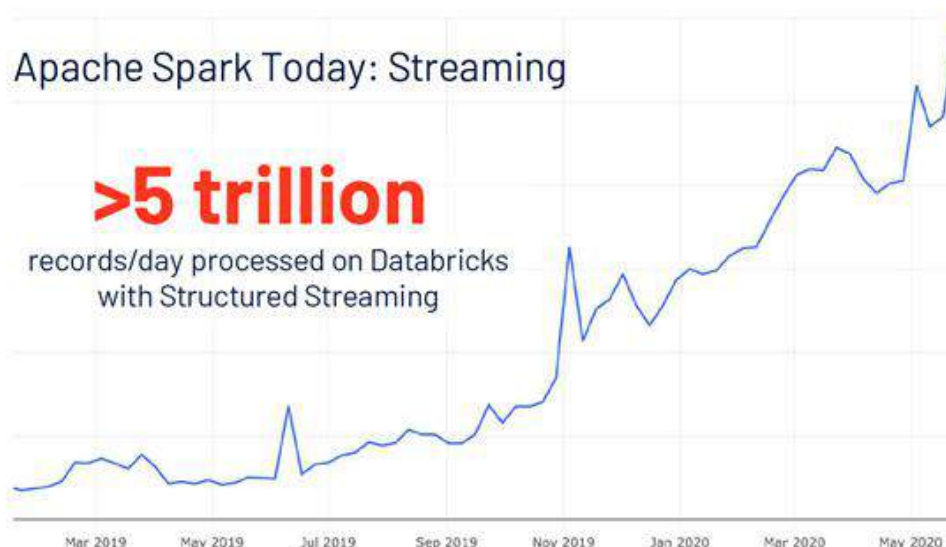


在SQL方面，大约有90%的Spark API调用实际上跑在Spark SQL这个模块上。无论开发人员使用Python, Scala, Java或者R调用Spark，这些开发人员都实际受益于SQL engine的优化。在Databricks上，每天由Spark SQL处理的数据量都达到了exabytes量级。因此，整个社区在Spark SQL的投入相当大，这使得Spark SQL engine成为性能最好的开源SQL engine。在TPC-DS benchmark 项目上，Spark 3.0 中 Spark SQL engine的性能比Spark 2.0整整快了两倍，比Presto快了1.5倍。今年年初，阿里巴巴的E-MapReduce团队使用Spark SQL engine打破了TPC-DS benchmark项目的最高性能记录。总的来说，Spark SQL engine是一个非常强大且高效的SQL engine。

### Apache Spark Today: SQL



在流处理方面，每天使用Databricks做流处理的数据超过5兆。Structured Streaming让流处理变得非常简单，能够轻松地将DataFrames和SQL计算变成流计算。近年来，使用Databricks做流计算的数据量每年翻4倍，增长非常迅速。



基于Spark的这些变化可以总结得到以下两个经验：

第一，易用性，即如何使用户更简单地使用和操作Spark 以及如何帮助用户快速定位错误，这对于一个好的项目非常重要；

第二，Spark API的设计需要关注这些设计能否支持用户实现软件开发的最佳实践，比如通过组合不同库函数实现不同应用，同时支持可测试和模块化。API的设计还需要融入标准的编程环境，比如Python, Java, Scala, R等， 以使其能够被方便地嵌入不同应用中。确保API的设计支持软件开发最佳实践，才能够使用户更安全和方便地使用它们。

以上是Spark已经做到的，随着时间的不断推进，Spark还将继续在这些方面不断完善。

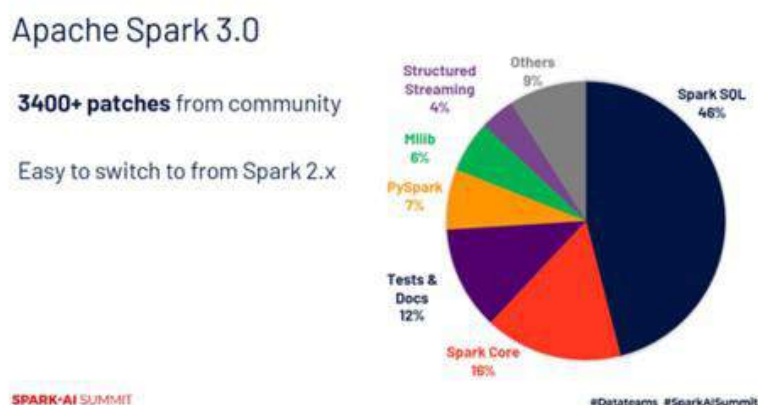
## Major Lessons

- 1 Focus on ease-of-use in both exploration *and* production
- 2 APIs to enable software best practices: composition, testability, modularity

### 三、Spark的最新进展

Spark 3.0是Spark有史以来最大的Release，共包含3400多个patch。下面这张图显示了这些patch所属的模块，几乎一半的patch都属于Spark SQL。Spark SQL 的优化不仅服务于SQL language，还服务于机器学习、流计算和Dataframes等计算任务，这使得社区对Spark SQL的投入非常大。此外，Spark团队还付出了大量努力使Spark 2.0的用户方便地升级到3.0。

以下主要从SQL和Python两个方面讨论Spark的最新进展。



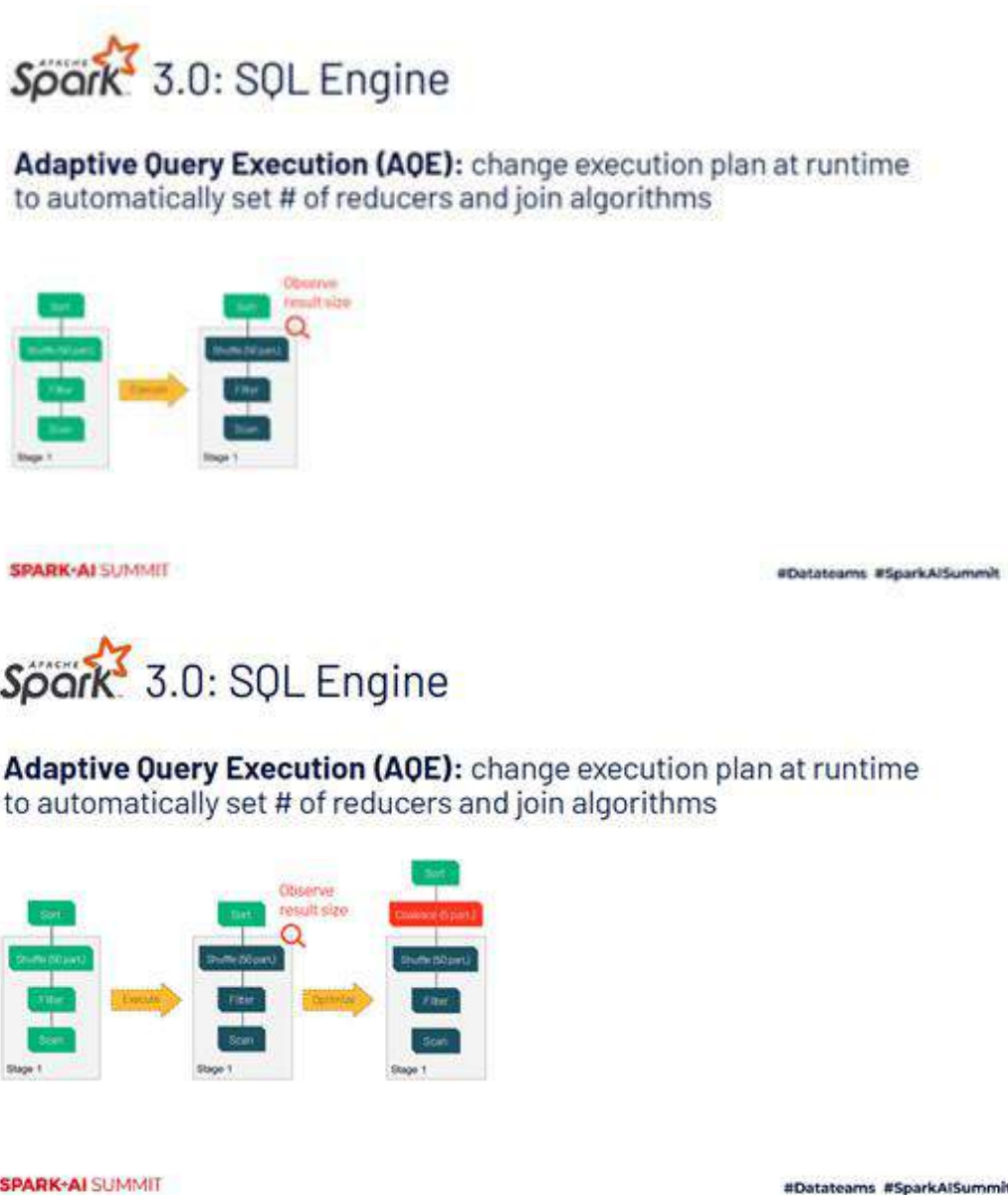
#### SQL方面

近几年，SQL engine方面主要的改进就是Adaptive Query Execution (AQE)。AQE能够在运行时根据计算任务的特性更新计算计划，也就是execution plan，比如自动调整reducer数量，自动改变join算法，自适应地处理数据倾斜问题。AQE的这种能力让Spark变得更加简单易用，使得Spark使用者们不需要再繁冗地进行各种数据调优。

AQE可以自动调整reducer数量。过去，Dataframes 上60%的集群都需要用户手动更改reducer数量，这使得用户不胜其扰，而AQE可以帮助解决这个问题。



AQE可以自动减小partition数量。在做聚合时，AQE还可以通过调整和合并小的partition，自适应地减小partition的数量，以减少资源浪费。



AQE还可以优化join操作。即使有高质量的数据统计信息，用户仍然很难获悉已经进行join操作的记录数。而AQE可以在join操作的初始阶段获悉数据的输入特性，并基于此选择适合的join算法从而最大化地优化性能。

AQE还能够解决数据倾斜问题，通过调整不同key的数据来避免数据倾斜，从而提高性能。比如，在TPC-DS的查询问题上，AQE能够调整不同key以达到8倍的性能加速。这使得用户能够使用Spark处理更多数据，而不需要手动收集数据统计信息来优化性能。



AQE仅仅是Spark 3.0在性能方面的一种改进，提升Spark性能的例子还包括Dynamic partition pruning, Query compile speedups, 以及Optimizer hints等。正如前文所述，Spark 3.0相比Spark 2.0的性能提速达到2倍，并且这种改进在真实场景下可能更明显。



除了性能，Spark 3.0在SQL的兼容性方面也有了很大的提升。比如，ANSI SQL方言做了标准化的SQL支持,使得其它SQL系统的负载和业务能够很容易转移到Spark SQL上。





## Python方面

Spark 3.0在Python方面的改善包括易用性和性能。

对于易用性，Spark 3.0使用户能够更方便地定义Pandas UDFs。用户可以通过Python type hints 指定其期待的数据格式和函数类型。在Spark 3.0中，用户可以仅仅指明数据的type，而不是写许多冗余的模板式代码，类似于Spark 2.0。

**Spark 3.0: Python Usability**

**Python type hints for Pandas UDFs**

```
def pandas_plus_one(v: pd.Series) -> pd.Series:
    return v + 1

def pandas_plus_one(itr: Iterator[pd.Series]) -> Iterator[pd.Series]:
    return map(lambda v: v + 1, itr)

def pandas_plus_one(pdf: pd.DataFrame) -> pd.DataFrame:
    return pdf + 1

spark.range(10).select(pandas_plus_one("id"))
```

**Old API**

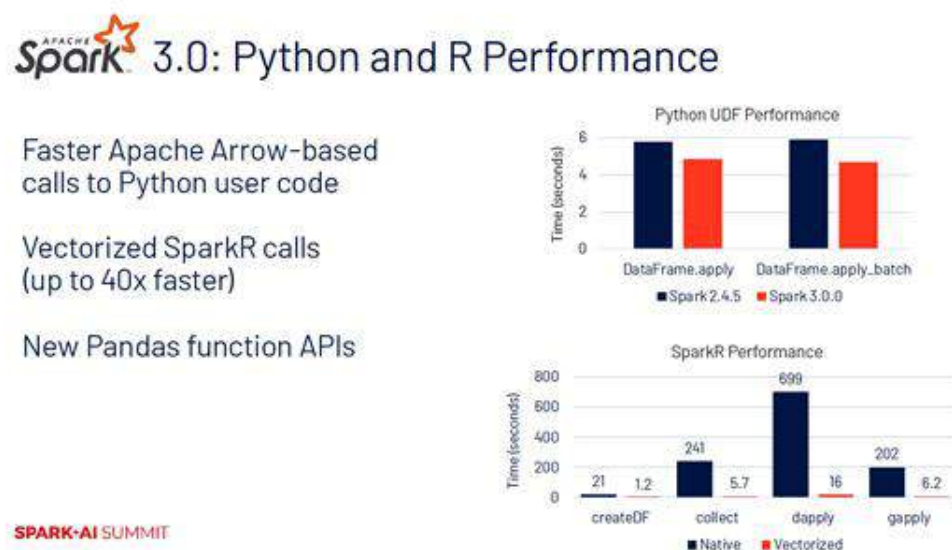
```
@pandas_udf('long', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # v is a pandas Series
    return v + 1

@pandas_udf('long', PandasUDFType.SCALAR_ITER)
def pandas_plus_one(itr):
    # Iterator is an iterator of pandas Series.
    return map(lambda v: v + 1, itr)

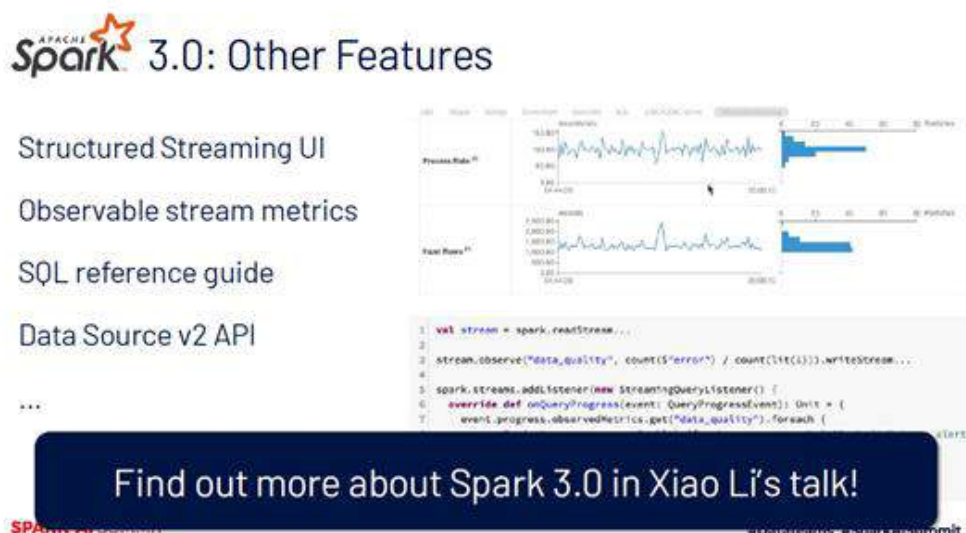
@pandas_udf('id', PandasUDFType.GROUPED_MAP)
def pandas_plus_one(pdf):
    # pdf is a pandas DataFrame
    return pdf + 1
```

SPARK-AI SUMMIT #DataTeams #SparkAiSummit

在性能方面，Spark 3.0做了大量Apache Arrow的性能升级，20%-25%的性能提升来自于Apache Arrow自身。Spark 3.0还使用Apache Arrow实现了大量Python 和R之间的数据交换，而这些对Spark使用者都是透明的。此外，Spark 3.0在SparkR方面的性能提升高达40倍，Spark还将提出更多的API来合并Pandas和Spark。



Spark 3.0新功能分布在不同的模块上，比如阿里巴巴贡献的可用来监控流计算的Structured Streaming UI，可检测的流作业监控指标，全新的Spark language查询手册，新的Data Source API等。更多的Spark 3.0新功能可参见Xiao Li的讲座。



**Spark 3.0: Other Features**

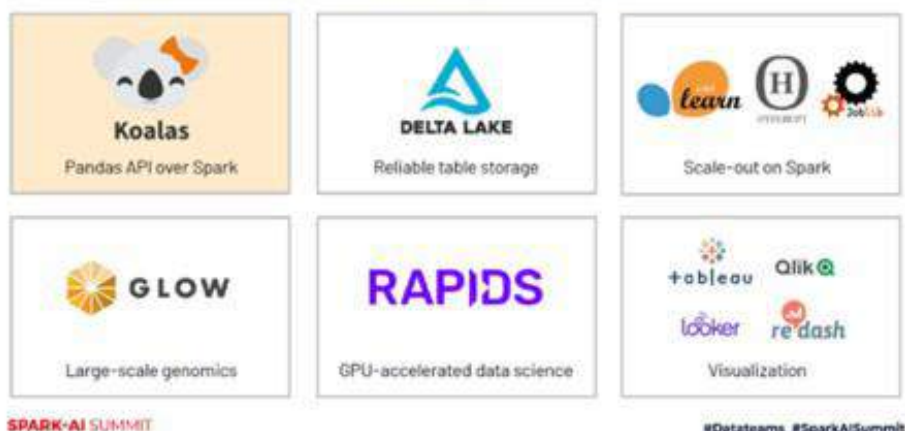
- Structured Streaming UI
- Observable stream metrics
- SQL reference guide
- Data Source v2 API







Find out more about Spark 3.0 in Xiao Li's talk!

## Spark生态圈的其它项目

除了Spark项目自身的发展，整个社区还围绕Spark 做出了许多创新。去年，Databricks发布了Koalas 项目,支持直接在Spark上运行Pandas API，使得更多的Pandas用户能够使用Spark解决大数据问题。Delta LAKE提供了可靠的表存储。社区还给Scikit learn, HYPEROPT, Joblib等添加了基于Spark的后端引擎，帮助它们解决大数据问题。Spark社区还和著名的基因公司一起开发了Glow项目，被大规模地应用于基因领域进行基因分析。Rapids提供了大量的数据科学和机器学习算法，使用GPU加速计算。最后，Databricks也进行了优化，改善了Spark 和可视化系统的交互，使得用户可以快速地开发以Spark作后端引擎的交互式界面。

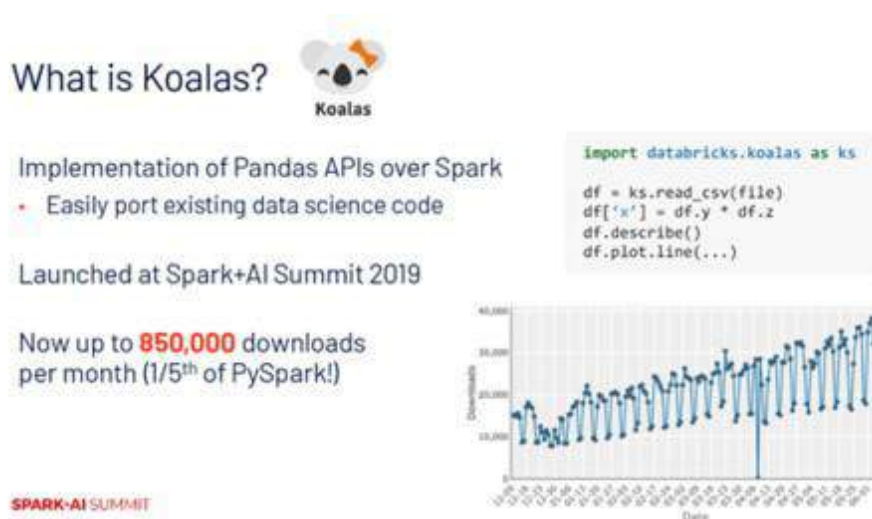
## Other Apache Spark Ecosystem Projects



 <b>Koalas</b> Pandas API over Spark	 <b>DELTA LAKE</b> Reliable table storage	 Scale-out on Spark
 <b>GLOW</b> Large-scale genomics	 <b>RAPIDS</b> GPU-accelerated data science	 Visualization

下面以Koalas为例展开具体介绍。

Koalas是Pandas API在Spark上的实现，能够使更多的数据科学代码直接运行在Spark上。从去年在Spark 3.0中发布至今，Koalas已经达到了每个月85 万的下载量，大约是PySpark的1/5。未来Spark社区还将在Koalas上投入更多。



这次，Spark社区还发布了Koalas 1.0，具有以下特性：

第一，达到了80%的Pandas API覆盖率。

第二，由于Spark 3.0的发布，Koalas的性能也大大提升。

第三，能够支持更多的功能，包括missing values, NA以及in-place updates等。

第四，具有分布式的索引类型。

第五，能够非常方便地使用pip进行安装，对Python用户非常友好。

## 四、Spark的未来

在回顾了Spark的发展过程后，接下来再来展望数据科学的发展和数据驱动的AI进步。显然，Spark在过去的十年生逢其时，多次重大的决策导致Spark发展神速。然而，基于Spark开发大数据和AI应用仍然过于复杂，让基于Spark的应用开发变得更加简单仍大有可为。为此，Spark 开源社区正在做更多的努力：

第一，要使数据的探索和使用变得更加简单，也就是数据科学和数据工程。

第二，还需要让Spark API更好地与生态圈的其它主流软件连接起来。

### What's Next for the Apache Spark Ecosystem?

If we step back, though software has made great strides, Data + AI apps are still **more complex to build than they should be**

We'll keep building on the two big lessons from past 10 years:

- Ease-of-use in both exploration and production
- APIs that connect to a rich software ecosystem

SPARK+AI SUMMIT

#DataTeams #SparkAISummit

接下来介绍Databricks在Apache Spark的接下来几个release中做出的具体努力。

### Zen

第一个项目叫Zen，中文名是禅。Zen的项目名来自Python社区的项目Python Zen，其定义了设计Python的一系列原则，正是这些原则带来了Python社区如今的繁荣。Zen项目旨在提高Apache Spark在Python方面的可用性，Spark社区希望通过Zen项目让Spark Python的使用和Python 生态圈的其它API一样易用。比如，提供更好的错误报告机制，将更易用的API加入Python的API中，进一步改善性能以使API更加Python化。

### AQE

Spark社区还将继续优化AQE的动态优化能力，同时改善用户体验，避免额外的数据准备工作。

### ANSI SQL

Spark社区还将标准化SQL语言支持，通过ANSI SQL使更多的主流SQL engine的工作能够迁移到Spark SQL中。

## OSS Spark Development Initiatives at Databricks

### Project Zen: Greatly improve Python usability

- Better error reporting
- API ports from Koalas
- Improved performance
- Pythonic API design



### Adaptive Query Execution: Cover most current optimizer decisions

### ANSI SQL: Run unmodified queries from major SQL engines

SPARK-AI SUMMIT

#Databricks #SparkAISummit

## 以Zen的Python Error Messages为例

在Spark 2.4中如果发生意味除零，用户会在Python Error Messages中得到冗长的错误信息，甚至还包括了大量的Java信息，这对Python程序员非常不友好。Spark 3.0中将简化Python Error Messages以避免冗余的错误信息，使用户能够快速查错。

## Python Error Messages

Spark 2.4

Caused by: org.apache.spark.api.python.PythonException: Traceback (most recent call last):  
File "/.../python/lib/spark.zip/spark/worker.py", line 585, in main  
process()  
File "/.../python/lib/spark.zip/spark/worker.py", line 587, in process  
serializer.dump\_stream(out\_iter, outfile)  
File "/.../python/lib/spark.zip/spark/serializers.py", line 223, in dump\_stream  
self.serializer.dump\_stream(self.\_batched\_iterator(), stream)  
File "/.../python/lib/spark.zip/spark/serializers.py", line 141, in dump\_stream  
for obj in iterator:  
File "/.../python/lib/spark.zip/spark/serializers.py", line 214, in \_batched  
for item in iterator:  
File "/.../python/lib/spark.zip/spark/worker.py", line 406, in wrapper  
result = func(f(x) for x in arg\_offsets) for (arg\_offset, f) in udfs)  
File "/.../python/lib/spark.zip/spark/worker.py", line 406, in wrapper  
result = func(f(x) for x in arg\_offsets) for (arg\_offset, f) in udfs)  
File "/.../python/lib/spark.zip/spark/worker.py", line 94, in lambda  
return lambda \*a: f(a)  
File "/.../python/lib/spark.zip/spark/util.py", line 387, in wrapper  
return f(\*args, \*\*kwargs)  
File "/.../python/lib/spark.zip/spark/util.py", line 387, in wrapper  
return f(\*args, \*\*kwargs)  
ZeroDivisionError: division by zero

## Python Error Messages

Spark 2.4

New Format

Traceback (most recent call last):  
File "/.../python/lib/spark.zip/spark/worker.py", line 427, in show  
print(load\_func(showing, 20, vertical))  
File "/.../python/lib/spark.zip/spark/worker.py", line 130, in \_call\_  
raise ValueError()  
File "/.../python/lib/spark.zip/spark/worker.py", line 585, in main  
process()  
File "/.../python/lib/spark.zip/spark/worker.py", line 587, in process  
serializer.dump\_stream(out\_iter, outfile)  
File "/.../python/lib/spark.zip/spark/serializers.py", line 223, in dump\_stream  
self.serializer.dump\_stream(self.\_batched\_iterator(), stream)  
File "/.../python/lib/spark.zip/spark/serializers.py", line 141, in dump\_stream  
for obj in iterator:  
File "/.../python/lib/spark.zip/spark/serializers.py", line 214, in \_batched  
for item in iterator:  
File "/.../python/lib/spark.zip/spark/worker.py", line 406, in wrapper  
result = func(f(x) for x in arg\_offsets) for (arg\_offset, f) in udfs)  
File "/.../python/lib/spark.zip/spark/worker.py", line 406, in wrapper  
result = func(f(x) for x in arg\_offsets) for (arg\_offset, f) in udfs)  
File "/.../python/lib/spark.zip/spark/worker.py", line 94, in lambda  
return lambda \*a: f(a)  
File "/.../python/lib/spark.zip/spark/util.py", line 387, in wrapper  
return f(\*args, \*\*kwargs)  
File "/.../python/lib/spark.zip/spark/util.py", line 387, in wrapper  
return f(\*args, \*\*kwargs)  
ZeroDivisionError: division by zero



## Python文档

Spark社区还提供了对用户更加友好的Python文档，而之前的PySpark API文档存在许多无用的API。



Spark社区对Spark的未来非常有信心。有理由相信，Spark 3.0将解决更多问题，让大数据和AI数据处理变得更加简单！

## OSS Spark Development Initiatives at Databricks

### Project Zen: Greatly improve Python usability

- Better error reporting
- API ports from Koalas
- Improved performance
- Pythonic API design



### Adaptive Query Execution: Cover most current optimizer decisions

### ANSI SQL: Run unmodified queries from major SQL engines

SPARK-AI SUMMIT

#Databricks #SparkAISummit

# Delta Lake 深度解析

## 数据工程师眼中的 Delta lake

简介：SPARK+AI SUMMIT 2020中文精华版线上峰会带领大家一起回顾2020年的SPARK又产生了怎样的最佳实践，技术上取得了哪些突破，以及周边的生态发展。本文中Databricks开源组技术主管范文臣从数据工程师的角度出发向大家介绍Delta Lake。以下是视频内容精华整理。

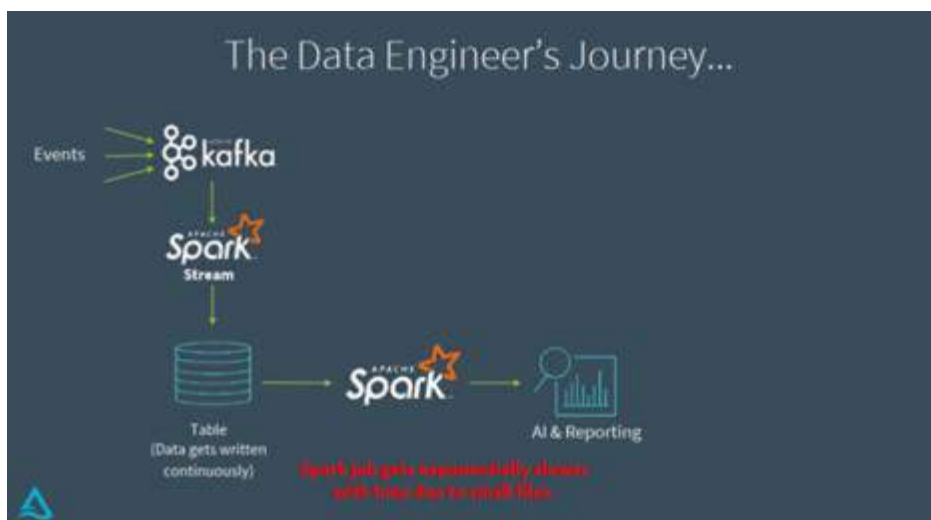
### 一、Delta Lake的诞生

相信作为一个数据工程师，心中都有这么一个理想的工具：

- 可以持续不断地对各种各样的数据源进行增量处理；
- 批流合一；
- 处理速率高效，智能化生成报表；
- .....



想要实现上面的工具，一个最简单的办法就是先用一个Spark Streaming Job把各种各样的数据源写到一个表中，如下图，然后再根据业务需求选择是用流作业还是批作业去进行相应的查询工作。但是，这种方式会存在一些问题，比如因为是流式写入，会产生大量的小文件，对后续的性能产生很大的影响。



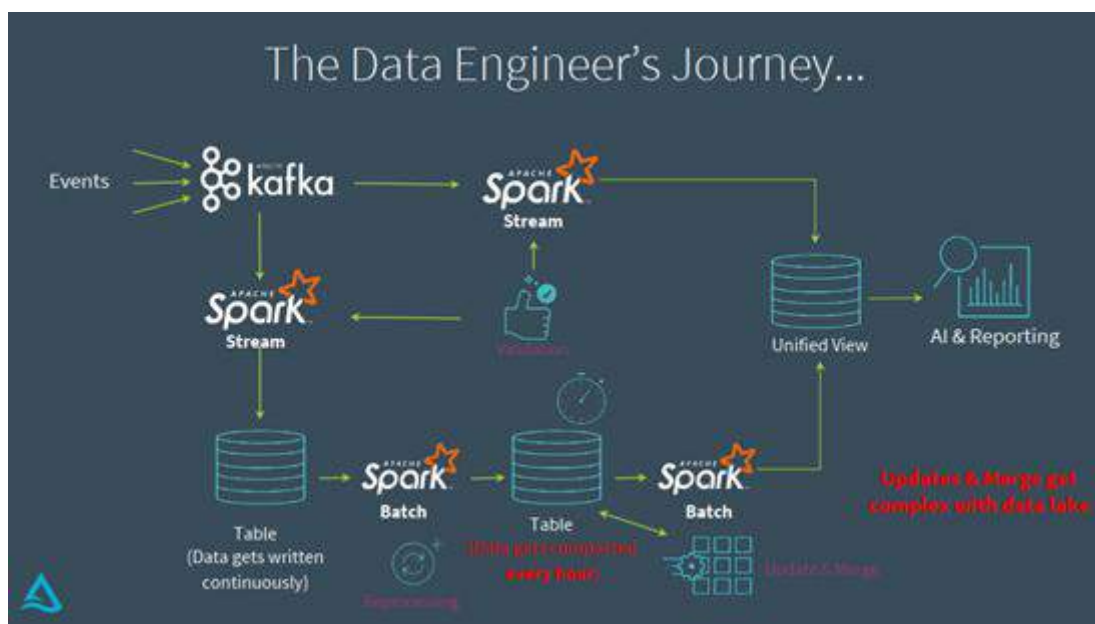
面对上面遇到的小文件问题，一个改进的方法如下图所示，是在上述方法中创建的表之后加一个批作业定时的将小文件合并起来，但是这个改进方法仍然有明显的缺点，那就是存在着小时级别的延迟，这种级别的延迟对于很多业务来讲是无法满足要求的。



为了解决上述延迟问题，Lambda架构畅行一时。其架构思路如下图所示，简单说就是分别用流和批的方式对数据源处理两次，然后将批和流的视角合起来提供给后续业务。Lambda架构虽然解决了上述的问题，但是也存在自身的缺点：

- 因为业务逻辑在要用批和流的方式处理两次，而批和流的处理方式不一致，可能会导致某些问题；

- 如果处理逻辑中加入了数据校验的工作，就需要在批和流上分别校验两次，一旦需要回滚等操作，数据修正也需要进行两次，费时费力；
- 如果涉及到Merge、Update等操作，也需要进行两次修改，使得整个事务变得复杂；
- . . . . .



上面的几种方案都有自己的缺点，Lambda架构虽然看似有效但是架构过于复杂。那么，有没有一种方案可以将Lambda架构进行简化呢？其实，我们的目标很简单，就是让流作业处理我们的源数据，并且后续作业可以批流统一的处理，具体来说有：

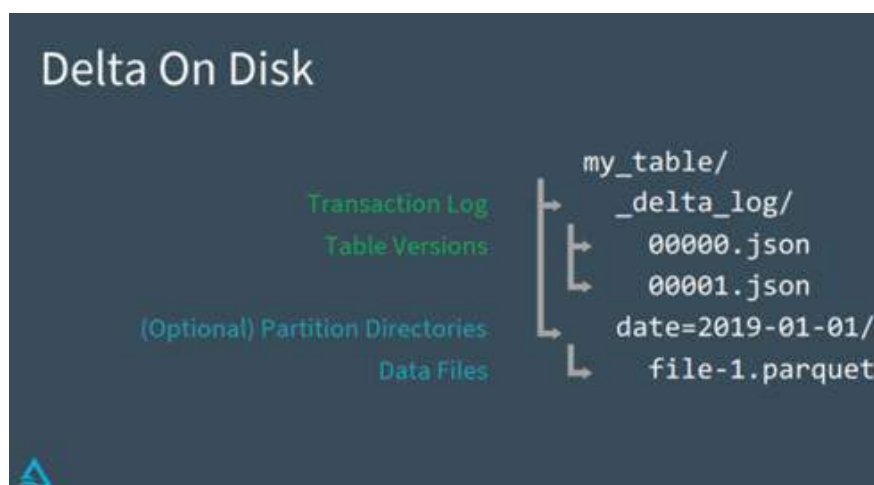
- 保证数据的一致性；
- 保证每次是增量的读取；
- 能够做回滚；
- 能够访问历史记录；
- 能够在不影响下游作业的同时合并小文件。

结合以上几点目标，有了目前的解决方案：Delta Lake + Structured Streaming = The Delta Architecture。这套方案的优点很明显，首先是批流合一的，其次Delta Lake可以很方便的做时间旅行类似的操作，且Delta Lake是单纯的储存层，与计算层分离，符合当前云数据计算的大方向，方便用户灵活的进行扩容。



## 二、Delta Lake的工作原理

Delta Lake的核心是其事务日志，它的表跟普通的表没有大的区别，但是在表下会建立一个隐藏文件，其中的JSON存储了一些关于事务的记录，如下图所示：



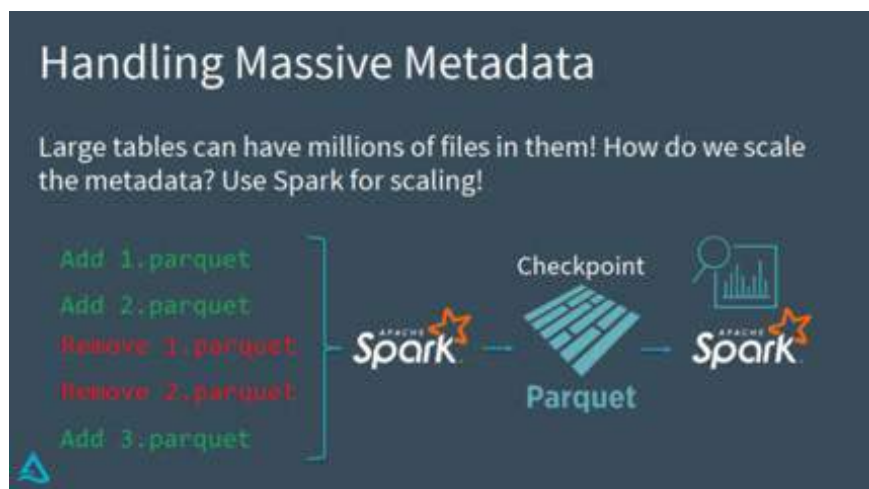
因此，在Delta Lake中，读取一张表也会重放这张表的历史记录，比如表的重命名、修改Schema等等操作。

更细节地来说，在Delta Lake中的每个JSON文件都是一次commit，这个commit是原子性的，保存了事务相关的详细记录。另外，Delta Lake还可以保证多个用户同时commit而不会产生冲突，它用的是一种基于乐观锁处理的方式，其逻辑如下图所示。这种解决冲突的方案适用于写比较少，读取比较多的场景，大家在使用的时候要注意场景是否适用。

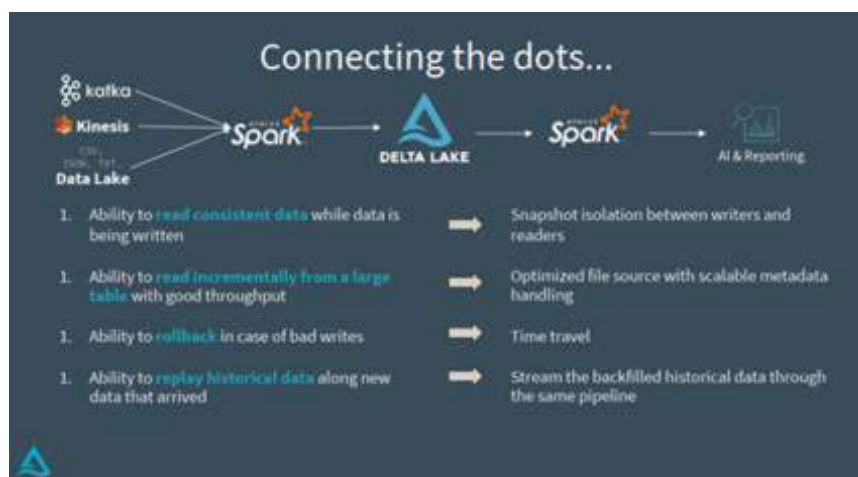


假设我们要处理一个非常大的表，有百万级别的文件，那么如何高效的处理元数据呢？

Delta Lake的处理方案如下图所示，用Spark来读取事务日志，然后Delta Lake隔一段时间对commit做一次合并，之后可以从Checkpoint开始应用后续的commit。



总结起来，Delta Lake解决数据一致性、增量读取、历史回溯等问题的方案即为下图所示：



### 三、Demo

Demo中提供了Python API和Scala API的实现文件，大家可以根据自己的实际情况进行尝试。上面链接的Demo中展示的主要features有：

- Schema Enforcement: 在做Pipeline的时候我们一定要保证数据质量，因此Schema

Enforcement 可以帮助我们做到这点。

- Schema Evolution: 随着公司业务的发展, 一开始的表结构可能不适用于当前的业务, Schema Evolution可以帮助我们进行表结构的演化。
- Delete from Delta Lake table: Delete操作可以控制表的无限制增长, 并且通过事务日志来进行操作, 实际上数据没有被删掉, 只是在Log中进行了标记。
- Audit Delta Lake Table History: 通过此功能可以看到对表的详细历史操作。
- Travel back in time: 有了表的历史数据, 我们便可以访问表在各个历史节点的数据。
- Vacuum old versions of Delta Lake tables: Delta Lake通过标记的方式来实现删除, 随着时间的增长会占用大量储存空间, Vacuum操作将删除在一定时间内从表中删除的数据文件, 实现物理删除, 默认会保留七天内的数据。
- Upsert into Delta Lake table using Merge: 在一个命令中同时做update和insert操作。

上述Features的具体代码可以在Github中查看。

## 四、Q&A

**Q1:** Delta Lake可以线上使用吗? 支持实时增删查改吗?

**A1:** Delta 最新发布了0.7.0, 支持Spark 3.0。Databricks 已经有很多客户在使用Delta Lake, 其他公司也有在用, 比如eBay。实时增删查改如demo演示的那样都是支持的。

**Q2:** 是否可以纯SQL实现?

**A2:** Delta Lake 是一个数据储存层, 如果是与Hive等引擎做整合, 只支持基本的SELECT/INSERT, 没有支持DELETE等SQL操作, 只能用Delta Lake自己的Scala或Python API。如果使用的是Spark 3.0的话, 像MERGE、DELTE等都支持SQL API, 可以直接用SQL开发。但是某些管理操作比如VACCUM没有对应的SQL API, 还是要用Delta Lake自己的Scala或Python API。

# Data Lake 三剑客 —— Delta、Hudi、Iceberg对比分析

**简介：**定性上讲，三者均为 Data Lake 的数据存储中间层，其数据管理的功能均是基于一系列的 meta 文件。meta 文件的角色类似于数据库的 catalog/wal，起到 schema 管理、事务管理和数据管理的功能。

**作者：**辛庸，阿里巴巴计算平台事业部 EMR 技术专家。Apache Hadoop, Apache Spark contributor。对 Hadoop、Spark、Hive、Druid 等大数据组件有深入研究。目前从事大数据云化相关工作，专注于计算引擎、存储结构、数据库事务等内容。

## 共同点

定性上讲，三者均为 Data Lake 的数据存储中间层，其数据管理的功能均是基于一系列的 meta 文件。meta 文件的角色类似于数据库的 catalog/wal，起到 schema 管理、事务管理和数据管理的功能。与数据库不同的是，这些 meta 文件是与数据文件一起存放在存储引擎中的，用户可以直接看到。这种做法直接继承了大数据分析中数据对用户可见的传统，但是无形中也增加了数据被不小心破坏的风险。一旦某个用户不小心删了 meta 目录，表就被破坏了，想要恢复难度非常大。

Meta 文件包含有表的 schema 信息。因此系统可以自己掌握 Schema 的变动，提供 Schema 演化的支持。Meta 文件也有 transaction log 的功能（需要文件系统有原子性和一致性的支持）。所有对表的变更都会生成一份新的 meta 文件，于是系统就有了 ACID 和多版本的支持，同时可以提供访问历史的功能。在这些方面，三者是相同的。

下面来谈一下三者的不同。

## Hudi

先说 Hudi。Hudi 的设计目标正如其名，Hadoop Upserts Deletes and Incrementals（原为 Hadoop Upserts and Incrementals），强调了其主要支持 Upserts、Deletes 和 Incremental 数据处理，其主要提供的写入工具是 Spark HudiDataSource API 和自身提供的 DeltaStreamer，均支持三种数据写入方式：UPSERT、INSERT 和 BULK\_INSERT。其对 Delete 的支持也是通过写入时指定一定的选项支持的，并不支持纯粹的 delete 接口。

其典型用法是将上游数据通过 Kafka 或者 Sqoop，经由 DeltaStreamer 写入 Hudi。DeltaStreamer 是一个常驻服务，不断地从上游拉取数据，并写入 hudi。写入是分批次的，并且可以设置批次之间的调度间隔。默认间隔为 0，类似于 Spark Streaming 的 As-soon-as-possible 策略。随着数据不断写入，会有小文件产生。对于这些小文件，DeltaStreamer 可以自动地触发小文件合并的任务。

在查询方面，Hudi 支持 Hive、Spark、Presto。在性能方面，Hudi 设计了 HoodieKey，一个类似于主键的东西。HoodieKey 有 Min/Max 统计，BloomFilter，用于快速定位 Record 所在的文件。在具体做 Upserts 时，如果 HoodieKey 不存在于 BloomFilter，则执行插入，否则，确认 HoodieKey 是否真正存在，如果真正存在，则执行 update。对于查询性能，一般需求是根据查询谓词生成过滤条件下推至 datasource。Hudi 这方面没怎么做工作，其性能完全基于引擎自带的谓词下推和 partition prune 功能。

Hudi 的另一大特色是支持 Copy On Write 和 Merge On Read。前者在写入时做数据的 merge，写入性能略差，但是读性能更高一些。后者读的时候做 merge，读性能差，但是写入数据会比较及时，因而后者可以提供近实时的数据分析能力。

最后，Hudi 提供了一个名为 run\_sync\_tool 的脚本同步数据的 schema 到 Hive 表。Hudi 还提供了一个命令行工具用于管理 Hudi 表。

## hudi



## Iceberg

Iceberg 没有类似的 HoodieKey 设计，其不强调主键。上文已经说到，没有主键，做 update/delete/merge 等操作就要通过 Join 来实现，而 Join 需要有一个类似 SQL 的执行引擎。Iceberg 并不绑定某个引擎，也没有自己的引擎，所以 Iceberg 并不支持 update/delete/merge。如果用户需要 update 数据，最好的方法就是找出哪些 partition 需要更新，然后通过 overwrite 的方式重写数据。Iceberg 官网提供的 quickstart 以及 Spark 的接口均只是提到了使用 Spark dataframe API 向 Iceberg 写数据的方式，没有提及别的数据摄入方法。至于使用 Spark Streaming 写入，代码中是实现了相应的 StreamWriteSupport，应该是支持流式写入，但是貌似官网并未明确提及这一点。支持流式写入意味着有小文件问题，对于怎么合并小文件，官网也未提及。我怀疑对于流式写入和小文件合并，可能 Iceberg 还没有很好的生产 ready，因而没有提及（纯属个人猜测）。

在查询方面，Iceberg 支持 Spark、Presto。Iceberg 在查询性能方面做了大量的工作。



Iceberg 在查询性能方面做了大量的工作。值得一提的是它的 hidden partition 功能。Hidden partition 意思是说，对于用户输入的数据，用户可以选取其中某些列做适当的变换（Transform）形成一个新的列作为 partition 列。这个 partition 列仅仅为了将数据进行分区，并不直接体现在表的 schema 中。例如，用户有 timestamp 列，那么可以通过 hour(timestamp) 生成一个 timestamp\_hour 的新分区列。timestamp\_hour 对用户不可见，仅仅用于组织数据。Partition 列有 partition 列的统计，如该 partition 包含的数据范围。当用户查询时，可以根据 partition 的统计信息做 partition prune。

除了 hidden partition，Iceberg 也对普通的 column 列做了信息收集。这些统计信息非常全，包括列的 size，列的 value count，null value count，以及列的最大最小值等等。这些信息都可以用来在查询时过滤数据。

Iceberg 提供了建表的 API，用户可以使用该 API 指定表明、schema、partition 信息等，然后在 Hive catalog 中完成建表。

## Delta

我们最后来说 Delta。Delta 的定位是流批一体的 Data Lake 存储层，支持 update/delete/merge。由于出自 Databricks，spark 的所有数据写入方式，包括基于 dataframe 的批式、流式，以及 SQL 的 Insert、Insert Overwrite 等都是支持的（开源的 SQL 写暂不支持，EMR 做了支持）。与 Iceberg 类似，Delta 不强调主键，因此其 update/delete/merge 的实现均是基于 spark 的 join 功能。在数据写入方面，Delta 与 Spark 是强绑定的，这一点 Hudi 是不同的：Hudi 的数据写入不绑定 Spark（可以用 Spark，也可以使用 Hudi 自己的写入工具写入）。

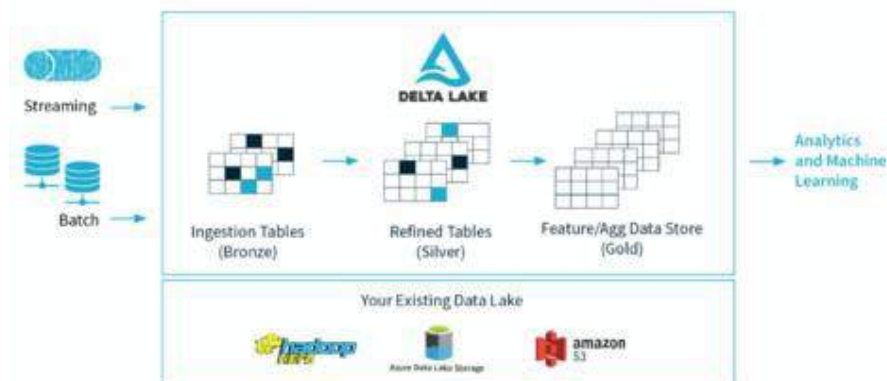
在查询方面，开源 Delta 目前支持 Spark 与 Presto，但是，Spark 是不可或缺的，因为 delta log 的处理需要用到 Spark。这意味着如果要用 Presto 查询 Delta，查询时还要跑一个 Spark 作业。更为蛋疼的是，Presto 查询是基于 SymlinkTextInputFormat。在查询之前，要运行 Spark 作业生成这么个 Symlink 文件。如果表数据是实时更新的，意味着每次在查询之前先要跑一个 SparkSQL，再跑 Presto。这样的话为何不都在 SparkSQL 里搞定呢？这是一个非常蛋疼的设计。为此，EMR 在这方面做了改进，支持了 DeltaInputFormat，用户可以直接使用 Presto 查询 Delta 数据，而不必事先启动一个 Spark 任务。

在查询性能方面，开源的 Delta 几乎没有任何优化。Iceberg 的 hidden partition 且不说，普通的 column 的统计信息也没有。Databricks 对他们引以为傲的 Data Skipping 技术做了保留。不得不说这对于推广 Delta 来说不是件好事。EMR 团队在这方面正在做一些工作，希望能弥补这方面能力的缺失。

Delta 在数据 merge 方面性能不如 Hudi，在查询方面性能不如 Iceberg，是不是意味着 Delta 一无是处了呢？其实不然。Delta 的一大优点就是与 Spark 的整合能力（虽然目前仍不是很完善，但 Spark-3.0 之后会好很多），尤其是其流批一体的设计，配合 multi-hop 的 data pipeline，可以

支持分析、Machine learning、CDC 等多种场景。使用灵活、场景支持完善是它相比 Hudi 和 Iceberg 的最大优点。另外，Delta 号称是 Lambda 架构、Kappa 架构的改进版，无需关心流批，无需关心架构。这一点上 Hudi 和 Iceberg 是力所不及的。

## delta



## 总结

通过上面的分析能够看到，三个引擎的初衷场景并不完全相同，Hudi 为了 incremental 的 upserts，Iceberg 定位于高性能的分析与可靠的数据管理，Delta 定位于流批一体的数据处理。这种场景的不同也造成了三者在设计上的差别。尤其是 Hudi，其设计与另外两个相比差别更为明显。随着时间的发展，三者都在不断补齐自己缺失的能力，可能在将来会彼此趋同，互相侵入对方的领地。当然也有可能各自关注自己专长的场景，筑起自己的优势壁垒，因此最终谁赢谁输还是未知之数。

下表从多个维度对三者进行了总结，需要注意的是此表所列的能力仅代表至 2019 年底。

	Delta	Hudi	Iceberg
Incremental Ingestion	Spark	Spark	Spark
ACID updates	HDFS, S3 (Databricks), OSS	HDFS	HDFS, S3

Upserts/Delete/Merge/ Update	Delete/Merge/ Update	Upserts /Delete	No
Streaming sink	Yes	Yes	Yes(not ready?)
Streaming source	Yes	No	No
FileFormats	Parquet	Avro,Parquet	Parquet, ORC
Data Skipping	File-Level Max-Min stats + Z-Ordering (Databricks)	File-Level Max-Min stats + Bloom Filter	File-Level Max- Min Filtering
Concurrency control	Optimistic	Optimistic	Optimistic
Data Validation	Yes (Databricks)	No	Yes

Merge on read	No	Yes	No
Schema Evolution	Yes	Yes	Yes
File I/O Cache	Yes (Databricks)	No	No
Cleanup	Manual	Automatic	No
Compaction	Manual	Automatic	No

注：限于本人水平，文中内容可能有误，也欢迎读者批评指正！

# 核桃编程 Delta Lake 实时数仓应用实践

简介： 本文简述了核桃编程应用EMR建设 Delta Lake 实时数仓的实践。

作者：卢圣刚，核桃编程数据架构师，拥有多年的大数据开发和架构经验。曾担任易观数据挖掘工程师，熊猫TV大数据架构师。

## 核桃编程简介

核桃编程成立于2017年8月9日，作为少儿编程教育行业的领导者，始终秉持“让每个孩子爱学习、会学习，让优质的教育触手可及”的使命，致力于以科技手段促进编程教育，凭借首创的AI人机双师教学模式与十级进阶课程体系，实现规模化因材施教，“启发中国孩子的学习力”。截止2019年8月，核桃编程已经成为付费学员规模最大的少儿编程教育机构，帮助超过65万名孩子收获学习兴趣，锻炼编程技能，养成良好思维习惯，学员复购率超91%，学员完课率高达98%，在线原创作品1873万份。

## 1. 业务现状

### 业务需求

- 业务上固定时间开课，在开课时间内，班主任需要实时/准实时地知道学生的学习情况
- 数据统计维度一般都是按班级，学期汇总，时间范围可能是几个月，甚至一年
- 业务变化快，需要及时响应业务变化带来的指标逻辑变更

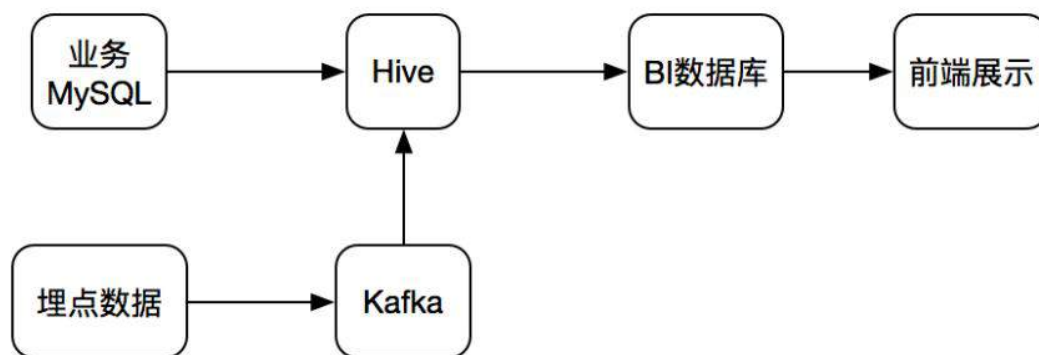
### 数据源

数据源	备注
Nginx accesslog	用户行为日志
Mysql binlog	通过阿里云DTS工具同步到Kafka
Kafka	业务方采集日志写入到Kafka集群



## 架构改造前方案

现有指标都是将Kafka/Mysql等的数写入HDFS，使用Hive离线批处理，每10分钟执行一次，循环统计历史累计指标，再定时把数据同步到Mysql，提供给数据后台查询。如下图所示：



## 2.实时数仓方案调研

离线的同步方案已经不能满足业务需求，计划迁移到实时方案上来，并做了一些调研。

### 迁移流式计算的问题

#### 开发周期长

现有离线任务基本都是动辄几百行SQL，逻辑复杂，把所有逻辑迁移到流式计算，开发难度和改造成本都比较大。

例如离线增量同步，需要先同步全量base数据

```
sqoop import \  
--hive-import \  
--hive-overwrite \  
--connect jdbc:mysql://<mysqlurl> \  
--table <mysqltable> \  
--hive-table <table_base> \  
--hive-partition-key <parcolumn> \  
--hive-partition-value <par1>
```

再消费增量binlog数据，流式写入到hive外部表，最后将两个表合并

```
insert overwrite table <result_storage_table>select
<col1>,

<col2>,
    <colN>
from(select
row_number() over(partition by t.<primary_key_column>
order by record_id
desc, after_flag desc) as row_number, record_id, operation_flag, after_flag,
<col1>, <col2>, <colN>
from(select
incr.record_id, incr.operation_flag, incr.after_flag, incr.<col1>,
incr.<col2>,incr.<colN>
from
<table_log> incr
where
utc_timestamp< <timestamp>
union all select 0
as record_id, 'I' as operation_flag, 'Y' as after_flag, base.<col1>,
base.<col2>,base.<colN>
from
<table_base> base) t) gtwhere record_num=1
and
after_flag='Y'
```

而应用Delta Lake只需要一个streaming sql即可实现实时增量同步。

```
CREATE SCAN <SCAN_TABLE> on <STREAM> using
stream;CREATE STREAM job
OPTIONS(
  checkpointLocation='/cdc',
  triggerInterval=30000
)MERGE INTO <CDC_TABLE> as targetUSING (
  SELECT
    from_unixtime(<col2>,'yyyyMMdd') as
par_date,
    <col1>
  FROM(
    SELECT
      recordId,
      recordType,
      CAST(before.id asLONG) as before_id,
      CAST(after.id asLONG) as id,

after.<col1>,
    after.ctime,
    dense_rank() OVER
(PARTITION BY coalesce(before.id,after.id) ORDER BY recordId DESC) as rank
  FROM (
    SELECT
      recordId,
      recordType,
```

```
from_json(CAST(beforeImages as STRING), 'id STRING, <col1>
<coltype1>,ctime string') as before,

from_json(CAST(afterImages as STRING), 'id STRING, <col1>
<coltype1>,ctime string') as after

FROM (

    select

from_avro(value) as (recordID, source, dbTable, recordType,
recordTimestamp,
extraTags, fields, beforeImages, afterImages) from <SCAN_TABLE>

    ) binlog WHERE
recordType != 'INIT'

    ) binlog_wo_init

    ) binlog_extractWHERE rank=1

) as sourceON target.id = source.before_idWHEN MATCHED AND
source.recordType='UPDATE' THENUPDATE SET *WHEN MATCHED AND
source.recordType='DELETE' THENDELETEWHEN NOT MATCHED AND
(source.recordType='INSERT' OR
source.recordType='UPDATE') THENINSERT *;
```

## 数据恢复困难

对离线任务来说数据恢复只需要重新执行任务就行。

但对流式计算，当数据异常，或者逻辑变更，需要重新跑全量数据的时候，只能离线补历史数据，再union实时数据。因为Kafka不可能存所有历史数据，而且从头消费追数据时间也会很久。

而为了满足快速恢复的需求，所有指标都需要从一开始准备离线和实时两套代码，类似Lambda架构。

## 数据验证困难

Kafka在大数据架构中一般充当消息队列的角色，数据保存周期较短。全量历史数据，会消费Kafka写到HDFS。如果一个指标计算了一个月，发现计算结果有异常，很难追溯是当时Kafka数据有问

题，还是计算逻辑有问题。HDFS数据虽然可以用来排查，但是HDFS里的数据和当时Kafka的数据是否一致，是不能保证的。

## 希望满足的功能

正因为迁移流式作业会有一些迁移成本和问题，所以对实时计算方案提出了一些功能要求。

## 开发灵活

互联网公司业务发展速度快，人力资源比较紧张，需要更低成本更快捷的开发新指标，满足业务敏捷性的要求。

## 重跑历史数据方便

业务指标的定义经常发生变更，一旦变更，或者有新的数据指标就需要从最早开始消费。但是历史数据通常非常多，而且一般实时数据源Kafka也不可能存历史所有数据。

## 数据异常时容易排查问题

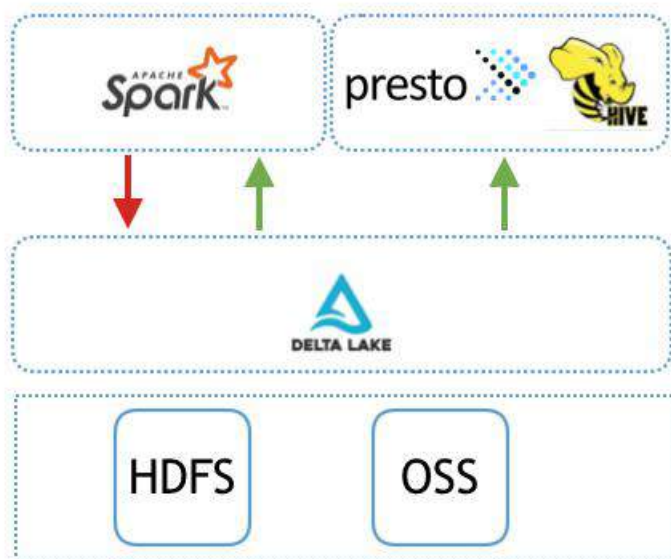
以离线数仓为例，几百行的SQL,可以分段执行，来逐步排查。Flink可以埋metrics获取中间过程。

## 3.基于Delta Lake实时数仓方案

### Delta Lake

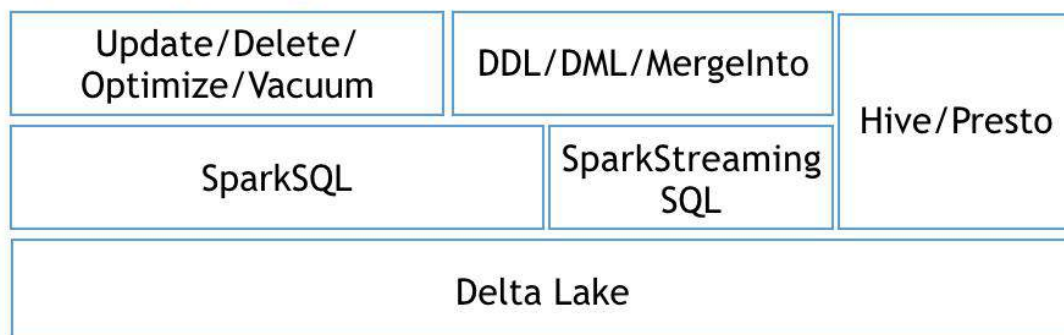
Delta Lake是美国Databricks开源的数据湖技术，基于Apache Parquet丰富了数据管理功能，如元数据管理/事务/数据更新/数据版本回溯等。使用Delta Lake可以很方便的将流处理和批处理串联起来，快速构建Near-RealTime的Data Pipeline.





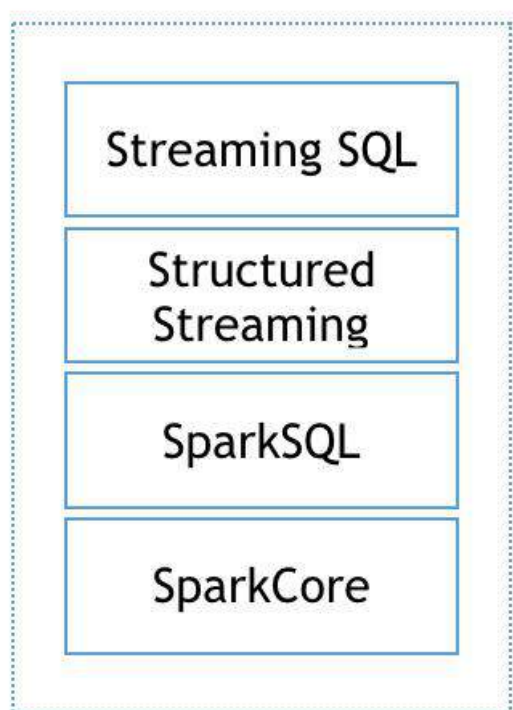
目前阿里巴巴E-MapReduce（简称“EMR”）团队对Delta Lake做了很多功能和性能上的优化，并和Spark做了深度集成，主要以下方面，更多信息详见[EMR官方文档](#)

- SparkSQL支持Update/Delete/Merge Into/Optimize/Vacuum等语法来操作Delta Lake
- 自研SparkStreaming SQL，支持Delta Lake的相关DML操作
- Hive&Presto On Delta Lake
- Delta Lake On OSS(阿里云对象存储)
- Delta Lake事务冲突检测优化
- DataSkipping & Zorder性能优化



## SparkStreaming SQL

阿里巴巴EMR团队在StructStreaming基础上自研了SparkStreaming SQL，用户可以很方便的使用SQL来写流式作业的逻辑，大大降低了开发门槛，详见 [SparkStreaming SQL官方文档](#)。

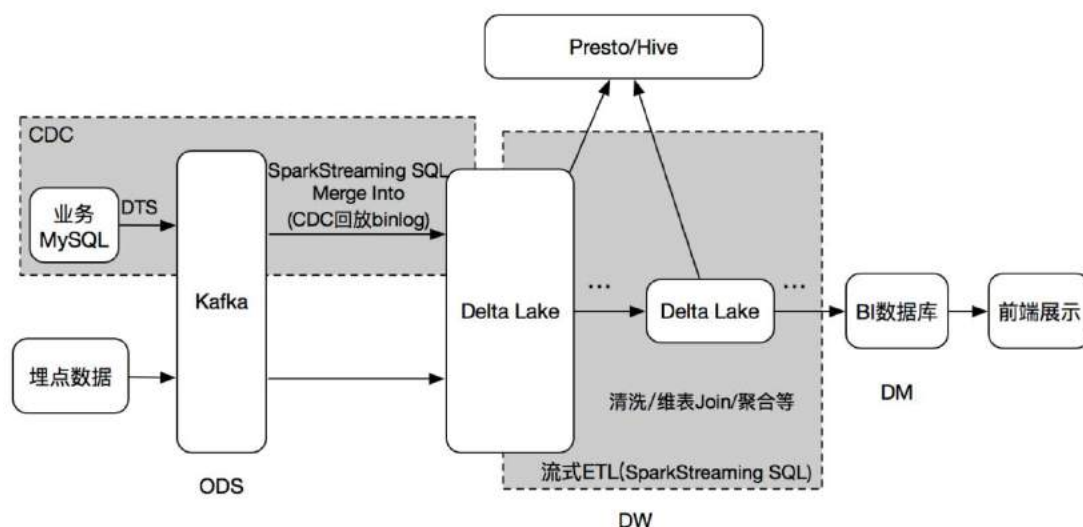


- 批流统一引擎  
可以复用底层SparkSQL/SparkCore的优化
  - 丰富的SQL支持  
CREATE TABLE / CREATE SCAN / CREAT STREAM / CTAS
  - INSERT INTO / MERGE INTO  
SELECT / WHERE/ GROUP BY / JOIN / UNION ALL
  - 丰富的UDF支持  
Hive UDF / 窗口函数
  - 丰富的数据源支持  
Delta/Kudu/Druid/HBase/MySQL/Redis/SLS/Datahub/TableStore
  - 并且支持Kafka的Exactly Once
- github: <https://github.com/aliyun/aliyun-emapreduce-sdk>
- Delta Lake深度集成  
结合Delta Lake的使用场景，新增了一些功能的支持(比如流式写动态分区表)

## 实时数仓方案

### 架构方案

基于Delta Lake+SparkStreaming SQL可以快速构建实时数仓的pipeline，如下所示：



- ODS层  
ODS的数据主要是实时埋点数据，CDC中的binlog日志等
- DIM维表
- 
- DW层  
DW层主要是一部分轻度汇总数据，例如用户维度的课程，作业等信息。

主要复用的是dw层数据，因此针对每一个指标，需要综合考虑是否聚合，聚合到哪一个维度，是否关联维表。

DW层分为两种

- 业务简单，基本不会变化。直接写入Kafka。
- 业务逻辑复杂，数据可能<频繁>变化，写入Delta Lake。实践上看，直接写入Kafka是最容易的方案，但是灵活性很低，历史数据无法追溯，也无法修改。DW层通过引入Delta Lake，可以实现流批统一数据源，历史分区数据恢复等功能。

DM层

DM层就是最后的报表展示指标了，可以将DW层delta表做为数据源，再次汇总后sink到展示用的DataBase。

备注: EMR团队提供了流式Merge Into功能, 可以通过写SparkStreaming SQL的方式来做CDC回放binlog到Delta表。

详见[CDC同步文档](#)。

## 问题的优化

在使用Delta Lake的过程中, 我们也发现了一些问题, 详细的解决方案和建议如下:

### 小文件多

CDC流式Merge回放binlog的过程中, 会不断产生小文件, 需要对小文件进行一些处理, EMR提供了一些优化方案

- 新增串行auto compaction的功能  
在CDC流式作业运行过程中, 根据一定的策略对小文件进行合并compact操作
- 使用Adaptive Execution  
打开自适应执行开关, 可以有效减少Merge过程产生的小文件, 如单个batch从100个小文件减少到1~2个文件。

## Compact冲突问题

如果不使用串行Compact功能, 需要定期手工对Delta表进行Compact合并小文件, 但是经常碰到Compact在事务提交的时候和CDC流作业事务提交产生冲突, 是的CDC流或者Compact失败, 这块也提供了一些优化以及建议:

- 优化Delta内核冲突机制, 使得CDC流能够稳定运行, 不会因为Compact挂掉
- 使用分区表, 批量对分区进行Compact, 减少冲突概率
- 在数据库表update/delete操作很少的时候进行Compact(可以使用EMR工作流调度)
- 使用EMR工作流中的作业重试功能, 当遇到Compact事务提交失败时进行重试

## 架构方案进一步说明

### ● 为什么不直接从ODS计算

以核桃的到课指标为例, 数据源是kafka的埋点topic, 需要计算的指标有个人维度到课数据, 学期维度, 班级维度, 学期维度, 市场渠道维度。  
每个维度都需要消费所有的埋点数据, 从中挑出到课相关的事件。并且每个维度的计算程序都需要

查询HBase/MySQL关联相关的学期，班级，unit等维表。

一旦有整体逻辑的调整，例如过滤测试班数据，不可能从ods层就把数据过滤掉(这样从底层就开始丢失数据，后期无法追查)，那么所有程序都需要重新调整，添加这个过滤逻辑。

## ● 怎么恢复数据

理想情况是，实时与离线使用同一套SQL，同一套计算逻辑，同一个数据源，这样随时可以用离线脚本重跑历史数据。但是现实是没有哪个框架支持。所谓流批一体，都是在引擎层面，例如Spark的streaming和SQL都是batch的方式，流只是更小的批。而Flink则希望用流的方式去处理批数据，批只是有边界的流。针对高阶的SQL API，流批都有很大的区别。基于Delta Lake的分区表，将dw层的实时数据按时间分区，这样可以随时用离线作业恢复历史分区的数据。而DW之上的汇总因为数据量相对较小，恢复之后可以用流作业从头消费。

### 1. 业务效果

Delta Lake实时数仓在核桃编程部分数据仓库生产环境上线后，部分业务统计指标已基于新架构产出，指标更新延迟从几十分钟，提升到1分钟以内。班主任可以更快获取学生的学习状态，及时跟进学习进度,从而显著提升了教学质量。

在CDC应用后，数据同步延迟从半小时提升到30秒，同时解决了Sqoop高并发同步时对业务数据库的影响。数据分析人员Ad-Hoc查询时，可以获取实时的业务数据，明显提升了数据分析效果，并且可以更及时的指导业务发展。

### 2. 后续计划

根据目前的业务应用效果，后续大数据团队会继续梳理业务范围所有实时指标，进一步优化实时数仓各层的结构，推进全面应用基于Delta Lake的实时数仓建设。

基于Delta Lake模式执行、时间旅行等特性，进一步推进机器学习场景下对Delta的应用，构造更可靠、易扩展的Data Pipeline。

# “脏数据” 走开：Schema 约束和 Schema 演变

简介：Schema 约束和 Schema 演变相互补益，合理地结合起来使用将能方便地管理好数据，避免脏数据侵染，保证数据的完整可靠。

编译：辰山，阿里巴巴计算平台事业部 EMR 高级开发工程师，目前从事大数据存储方面的开发和优化工作

在实践经验中，我们知道数据总是在不断演变和增长，我们对于这个世界的心智模型必须要适应新的数据，甚至要应对我们从前未知的知识维度。表的 schema 其实和这种心智模型并没什么不同，需要定义如何对新的信息进行分类和处理。

这就涉及到 schema 管理的问题，随着业务问题和需求的不断演进，数据结构也会不断发生变化。通过 Delta Lake，能够很容易包含数据变化所带来的新的维度，用户能够通过简单的语义来控制表的 schema。相关工具主要包括 Schema 约束（Schema Enforcement）和 Schema 演变（Schema Evolution），前者用以防止用户脏数据意外污染表，后者用以自动添加适当的新数据列。本文将详细剖析这两个工具。

## 理解表的 Schemas

Apache Spark 的每一个 DataFrame 都包含一个 schema，用来定义数据的形态，例如数据类型、列信息以及元数据。在 Delta Lake 中，表的 schema 通过 JSON 格式存储在事务日志中。

## 什么是 Schema 约束？

Schema 约束（Schema Enforcement），也可称作 Schema Validation，是 Delta Lake 中的一种保护机制，通过拒绝不符合表 schema 的写入请求来保证数据质量。类似于一个繁忙的餐厅前台只接受预定坐席的顾客，这个机制会检查插入表格的每一列是否符合期望的列（换句话说，就是检查每个列是否已经“预定坐席”），那些不在期望名单上的写入将被拒绝。



## Schema 约束如何工作？

Delta Lake 对写入进行 schema 校验，也就是说所有表格的写入操作都会用表的 schema 做兼容性检查。如果 schema 不兼容，Delta Lake 将会撤销这次事务（没有任何数据写入），并且返回相应的异常信息告知用户。

Delta Lake 通过以下准则判断一次写入是否兼容，即对写入的 DataFrame 必须满足：

- 不能包含目标表 schema 中不存在的列。相反，如果写入的数据没有包含所有的列是被允许的，这些空缺的列将会被赋值为 null。
- 不能包含与目标表类型不同的列。如果目标表包含 String 类型的数据，但 DataFrame 中对列的数据类型为 Integer，Schema 约束将会返回异常，防止该次写入生效。
- 不能包含只通过大小写区分的列名。这意味着不能在一张表中同时定义诸如 “Foo” 和 “foo” 的列。不同于 Spark 可以支持大小写敏感和不敏感（默认为大小写不敏感）两种不同的模式，Delta Lake 保留大小写，但在 schema 存储上大小写不敏感。Parquet 在存储和返回列信息上面是大小写敏感的，因此为了防止潜在的错误、数据污染和丢失的问题，Delta Lake 引入了这个限制。

以下代码展示了一次写入过程，当添加一次新计算的列到 Delta Lake 表中。

*# Generate a DataFrame of Loans that we'll append to our Delta Lake table*

```
loans = sql("""
    SELECT addr_state, CAST(rand(10)*count as bigint) AS count,
    CAST(rand(10) * 10000 * count AS double) AS amount
    FROM loan_by_state_delta
    """)
```

*# Show original DataFrame's schema*

```
original_loans.printSchema()
```

```
"""
```

```
root
```

```
 |-- addr_state: string (nullable = true)
```

```
 |-- count: integer (nullable = true)"""
```

*# Show new DataFrame's schema*

```
loans.printSchema()
```

```
"""
root
|-- addr_state: string (nullable = true)
|-- count: integer (nullable = true)
|-- amount: double (nullable = true) # new column"""

# Attempt to append new DataFrame (with new column) to existing table
loans.write.format("delta") \
    .mode("append") \
    .save(DELTALAKE_PATH)

""" Returns:

A schema mismatch detected when writing to the Delta table.

To enable schema migration, please set:'.option("mergeSchema", "true")\'

Table schema:
root
-- addr_state: string (nullable = true)
-- count: long (nullable = true)

Data schema:
root
-- addr_state: string (nullable = true)
-- count: long (nullable = true)
-- amount: double (nullable = true)
```

```
If Table ACLs are enabled, these options will be ignored. Please use the ALTER TABLE command for changing the schema.
```

不同于自动添加新的列，Delta Lake 受到 schema 约束并阻止了这次写入生效。并且为了帮助定位是哪个列造成了不匹配，Spark 会在错误栈中打印出两者的 schema 作为对照。

## Schema 约束有何作用？

由于 Schema 约束是一种严格的校验，因此可以用于已清洗、转化完成的数据，保证数据不受污染，可用于生产或者消费。典型的应用场景包括直接用于以下用途的表：

- 机器学习算法
- BI 仪表盘
- 数据分析和可视化工具
- 任何要求高度结构化、强类型、语义 schema 的生产系统
- 

为了准备好最终的数据，很多用户使用简单的“多跳”架构来逐步往表中添加结构。更多相关内容可以参考 [Productionizing Machine Learning With Delta Lake](#)。

当然，Schema 约束可以用在整个工作流程的任意地方，不过需要注意的是，有可能因为诸如不经意对写入数据添加了某个列，导致写入流失败的情况。

## 防止数据稀释

看到这，你可能会问，到底需不需要大费周章做 Schema 约束？毕竟，有时候一个意料之外的 schema 不匹配问题反而会影响整个工作流，特别是当新手使用 Delta Lake。为什么不直接让 schema 接受改变，这样我们就能任意写入 DataFrame 了。

俗话说，防患于未然，有些时候，如果不对 schema 进行强制约束，数据类型兼容性的问题将会很容易出现，看上去同质的数据源可能包含了边缘情况、污染列、错误变换的映射以及其他可怕的情况都可能会一夜之间污染了原始的表。所以更好的做法应该从根本上阻止这样的情况发生，通过 Schema 约束就能够做到，将这类错误显式地返回进行恰当的处理，而不是让它潜伏在数据中，看似写入时非常顺利，但埋下了无法预知的隐患。

Schema 约束能够确保表 schema 不会发生改变，除非你确切地执行了更改操作。它能有效的防止“数据稀释”——当新的列频繁添加，原本简洁的表结构可能因为数据泛滥而失去原有的含义和用处。Schema 约束的设计初衷就是通过设定严格的要求来保证质量，确保表数据不受污染。另一方面，假如经过再三确认之后，确定的确需要添加新的列，那解决方法也非常简单，也就是下文即将介绍的 Schema 演变！

## 什么是 Schema 演变

Schema 演变（Schema Evolution）允许用户能够方便地修改表的当前 schema，来适应不断变化的数据。最常见的用法就是在执行添加和覆盖操作时，自动地添加一个或多个列来适应 schema。

## Schema 演变如何工作？

继续沿用上文的例子，对于之前由于 schema 不匹配导致请求被拒绝的情况，开发人员可以方便地使用 Schema 演变来添加新的列。Schema 演变的使用方式是在 `.write` 或 `.writeStream` 的 Spark 命令后面添加上 `.option('mergeSchema', 'true')`。

```
# Add the mergeSchema option
loans.write.format("delta") \
    .option("mergeSchema", "true") \
    .mode("append") \
    .save(DELTALAKE_SILVER_PATH)
```

可以执行以下 Spark SQL 语句来察看图表。

```
# Create a plot with the new column to confirm the write was successful
%sql
SELECT addr_state, sum(`amount`) AS amount
FROM loan_by_state_delta
GROUP BY addr_state
ORDER BY sum(`amount`)
DESC LIMIT 10
```

当然，也可以选择通过添加 `spark.databricks.delta.schema.autoMerge = True` 到 Spark 配置文件中使得该选项对整个 Spark session 生效。不过需要注意的是，这样使用的话，Schema 约束将不再会对 schema 不匹配问题进行报警提示。

通过指定 `mergeSchema` 选项，所有在输入 DataFrame 中存在但在目标表中不存在的列都将被作为该事务操作的一部分添加到 schema 末尾。也允许添加嵌套字段，这些字段将被添加到对应列的末尾。

数据科学家可以利用这个选项来添加新的列（例如一个新增的跟踪指标，或是这个月的销售数据）

到已有的机器学习表中，而不必废弃现有依赖于旧的列信息的模型。

以下对表的添加和覆盖操作都是合法的 Schema 演变操作：

- 添加新列（这是最常用的场景）
- 修改数据类型，Null->其他类型，或者向上类型转换 Byte->Short->Integer

其他改动都是非法的 Schema 演变操作，需要通过添加 `.option("overwriteSchema", "true")` 选项来覆盖 schema 以及数据。举个例子，表原本包含一个类型为 integer 的列 “Foo”，而新的 schema 需要改成 string 类型，那么所有的 Parquet 数据文件都需要覆盖重写。包括以下步骤：

- 删除这个列
- 修改列的数据类型

- 修改列名，仅用大小写区分（例如 “Foo” 和 “foo”）

最后，在 Spark 3.0 中，支持了显式 DDL（通过 ALTER TABLE 方式），允许用户能够对 schema 执行以下操作：

- 添加列
- 修改列注释
- 设置表的属性来定义表的行为，例如设置事务日志的保留时间

## Schema 演变有何作用？

Schema 演变可以用来显式地修改表的 schema（而不是意外添加了并不想要的列）。这提供了一种简单的方式来迁移 schema，因为它能自动添加上正确的列名和数据类型，而不需要进行显式的定义。

## 总结

Schema 约束能够拒绝与表不兼容的任何的新的列或者 schema 的改动。通过设置严格的限制，数据工程师们可以完全信任他们的数据，从而能够作出更好的商业决策。

另一方面，schema 演变则对 schema 约束进行了补充，使得一些期望的 schema 变更能够自动地生效。毕竟，添加一个新的列本就不应该是一件困难的事情。

Schema 约束和 Schema 演变相互补益，合理地结合起来使用将能方便地管理好数据，避免脏数据污染，保证数据的完整可靠。

原文链接：<https://databricks.com/blog/2019/09/24/diving-into-delta-lake-schema-enforcement-evolution.html>

# 如何用事务日志优雅地解决并发读写

简介：事务日志（Transaction log）是理解 Delta Lake 的一个关键点，很多 Delta Lake 的重要特性都是基于事务日志实现的，包括 ACID 事务性、可扩展元数据处理、时间回溯等等。本文将探讨什么是事务日志，如何在文件层面实现，以及怎样优雅地解决并发读写的问题。

编译：辰山，阿里巴巴计算平台事业部 EMR 高级开发工程师，目前从事大数据存储方面的开发和优化工作

事务日志（Transaction log）是理解 Delta Lake 的一个关键点，很多 Delta Lake 的重要特性都是基于事务日志实现的，包括 ACID 事务性、可扩展元数据处理、时间回溯等等。本文将探讨什么是事务日志，如何在文件层面实现，以及怎样优雅地解决并发读写的问题。

## 什么是事务日志？

Delta Lake 的事务日志（简称 DeltaLog）是一种有序记录集，按序记录了 Delta Lake 表从生成伊始的所有事务操作。

## 事务日志有何作用？

### 单一信息源

Delta Lake 基于 Apache Spark 构建，用来支持多用户同时读写同一数据表。事务日志作为单一信息源——跟踪记录了用户所有的表操作，从而为用户提供了在任意时刻准确的数据视图。

当用户首次访问 Delta Lake 的表，或者对一张已打开的表提交新的查询但表中的数据在上一次访问之后已发生变化时，Spark 将会检查事务日志来确定该表经历了哪些事务操作，并将更新结果反馈给用户。这样的流程保证了用户所看到的数据版本永远保持与主分支一致，不会对同一个表产生有冲突的修改。

## Delta Lake 原子性实现

原子性，作为 ACID 四个特性之一，保证了对数据湖的操作（例如 INSERT 或者 UPDATE）或者全部完成，或者全部不完成。如果没有原子性保证，那么很容易因为硬件或软件的错误导致表中的数据被部分修改，从而导致数据错乱。

事务日志提供了一种机制来保证 Delta Lake 的原子性。任何操作只要没有记录在事务日志中，都



会被认为没有发生过。事务操作只有在完全执行成功后才会被记录到事务日志中，并且将事务日志作为单一信息源，这两者保证了数据的可靠性，保证用户可以安心处理 PB 级的数据。

## 事务日志如何工作？

### 将事务分解为原子提交

每当用户提交一个修改表的操作时（例如 INSERT, UPDATE 或 DELETE），Delta Lake 将该操作分解为包括如下所示的一系列离散步骤：

- 添加文件：添加一个数据文件。
- 删除文件：删除一个数据文件。
- 更新元数据：更新表的元数据（例如修改表的名称、schema 或分区）。
- 设置事务：记录 Spark structured streaming 任务提交的 micro batch 的 ID。
- 更改协议：将事务日志切换到最新软件协议以支持新的特性。
- 提交信息：包含提交所需的信息——执行了什么操作、操作来源以及操作时间。

这些操作都会被记录在事务日志中，形成一系列原子的单元，称作提交。

例如，假设用户创建了一个事务，往表中新增一列，并且添加一些数据。Delta Lake 会将该事务分解成离散步骤，当事务完成后，将以下提交添加到事务日志中：

1. 更新元数据：更改 schema 添加新列
2. 添加文件：添加每个新的文件

### 事务日志在文件层面的实现

当用户创建一个 Delta Lake 的表时，会在 `_delta_log` 子目录下自动创建该表的事务日志。后续对表的修改操作都将被记录为有序的原子提交，写入事务日志中。每个提交都是一个 JSON 文件，序号从 `000000.json` 开始。之后的修改操作都将生成递增的文件序号，例如 `000001.json`、`000002.json`，以此类推。



举个例子，假如我们需要往数据文件 1.parquet 和 2.parquet 中添加新的记录，该事务会被自动写入到事务日志中，保存成 000000.json 文件。然后，我们又决定删除这些文件并且添加一个新的文件（3.parquet），这些操作将被记录成事务日志中的下一个新的提交 000001.json，如下图所示：

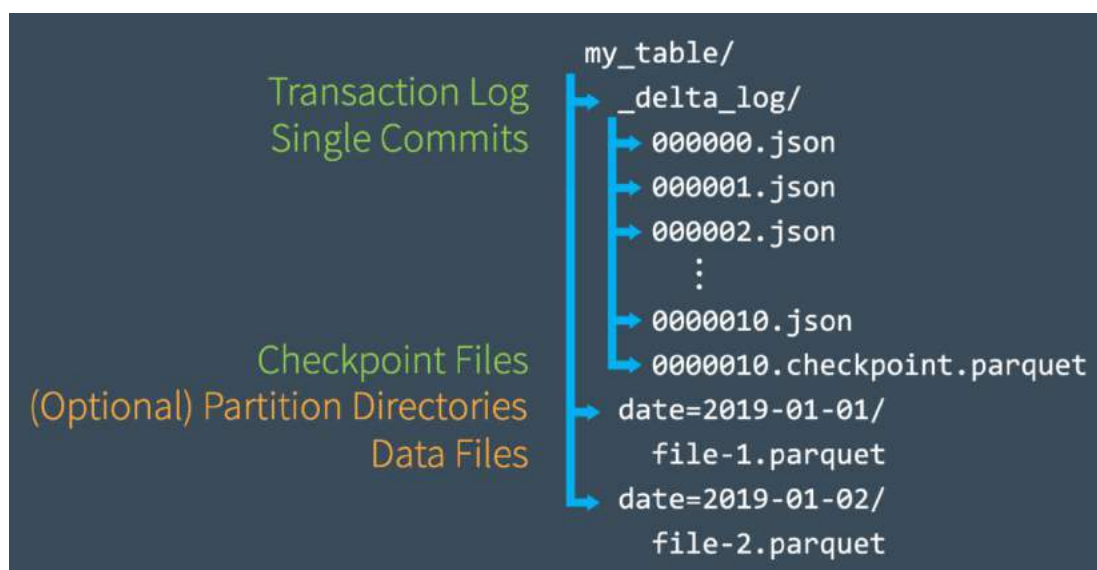


即使现在 1.parquet 和 2.parquet 已经不再是 Delta Lake 表中的数据，对它们的添加删除操作仍会记录在事务日志中，因为即便增删操作的作用最后相互抵消，但是这些操作是确实发生过的。Delta Lake 仍会保留这些原子提交，来保证当我们需要对事件进行审计，或者进行时间回溯查询表在某个历史时间点的视图时，我们都可以获得精确的结果。

另外，Spark 也不会从磁盘上删除这些文件，即使我们执行了删除了底层的数据文件的操作。用户可以通过 VACUUM 命令显式地删除不再需要的文件。

## 从 Checkpoint 文件快速重构状态

每隔 10 个提交，Delta Lake 会在 `_delta_log` 子目录下自动生成一个 Parquet 格式的 checkpoint 文件。



这些 checkpoint 文件保存了表在该时间点上的所有状态，而原生 Parquet 格式对 Spark 读取也比较友好和高效。换句话说，checkpoint 文件给 Spark 提供了一种捷径来重构表状态，避免低效地处理可能上千条的 JSON 格式的小文件。

为了同步提交进度，Spark 可以执行 `listFrom` 操作查看所有事务日志的文件，快速跳转到最新的 checkpoint 文件，这样只需处理该 checkpoint 之后的 JSON 提交即可。

下面详细阐述一下该工作流程，假设我们的提交一直创建到 `000007.json`，如下图所示，Spark 同步到该提交，也就是说已经将表的最新版本缓存在内存中。同时，其他提交者添加了新的提交一直创建到 `000012.json`。

为了包含这些新的事务并且更新我们的表状态，Spark 会运行 `listFrom version 7` 操作来查看新的修改。



Spark 将会直接跳转到最新的 checkpoint 文件，而不是逐条处理所有的 JSON 文件，因为

checkpoint 文件包含了 commit #10 之前的所有表状态。现在，Spark 只需增量执行 0000011.json 和 0000012.json，来构建表的当前状态，然后将版本12缓存在内存中。通过这样的流程，Delta Lake 能够利用 Spark 来高效地维护任意时刻的表状态。

## 处理并发读写

我们已经阐述了事务日志的大致工作原理，接下来我们讨论一下如何处理并发。以上我们的示例基本覆盖了用户顺序提交事务的场景，或者说是没有冲突的场景。但如果 Delta Lake 处理并发读写会发生什么？

这个问题非常简单，由于 Delta Lake 是基于 Apache Spark 实现的，多个用户同时修改一个表完全是一种非常常见的场景，Delta Lake 使用了乐观锁来解决这个问题。

## 什么是乐观锁

乐观并发控制（又名“乐观锁”，Optimistic Concurrency Control，缩写“OCC”）是一种并发控制的方法，它假设多用户并发的事务在处理时不会彼此互相影响。乐观锁非常高效，因为在处理 PB 级大数据时，有很大概率不同用户处理的是数据的不同部分，乐观锁使得各事务能够在不产生锁的情况下处理各自影响的那部分数据。

举个例子，假设你和我在一起合作玩拼图游戏，只要我们负责拼不同的部分，比如你负责拼角，我负责拼边，那么我们完全可以同时处理我们各自负责的部分，最终能够以翻倍的速度完成拼图。只有当我们同时需要拿同一块拼图时才会发生冲突。这就是乐观锁的原理。

相对而言，有一些数据库使用了悲观锁，悲观锁假设了最坏的情况，就是说即使我们有 10,000 块拼图，也假设我们会同时拿同一块拼图，从而会造成大量的冲突情况。悲观锁规定同时只能有一个人对拼图进行操作，其他人不能同时操作拼图，这并不是一个完成拼图游戏的高效方式。

当然，即使使用乐观锁，还是会存在不同用户同时修改数据同一部分的场景，幸运的是，Delta Lake 有一套自己的协议来处理这个问题。

## 乐观处理冲突

为了提供 ACID 事务性，Delta Lake 有一套协议来规定 commit 如何排序（也就是数据库领域串行性 serializability 的概念），协议规定了如何处理同一时间点的有多个 commit。Delta Lake 通过互斥准则来处理这种场景，并且试图乐观处理冲突。协议允许 Delta Lake 实现 ACID 的隔离性准则（isolation），保证了经过多个并发提交后表的最终状态和单独顺序提交的结果是一致的。通常来说，整个流程如下所示：

1. 记录表的初始版本。
2. 记录读写操作。
3. 尝试提交。

4. 如果另一个并发的提交已经成功，检查本次读的数据是否被修改。
5. 重复以上步骤。

下图具体阐述了 Delta Lake 如何处理冲突，假设两个用户从同一个表中读取数据，然后同时去尝试添加数据。



- Delta Lake 在进行修改之前记录表的初始版本（version 0）。
- User 1 和 2 同时尝试添加新数据到表中，这样就会发生冲突，只有一个 commit 可以被接受并记录为 000001.json。
- Delta Lake 以互斥准则处理冲突，也就是说只有一个用户能够提交 000001.json，这里假设 User 1 的 commit 被接受，而 User 2 被拒绝。
- Delta Lake 将会乐观处理此冲突，而不是直接给 User 2 返回异常。检查是否有新的 commit 提交到该表，后台更新这些修改，然后基于更新后的表重试 User 2 的 commit（没有任何数

据处理），最终成功生成 000002.json 的提交。

在大部分情况下，这种重试能够在后台无感知的完成，但也存在一些情况 Delta Lake 无法通过这种重试成功完成（例如 User 1 和 2 都同时删除同一个文件），在这种情况下将会返回异常给用户。

最后提一点，由于 Delta Lake 表的所有事务都直接保存在磁盘上，因此整个过程满足 ACID 的持久性（durability）特性，也就是说能够容忍诸如系统崩溃等错误情况。

## 其他应用场景

### 时间回溯

每张表的状态都是由记录在事务日志中的所有 commit 所决定的，事务日志相当于提供了每一步操作的历史记录，详细记录了表从初始状态到当前状态的所有操作步骤。因此，我们可以通过遍历

表从初始状态到某个时间点的所有 commit，来重构出表在任意时间点的状态。这个强大的功能就是时间回溯，或者叫做数据版本控制。更多关于时间回溯的说明，可以参考 [Introducing Delta Time Travel for Large Scale Data Lakes](#)。

## 数据血缘和调试

事务日志确切地记录了 Delta Lake 表的所有修改，因此它能提供可信的数据血缘，这对治理、审计和合规目的很有用处。它也可以用来跟踪一些无意或者有错误的修改，从而能够回退到期望的版本。用户可以执行 DESCRIBE HISTORY 来查看指定修改附近的元数据。

## 总结

本文详细探讨了 Delta Lake 的事务日志，主要包含了以下几点：

- 事务日志是什么，怎样构建，以及提交（commit）如何在文件层面进行存储。
- 事务日志怎么作为单一信息源，实现 Delta Lake 的原子性特性。
- Delta Lake 如何计算表的状态，包括如何从最新 checkpoint 构建当前状态，以及怎样处理小文件问题。
- 通过 Apache Spark 来处理大规模元数据。
- 通过乐观锁实现并发读写。
- Delta Lake 如何通过互斥准则保证 commit 被正确排序，以及在冲突时如何解决。

原文链接：<https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html>



# 使用 Jupyter Notebook 运行 Delta Lake 入门教程

简介：因为官方教程是基于商业软件Databricks Community Edition 构建，虽然教程中使用的软件特性都是开源 Delta Lake 版本所具备的，但是考虑到国内的网络环境，注册和使用 Databricks Community Edition 门槛较高。所以本文尝试基于开源的 Jupiter Notebook 重新构建这个教程。

**作者：**吴威，花名无谓，阿里巴巴高级技术专家，2008年加入阿里巴巴集团，先后在B2B和阿里云工作，一直从事大数据和分布式计算相关研究，作为主要开发和运维。人员经历了阿里内部大数据集群的上线和发展壮大，现在阿里云EMR团队，负责Spark、Hadoop等计算引擎研发。

本文的例子来自 Delta Lake 官方教程。因为官方教程是基于商业软件 Databricks Community Edition 构建，虽然教程中使用的软件特性都是开源 Delta Lake 版本所具备的，但是考虑到国内的网络环境，注册和使用 Databricks Community Edition 门槛较高。所以本文尝试基于开源的 Jupiter Notebook 重新构建这个教程。

## 准备一个环境安装 Spark 和 jupyter

本文基于 Linux 构建开发环境，同时使用的软件比如 conda、jupyter以及 pyspark 等都可以在 **Windows** 和 **MacOS** 上找到，理论上来说也完全可以在这两个系统上完成此教程。

假设系统已经安装 anaconda 或 miniconda，我们使用 conda 来构建开发环境，可以非常方便的安装 pyspark 和 jupyter notebook。

```
conda create --name spark
conda activate spark

conda install pyspark
conda install -c conda-forge jupyterlab
```

## 环境变量设置

我们在设置一些环境变量之后，就可以使用pyspark 命令来创建 jupyter notebook 服务

```
export SPARK_HOME=$HOME/miniconda3/envs/spark/lib/python3.7/site-  
packages/pyspark  
export PYSPARK_DRIVER_PYTHON=jupyter  
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
```

启动服务（注意这里的参数里指定了 Delta Lake 的 package，Spark 会帮忙自动下载依赖）：

```
pyspark --packages io.delta:delta-core_2.11:0.5.0
```

接下去所有代码在 notebook 里运行下载需要 parquet 文件

## 下载需要 parquet 文件

```
%%bash  
rm -fr /tmp/delta_demo  
mkdir -p /tmp/delta_demo/loans/  
wget -O /tmp/delta_demo/loans/SAISEU19-loan-risks.snappy.parquet  
https://pages.databricks.com/rs/094-YMS-629/images/SAISEU19-loan-  
risks.snappy.parquet  
ls -al /tmp/delta_demo/loans/
```

## Delta Lake的批流处理

在这里我们进入正题，开始介绍 Delta Lake 的批流处理能力。

首先，我们通过批处理的形式创建一张 Delta Lake 表，数据来自前面我们下载的 parquet 文件，可以和方便的把一张 parquet 表转换为 Delta Lake 表：

```
import os

import shutil

from pyspark.sql.functions import *


delta_path = "/tmp/delta_demo/loans_delta"


# Delete a new delta table with the parquet file
if os.path.exists(delta_path):
    print("Deleting path " + delta_path)
    shutil.rmtree(delta_path)


# Create a new delta table with the parquet file
spark.read.format("parquet").load("/tmp/delta_demo/loans") \
    .write.format("delta").save(delta_path)

print("Created a Delta table at " + delta_path)
```

我来查一下这张表，数据量是否正确：

```
# Create a view on the table called loans_delta
spark.read.format("delta").load(delta_path).createOrReplaceTempView("loans_delta")
print("Defined view 'loans_delta'")

spark.sql("select count(*) from loans_delta").show()

Defined view 'loans_delta'
+-----+
|count(1)|
+-----+
|  14705|
+-----+
```

接下去我们会使用Spark Streaming流式写入这张 Delta Lake 表，同时展示 Delta Lake 的 Schema enforcement 能力（本文省略了流式写 Parquet 表的演示部分，那部分指出了 parquet 文件的不足，比如无法强制指定 Schema）

```
import random
from pyspark.sql.functions import *
from pyspark.sql.types import *

def random_checkpoint_dir():
    return "/tmp/delta_demo/chkpt/%s" % str(random.randint(0, 10000))

# User-defined function to generate random state

states = ["CA", "TX", "NY", "IA"]
```

```
@udf(returnType=StringType())
def random_state():
    return str(random.choice(states))

# Generate a stream of randomly generated load data and append to the delta table
def generate_and_append_data_stream_fixed(table_format, table_path):

    stream_data = spark.readStream.format("rate").option("rowsPerSecond", 50).load() \
        .withColumn("loan_id", 10000 + col("value")) \
        .withColumn("funded_amnt", (rand() * 5000 + 5000).cast("integer")) \
        .withColumn("paid_amnt", col("funded_amnt") - (rand() * 2000)) \
        .withColumn("addr_state", random_state()) \
        .select("loan_id", "funded_amnt", "paid_amnt", "addr_state") # ***** FIXED
    THE SCHEMA OF THE GENERATED DATA *****

    query = stream_data.writeStream \
        .format(table_format) \
        .option("checkpointLocation", random_checkpoint_dir()) \
        .trigger(processingTime="10 seconds") \
        .start(table_path)

    return query
```

启动两个流式作业：

```
stream_query_1 = generate_and_append_data_stream_fixed(table_format = "delta",
table_path = delta_path)

stream_query_2 = generate_and_append_data_stream_fixed(table_format = "delta",
table_path = delta_path)
```

因为 Delta Lake 的乐观锁机制，多个流可以同时写入一张表，并保证数据的完整性。通过批处理的方式来查询一下当前表中的数据量，我们发现数据被插入了：

```
spark.sql("select count(*) from loans_delta").show()

+-----+
|count(1)|
+-----+
| 17605|
+-----+
```

接下去我们停止所有流的写入，接下去会展示 Delta Lake 的其他特性

```
# Function to stop all streaming queries

def stop_all_streams():

    # Stop all the streams

    print("Stopping all streams")

    for s in spark.streams.active:

        s.stop()

    print("Stopped all streams")

    print("Deleting checkpoints")

    shutil.rmtree("/tmp/delta_demo/chkpt/", True)

    print("Deleted checkpoints")

stop_all_streams()

Schema evolution (Schema演化)
```



Delta Lake 支持Schema演化，也就是说我们可以增加或改变表字段。接下去的批处理 SQL 会增加一些数据，同时这些数据比之前的多了一个“closed”字段。我们将新的 DF 配置参数 mergeSchema 为 true 来显示指明 Delta Lake 表 Schema 的演化：

```
cols = ['loan_id', 'funded_amnt', 'paid_amnt', 'addr_state', 'closed']

items = [
    (1111111, 1000, 1000.0, 'TX', True),
    (2222222, 2000, 0.0, 'CA', False)
]

loan_updates = spark.createDataFrame(items, cols) \
    .withColumn("funded_amnt", col("funded_amnt").cast("int"))

loan_updates.write.format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \
    .save(delta_path)
```

来看一下插入新数据之后的表内容，新增加了 closed 字段，之前的老数据行这个字段默认为 null。

```
spark.read.format("delta").load(delta_path).show()

+-----+-----+-----+-----+-----+
|loan_id|funded_amnt|paid_amnt|addr_state|closed|
+-----+-----+-----+-----+-----+
| 0| 1000| 182.22| CA| null|
| 1| 1000| 361.19| WA| null|
| 2| 1000| 176.26| TX| null|
| 3| 1000| 1000.0| OK| null|
| 4| 1000| 249.98| PA| null|
| 5| 1000| 408.6| CA| null|
| 6| 1000| 1000.0| MD| null|
| 7| 1000| 168.81| OH| null|
| 8| 1000| 193.64| TX| null|
| 9| 1000| 218.83| CT| null|
| 10| 1000| 322.37| NJ| null|
| 11| 1000| 400.61| NY| null|
```

```
| 12| 1000| 1000.0| FL| null|
| 13| 1000| 165.88| NJ| null|
| 14| 1000| 190.6| TX| null|
| 15| 1000| 1000.0| OH| null|
| 16| 1000| 213.72| MI| null|
| 17| 1000| 188.89| MI| null|
| 18| 1000| 237.41| CA| null|
| 19| 1000| 203.85| CA| null|
```

```
+-----+-----+-----+-----+-----+
```

only showing top 20 rows

新的数据行具有 closed 字段：

```
spark.read.format("delta").load(delta_path).filter(col("closed") == True).show()
```

```
+-----+-----+-----+-----+-----+
|loan_id|funded_amnt|paid_amnt|addr_state|closed|
+-----+-----+-----+-----+-----+
|1111111| 1000| 1000.0| TX| true|
+-----+-----+-----+-----+-----+
```

## Delta Lake 表的删除操作

除了常规的插入操作，Delta Lake 还支持 update 和 delete 等功能，可以更新表格内容。下面展示删除操作，我们希望删除表格中贷款已经被完全还清的记录。下面几条命令可以简单和清晰的展示删除过程。

首先，我们看看符合条件的记录有多少条：

```
spark.sql("SELECT COUNT(*) FROM loans_delta WHERE funded_amnt =  
paid_amnt").show()
```

```
+-----+  
|count(1)|  
+-----+  
|   5134|  
+-----+
```

然后，我们执行一个 delete 命令：

```
from delta.tables import *  
  
deltaTable = DeltaTable.forPath(spark, delta_path)  
deltaTable.delete("funded_amnt = paid_amnt")
```

最后，我们看一下删除后的结果，发现符合条件的记录都已被删除：

```
spark.sql("SELECT COUNT(*) FROM loans_delta WHERE funded_amnt =  
paid_amnt").show()  
  
+-----+  
|count(1)|  
+-----+  
|      0|  
+-----+
```

## 版本历史和回溯

Delta Lake 还具有很大历史版本记录和回溯功能。history()方法清晰的展示了刚才那张表的修改记录，包括最后一次 Delete 操作。

```
deltaTable.history().show()
```

```
+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+
|version|      timestamp|userId|userName|      operation| operationParameters|
job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|
+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+
|   10|2020-02-22 22:14:06| null|   null|      DELETE|[predicate -> ["(...|null|   null|
null|      9|      null|   false|
|    9|2020-02-22 22:13:57| null|   null|      WRITE|[mode -> Append, ...|null|   null|
null|      8|      null|   true|
|    8|2020-02-22 22:13:52| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      6|      null|   true|
|    7|2020-02-22 22:13:50| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      6|      null|   true|
|    6|2020-02-22 22:13:42| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      4|      null|   true|
|    5|2020-02-22 22:13:40| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      4|      null|   true|
|    4|2020-02-22 22:13:32| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      2|      null|   true|
|    3|2020-02-22 22:13:30| null|   null|STREAMING UPDATE|[outputMode ->
Ap...|null|   null|   null|      2|      null|   true|
```

```
| 2|2020-02-22 22:13:22| null| null|STREAMING UPDATE|[outputMode -> Ap...|null|
null| null| 1| null| true|
| 1|2020-02-22 22:13:20| null| null|STREAMING UPDATE|[outputMode -> Ap...|null|
null| null| 0| null| true|
| 0|2020-02-22 22:13:18| null| null| WRITE|[mode -> ErrorIfE...|null| null|
null| null| null| true|
+-----+-----+-----+-----+-----+-----+-----+-----+
-- +-----+-----+-----+-----+-----+-----+-----+-----+
```

如果我们希望看一下刚才删除操作前的数据表状态，可以很方便的回溯到前一个快照点，并进行再次查询（我们可以看到被删除的记录又出现了）。

```
previousVersion = deltaTable.history(1).select("version").collect()[0][0] - 1

spark.read.format("delta") \
  .option("versionAsOf", previousVersion) \
  .load(delta_path) \
  .createOrReplaceTempView("loans_delta_pre_delete") \

spark.sql("SELECT COUNT(*) FROM loans_delta_pre_delete WHERE funded_amnt =
paid_amnt").show()

+-----+
|count(1)|
+-----+
| 5134|
+-----+
```

## 结论

本文通过 jupyter notebook 工具演示了 Delta Lake 的官方教程，你可以在原文链接末尾下载到完整的notebook 文件。

# Spark SQL 性能优化

## Apache Spark 3.0中的SQL性能改进概览

简介： 阿里巴巴高级技术专家李呈祥为大家带来Apache Spark 3.0中的SQL性能改进概览的介绍。以下由Spark+AI Summit中文精华版峰会的精彩内容整理。

今天主要跟大家分享一下spark 3.0在SQL方向上的一些优化工作。从spark 2.4开始，大概有超过一年半的时间。对于一个比较活跃的开源项目来说，这个时间是非常长的。所以里面包含了大量的这种功能增强，性能优化，等各方面的新的feature在里面。大概超过50%的相关的issue都是和SQL相关的。在SQL这个方向上主要做的工作，大概分成四个方面。第一方面是工具类的。就是说基于spark的一个开发者怎么去和spark交互，提供一些更多的工具。第二个是dynamic optimization。简单来说就是运行时的优化。在这里面，包含了几个重大的性能改进。第三个是在spark的catalyst优化器方面有很多新的改进。第四个是基础依赖的更新。主要在语言层面引入了一些新的支持和依赖。

### Spark 3.0

#### Four Categories of Major Changes for SQL

##### Interactions with developers



##### Dynamic optimizations



##### Catalyst improvements



##### Infrastructure updates



5

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

IBM

Spark 3.0是一个时间跨度非常长的release，包含了非常多的社区的工作。统计下来有接近3400多个issue在spark 3.0里面进行了处理。针对这么多的issue，我们用spark 3.0的时候，需要考虑有哪些东西对于实际的生产环境可能有好处，有哪些新的特性。

### Many Many Changes for 1.5 years

Version 3.0.0

UNRELEASED

Start date not set

Release date not set

Release Notes

0 Warnings

3464 Issues in version

3463 Issues done

0 Issues in progress

1 Issues to do

**Hard to understand what's new  
due to many many changes**

**This session guides you to understand  
what's new for SQL performance**

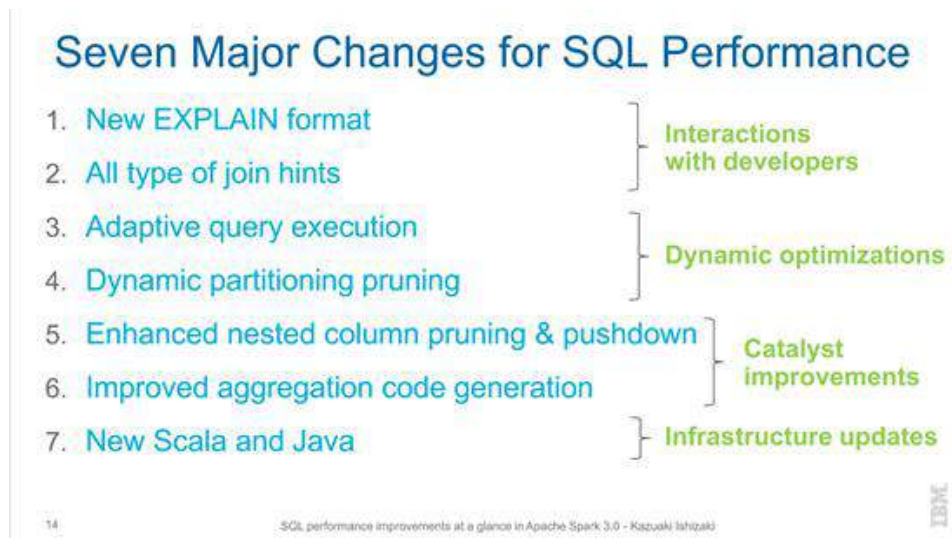
12

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

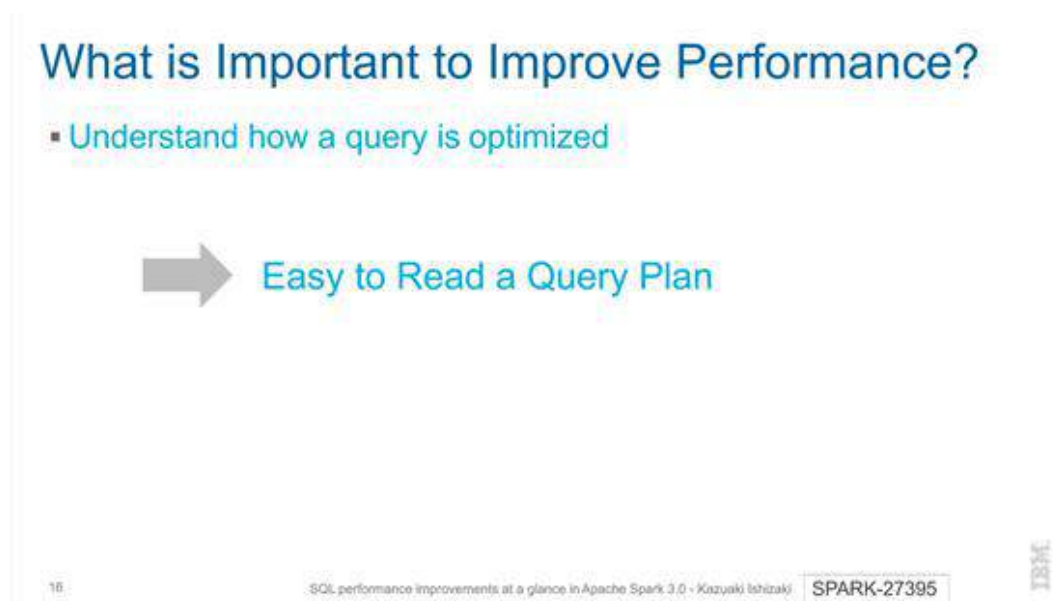
IBM



总结下来，大概可以把在SQL方向上的这种大的改动分成七个部分，分属于上文中提到的四个类别。



第一部分是new explain format。当我们想去改进，去优化一个spark SQL的性能的时候，首先需要去了解SQL的查询计划大概是一个什么样子，有针对性的去进行这种SQL的重写，或其他的一些改进。前提就是我的查询计划可读性比较强，是很容易去看的。



对于之前2.4的版本，可以通过explain SQL去展示。只不过是这种展示的方式看起来繁杂一点。我们可以看到针对于SQL，这么一个物理查询计划，是一个树状的结构。也是可以去看，但是可读性相对来说不够好。

## Not Easy to Read a Query Plan on Spark 2.4

- Not easy to understand how a query is optimized

**Output is too long!!**

```
scala> val query = "SELECT key, Max(val) FROM temp WHERE key > 0 GROUP BY key HAVING max(val) > 0"
scala> sql("EXPLAIN " + query).show(false)
```

```
!== Physical Plan ==
*(2) Project [key#2, max(val)#15]
+- *(2) Filter (isnotnull(max(val#3)#18) AND(max(val#3)#18 > 0))
   +- *(2) HashAggregate(keys=[key#2], functions=[max(val#3)], output=[key#2, max(val)#15,
      max(val#3)#18])
      +- Exchange hashpartitioning(key#2, 200)
         +- *(1) HashAggregate(keys=[key#2], functions=[partial_max(val#3)], output=[key#2,
            max#21])
            +- *(1) Project [key#2, val#3]
               +- *(1) Filter (isnotnull(key#2) AND(key#2 > 0))
                  +- *(1) FileScan parquet default.temp[key#2,val#3] Batched: true,
                     DataFilters: [isnotnull(key#2), (key#2 > 0)], Format: Parquet, Location:
                     InMemoryFileIndex[file:/user/hive/warehouse/temp], PartitionFilters: [], PushedFilters:
                     [IsNotNull(key), GreaterThan(key,0)], ReadSchema: struct<key:int,val:int>
```

From #24750

18

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-27395

在3.0里面，针对查询计划的这种展示进行了一定的优化，以简要的格式展示。根据节点的编号，可以找到对应的更详细的信息。而且对于每一个节点展示的信息也做了一些归类 and 整理，整理成 input, output, condition 等。通过这种方式，用户可以更加清晰的看到整个的查询计划。

## Easy to Read a Query Plan on Spark 3.0

- Show a query in a terse format with detail information

```
scala> sql("EXPLAIN FORMATTED" + query).show(false)
```

```
!== Physical Plan ==
Project (8)
+- Filter (7)
   +- HashAggregate (6)
      +- Exchange (5)
         +- HashAggregate (4)
            +- Project (3)
               +- Filter (2)
                  +- Scan parquet default.temp1 (1)
```

```
(1) Scan parquet default.temp [codegen id : 1]
Output: [key#2, val#3]
```

```
(2) Filter [codegen id : 1]
Input: [key#2, val#3]
Condition: (isnotnull(key#2) AND(key#2 > 0))
```

```
(3) Project [codegen id : 1]
Output: [key#2, val#3]
Input: [key#2, val#3]
```

```
(4) HashAggregate [codegen id : 1]
Input: [key#2, val#3]
```

```
(5) Exchange
Input: [key#2, max#11]
```

```
(6) HashAggregate [codegen id : 2]
Input: [key#2, max#11]
```

```
(7) Filter [codegen id : 2]
Input: [key#2, max(val)#5, max(val#3)#8]
Condition: (isnotnull(max(val#3)#8) AND
(max(val#3)#8 > 0))
```

```
(8) Project [codegen id : 2]
Output: [key#2, max(val)#5]
Input: [key#2, max(val)#5, max(val#3)#8]
```

19

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-27395

第二部分是 all type of join hints。在 spark 2.4 只支持 broadcast。而 spark 3.0 除了支持 broadcast，还支持 sort merge, shuffle hash 和 cartesian。

## All of Join Type Can be Used for a Hint

Join type	2.4	3.0
Broadcast	BROADCAST	BROADCAST
Sort Merge	X	SHUFFLE MERGE
Shuffle Hash	X	SHUFFLE_HASH
Cartesian	X	SHUFFLE_REPLICATE_NL

### Examples

```
SELECT /*+ SHUFFLE_HASH(a, b) */ * FROM a, b
WHERE a.a1 = b.b1
```

```
val shuffleHashJoin = aDF.hint("shuffle_hash")
    .join(bDF, aDF("a1") === bDF("b1"))
```

22

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-27225

第三部分是adaptive query execution。社区为什么要去做它，最主要的原因就是说，对于一些查询计划，在运行时能够拿到更准确的数据统计信息，可以选择最优的这种计划，对数据进行处理，从而提升spark处理数据的性能。主要包括三种场景。第一种是调整reducer的数量，从而避免额外的内存和IO的开销。第二种是说，选择最合适的join的策略。第三种是说，针对倾斜数据，在join

的时候提供更好的处理方式。上述场景都是自动的，根据运行时的情况，自动地收集相关的信息，然后去做判断。

## Automatically Tune Parameters for Join and Reduce

- Three parameters by using runtime statistics information (e.g. data size)
  1. Set the number of reducers to avoid wasting memory and I/O resource
  2. Select better join strategy to improve performance
  3. Optimize skewed join to avoid imbalance workload

**Without manual tuning properties run-by-run**

**Yield 8x performance improvement of Q77 in TPC-DS**

Source: Adaptive Query Execution: Speeding Up Spark SQL at Runtime

24

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

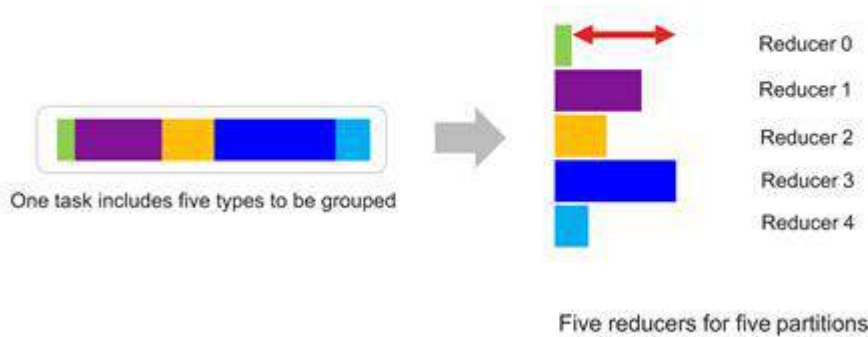
SPARK-23128 &amp; 30864

怎么去动态的调整reducer的数量。在spark 2.4，默认指定partition数量，每一个partition经过shuffle之后，对应的要处理的数据的大小可能是不一样的。这是由数据本身的特性来决定的，它的分布可能本来就是不均衡的。



## Used Preset Number of Reduces on Spark 2.4

- The number of reducers is set based on the property `spark.sql.shuffle.partitions` (default: 200)



25

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

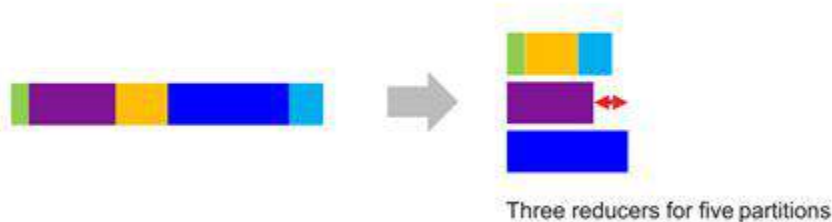
SPARK-23128 &amp; 30864

IBM

在spark 3.0中，在shuffle的时候，每一个partition有不同的数据量大小，需要把小的partition数据进行合并，给同一个reducer去处理，从而使得每一个reducer它所处理的数据量大小是相近的。

## Tune the Number of Reducers on Spark 3.0

- Select the number of reducers to meet the given target partition size at each reducer



```
spark.sql.adaptive.enabled -> true (false in Spark 3.0)
spark.sql.adaptive.coalescePartitions.enabled -> true (false in Spark 3.0)
```

26

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

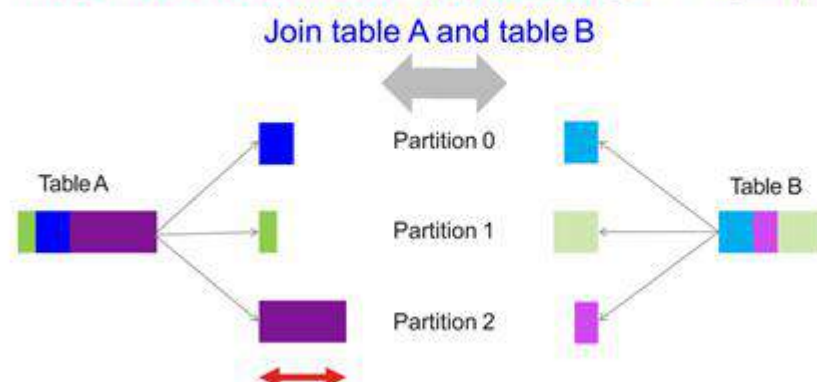
SPARK-23128 &amp; 30864

IBM

针对有数据倾斜的这种join，在spark 2.4中带来的主要的问题就是说，在处理最大的partition时，要花费很长的时间，影响整个join。

## Skewed Join is Slow on Spark 2.4

- The join time is dominated by processing the largest partition



29

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

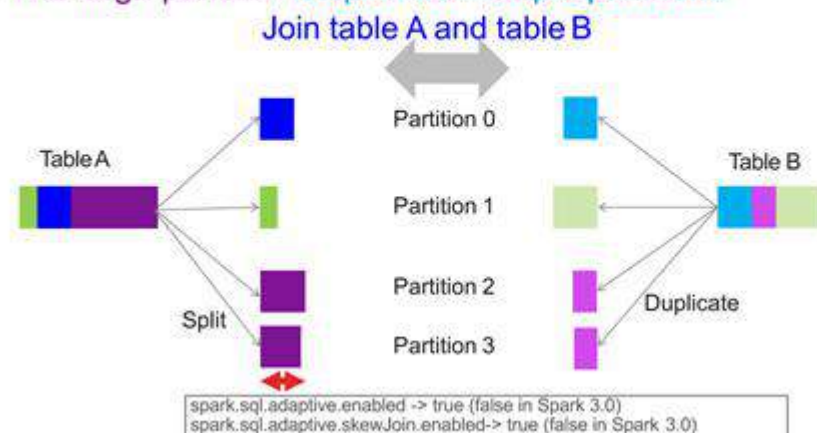
SPARK-23128 &amp; 30864

IBM

在spark 3.0中，有数据倾斜的join，比在spark 2.4中更快。如图所示，对于表A和表B，我把大表的数据做切分，小表的数据做全量的分发。第一个，满足join的语义要求。第二个，在倾斜的这些key上面，它是被切成多分，然后在多个task里面去处理。

## Skewed Join is Faster on Spark 3.0

- The large partition is split into multiple partitions



30

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-23128 &amp; 30864

IBM

第四部分是dynamic partitioning pruning。在join操作中，要避免读取不必要的partition。而dynamic filter能够避免读取不必要的partition。

## Dynamic Partitioning Pruning

- Avoid to read unnecessary partitions in a join operation
  - By using results of filter operations in another table
- Dynamic filter can avoid to read unnecessary partition

Yield 85x performance improvement of Q98 in TPC-DS 10TB

Source: Dynamic Partition Pruning in Apache Spark

32

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

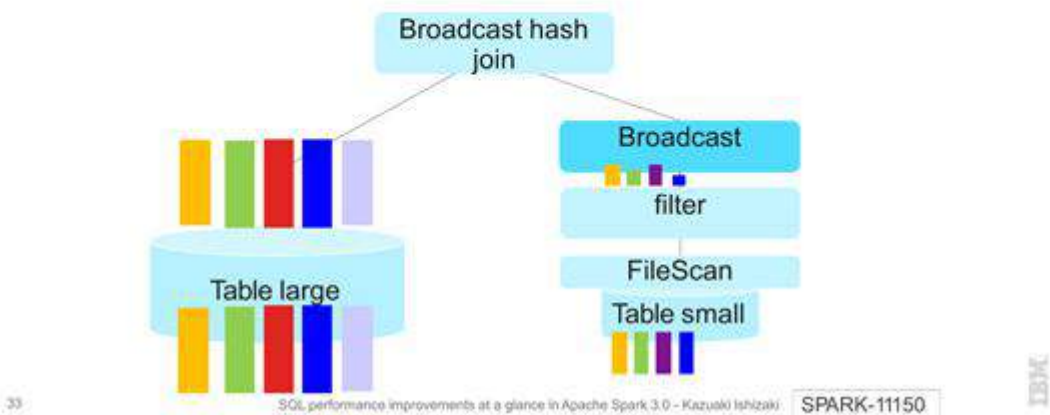
SPARK-11150

IBM

如下图所示，在spark 2.4中，大表中的所有数据都被读取。

## Naïve Broadcast Hash Join on Spark 2.4

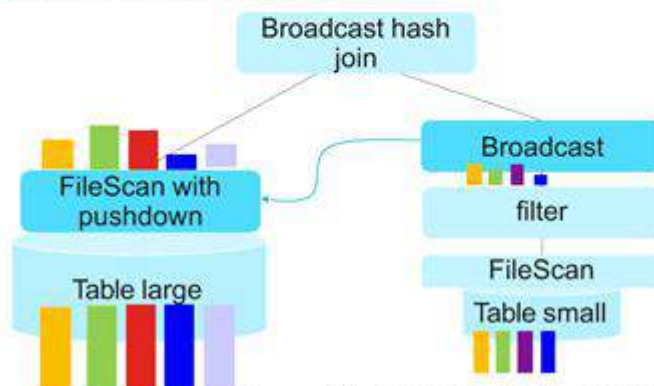
- All of the data in Large table is read



而在spark 3.0中，通过pushdown with dynamic filter，能够减少大表中需要被读取的数据量。

## Prune Data with Dynamic Filter on Spark 3.0

- Large table can reduce the amount of data to be read using pushdown with dynamic filter



34

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-11150

IBM

如下图所示，是一个dynamic partitioning pruning的例子。

## Example of Dynamic Partitioning Pruning

```
scala> spark.range(7777).selectExpr("id", "id AS key").write.partitionBy("key").saveAsTable("tableLarge")
scala> spark.range(77).selectExpr("id", "id AS key").write.partitionBy("key").saveAsTable("tableSmall")
scala> val query = "SELECT * FROM tableLarge JOIN tableSmall ON tableLarge.key = tableSmall.key AND tableSmall.id < 3"
scala> sql("EXPLAIN FORMATTED" + query).show(false)
```

```
== Physical Plan ==
* BroadcastHashJoin Inner BuildRight (8)
:- * ColumnarToRow (2)
:- +- Scan parquet default.tablelarge (1)
+- BroadcastExchange (7)
+- * Project (6)
+- * Filter (5)
+- * ColumnarToRow (4)
+- Scan parquet default.tablesmall (3)
```

(1) Scan parquet default.tablelarge

Output [2]: [id#19L, key#20L]

Batched: true

Location: InMemoryFileIndex [file:/home/ishizaki/Spark/300RC1/spark-3.0.0-bin-hadoop2.7/spark-warehouse/tablelarge/key=0, ... 7776 entries]

**PartitionFilters:** [isNotNull(key#20L), dynamicpruningexpression(key#20L IN dynamicpruning#56)]

ReadSchema: struct<id:bigint>

Source: Quick Overview of Upcoming Spark 3.0

35

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-11150

IBM

第五部分是Enhanced nested column pruning & pushdown，是针对于这种嵌套的数据结构的支持。在spark 2.4里面，其实已经提供了部分的这种支持。如下图所示的表里面，有column 1和column 2，而后者是一个嵌套的数据结构，它里面有两个字段。比如说，我查询的时候只查了column 2里面的第1个字段。去访问这个数据的时候，我只需要把column 2的第1个字段拿出来就行了，而不需要把整个column 2都拿出来。但是在spark 2.4里面它的支持是有限的。就是说，只能穿透有限的几个算子，比如说LIMIT这种算子，对于其他的一些算子是没办法的。

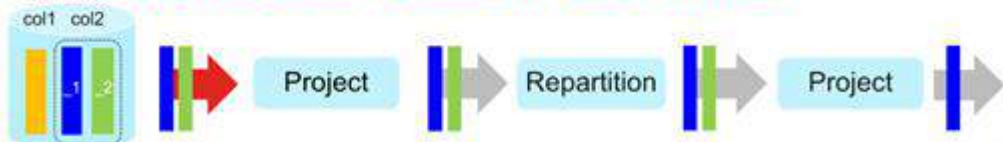


## Limited Nested Column Pruning on Spark 2.4

- Column pruning that read only necessary column for Parquet
  - Can be applied to limited operations (e.g. LIMIT)



- Cannot be applied other operations (e.g. REPARTITION)



38

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-25603 &amp; 25556

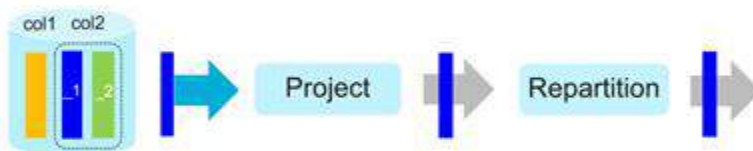
Source: #23964

IBM

而在spark 3.0里面，对这一块进行了进一步的优化，能够支持把column pruning推到穿透所有的算子。

## Generalize Nested Column Pruning on Spark 3.0

- Nested column pruning can be applied to all operators
  - e.g. LIMITS, REPARTITION, ...



39

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-25603 &amp; 25556

Source: #23964

IBM

另外一种场景，就是说filter过滤的条件是根据嵌套字段里面的某一个子字段去做过滤，是不是支持把过滤条件也推到table scan里面。在spark 2.4里面也是不能够完全支持的。

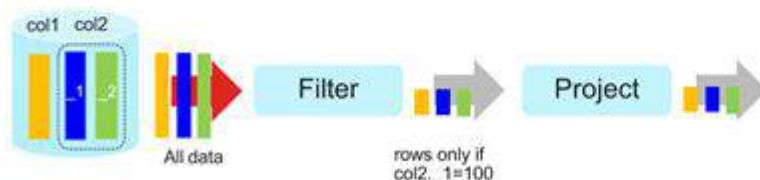
## No Nested Column Pushdown on Spark 2.4

### Parquet cannot apply predication pushdown

```
scala> spark.range(1000).map(x => (x, (x, s"$x" * 10))).toDF("col1", "col2").write.parquet("/tmp/p")
scala> spark.read.parquet("/tmp/p").filter("col2._1 = 100").explain
```

#### Spark 2.4

```
== Physical Plan ==
*(1) Project [col1#12L, col2#13]
+- *(1) Filter (isNotNull(col2#13) &&(col2#13._1 = 100))
   +- *(1) FileScan parquet [col1#12L,col2#13] ..., PushedFilters: [IsNotNull(nested)], ...
```



41

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-25603 &amp; 25556

Source: #28319

而在spark 3.0里面，针对嵌套字段的filter，也是一直可以往下推到具体访问数据的table scan里面。

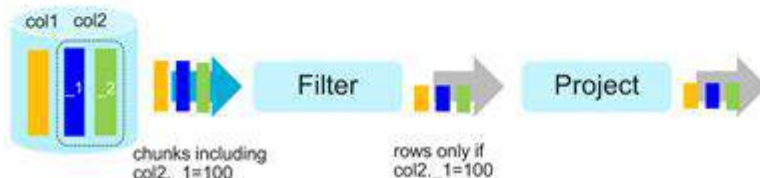
## Nested Column Pushdown on Spark 3.0

### Parquet can apply pushdown filter and can read part of columns

```
scala> spark.range(1000).map(x => (x, (x, s"$x" * 10))).toDF("col1", "col2").write.parquet("/tmp/p")
scala> spark.read.parquet("/tmp/p").filter("col2._1 = 100").explain
```

#### Spark 3.0

```
== Physical Plan ==
*(1) Project [col1#0L, col2#1]
+- *(1) Filter (isNotNull(col2#1) AND(col2#1._1 = 100))
   +- FileScan parquet [col1#0L,col2#1] ..., DataFilters: [isNotNull(col2#1), (col2#1.x = 100)],
      ..., PushedFilters: [IsNotNull(col2), EqualTo(col2._1,100)], ...
```



42

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-25603 &amp; 25556

Source: #28319

第六部分是Improved aggregation code generation，针对aggregation扩件的一个优化。

## Complex Aggregation is Slow on Spark 2.4

- A complex query is not compiled to native code

**Not good performance of Q66 in TPC-DS**

Source: #20695

44

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

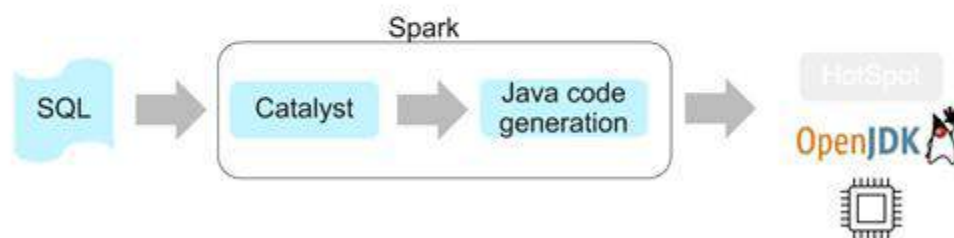
SPARK-21870

IBM

就是说，在spark里面我们去支持这种扩件，但是扩件会有一个限制。针对每个方法，如果大于8000 Java bytecode，HotSpot编译器就rollback，放弃生成native code。所以，如果你的这种SQL比较复杂，可能会没办法利用到扩件的这种特性。

## How SQL is Translated to native code

- In Spark, Catalyst translates a given query to Java code
- HotSpot compiler in OpenJDK gives up generating native code for more than **8000 Java bytecode instruction per method**



45

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-21870

IBM

在spark 3.0里面，针对这种情况做一些优化。简单来说，把一个方法拆分成多个方法，从而避免碰到8000 Java bytecode的限制。

## Making Aggregation Java Code Small

- In Spark, Catalyst translates a given query to Java code
- HotSpot compiler in OpenJDK **gives up** generating native code for more than **8000 Java bytecode instruction per method**

➔ Catalyst splits a large Java method into small ones to allow HotSpot to generate native code

47

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-21870

IBM

具体的例子如下图所示。

## Example of Small Aggregation Code

- Average function (100 rows) for 50 columns

```
scala> val numCols = 50
scala> val colExprs = (0 until numCols).map { i => s"id AS col$i" }
scala> spark.range(100).selectExpr(colExprs: _*).createOrReplaceTempView("temp")
scala> val aggExprs = (0 until numCols).map { i => s"AVG(col$i)" }
scala> val query = s"SELECT ${aggExprs.mkString(", ")} FROM temp"
scala> import org.apache.spark.sql.execution.debug._
scala> sql(query).debugCodegen()
```

Found 2 WholeStageCodegen subtrees.

== Subtree 1 / 2 (maxMethodCodeSize:3679; maxConstantPoolSize:1107(1.69% used); numInnerClasses:0) ==

...
 == Subtree 2 / 2 (maxMethodCodeSize:5581; maxConstantPoolSize:882(1.35% used); numInnerClasses:0) ==
 ...

### Disable this feature

```
scala> sql("SET spark.sql.codegen.aggregate.splitAggregateFunc.enabled=false")
scala> sql(query).debugCodegen()
```

Found 2 WholeStageCodegen subtrees.

== Subtree 1 / 2 (maxMethodCodeSize:8917; maxConstantPoolSize:957(1.46% used); numInnerClasses:0) ==

...
 == Subtree 2 / 2 (maxMethodCodeSize:9862; maxConstantPoolSize:728(1.11% used); numInnerClasses:0) ==
 ...

Source: PR #20965

49

SQL performance improvements at a glance in Apache Spark 3.0 - Kazuaki Ishizaki

SPARK-21870

IBM

第七部分是New Scala and Java，针对新的语言版本的支持。支持了新的Java 11这个版本，以及Scala 2.12版本。



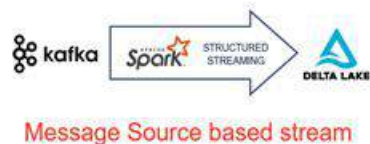
# Structured Streaming生产化实践及调优

简介：Databricks软件工程师李元健为大家带来structured streaming生产化实践及调优的介绍。内容包括输入参数，状态参数，输出参数的调优，以及部署。以下由Spark+AI Summit中文精华版峰会的精彩内容整理。

## 一、简介

通常来讲，从输入源和处理模式的角度来看，我们的streaming有多种模式。包括Message Source based stream, File Source based stream, 等等。

Suppose we have a stream set up like this



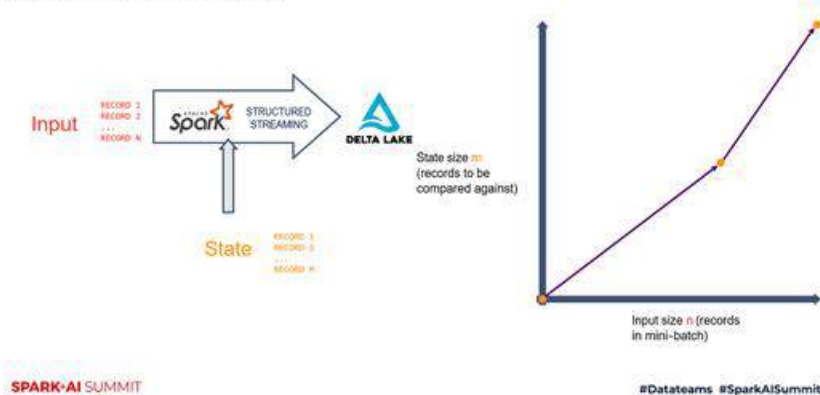
```
spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "...")
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("delta")
  .option("path", "/delta/table/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()
```

SPARK+AI SUMMIT

#Datateams #SparkAISummit

我们可以看到，针对一个streaming系统，无论在何种模式下，structured streaming的运行环境中都会有两个关键因素影响整体性能。一个是input size，一个是state size。这两个size完全指导于我们整个streaming的全部的流程，与端到端性能强相关。

## Scale dimensions



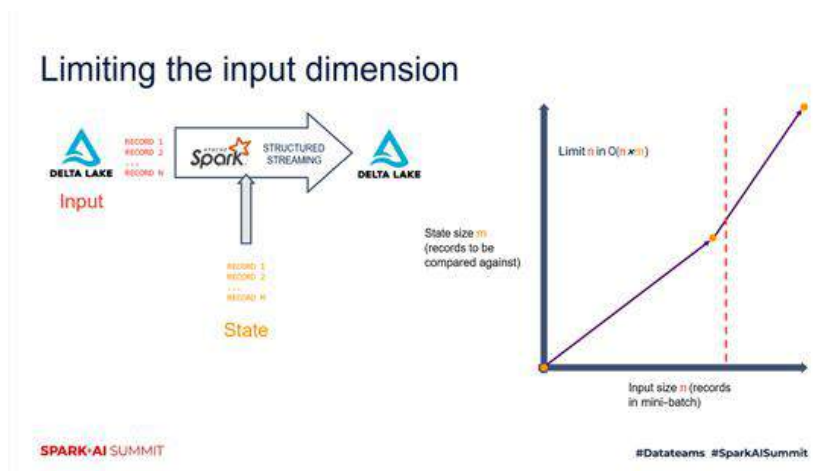
对于这样的场景，我们用下面这个例子来一步步展开，讲解和分析整个调优过程。

## Let's use this example!

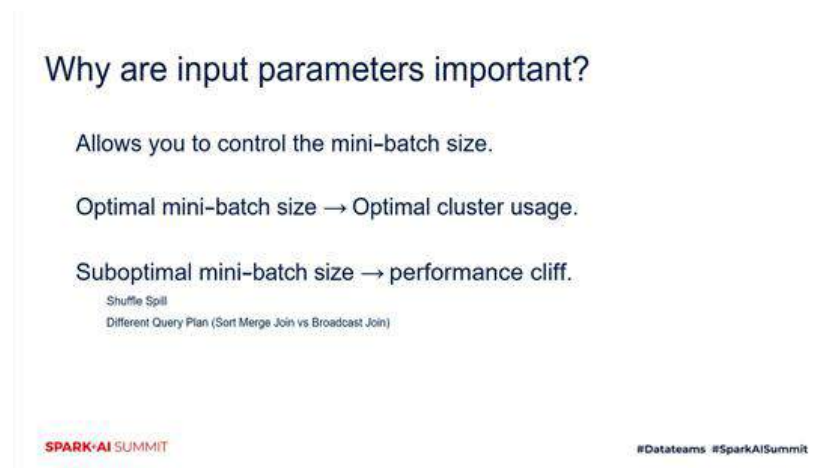


## 二、输入参数

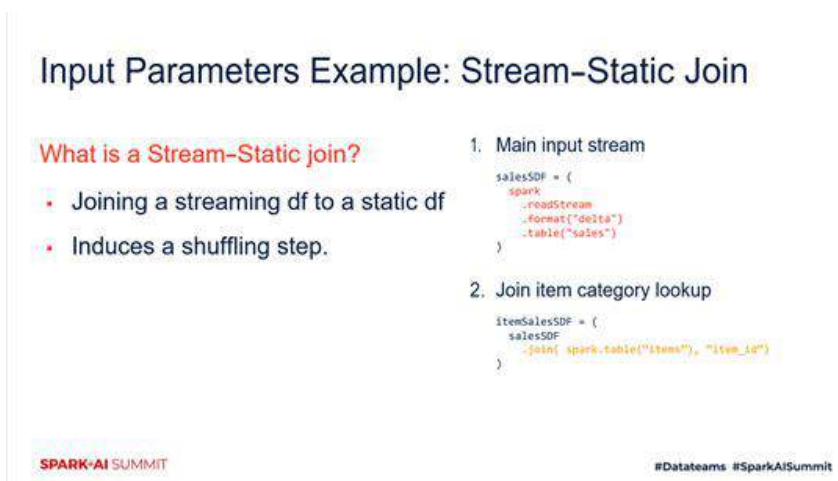
对于一个streaming系统来讲，数据来源是input上游。对于input的size的限制，直接决定了数据流的时间复杂度。我们整个streaming的处理流程，或者说处理的每一个batch当中的时间复杂度是 $O(n \times m)$ ，如下图所示。



输入参数的重要性主要体现在如下图三点：



另外，我们刚才也提到了关于join场景。比方说我们接下来给到的例子，streaming以及静态数据的这样一个join的场景当中。如果说我们对于每个分片的shuffle size的大小，以及如何把它能够转化为一个内存shuffle的场景，也是对于我们输入参数进行一个合适调优的过程。





如果我们用系统默认的最大文件数来进行这样一个计算，会触发哪些问题。以Delta为例，默认选项就是一千个文件。假设我们的生产环境，每个文件有200MB。在我们的demo中会看到mini-batch越变越慢，也就是说，我们的消费者永远大于生产者，随着时间的推移，整条流的latency越来越大。我相信大家在生产环境中也会经常遇到类似场景。

## Input Parameters: Not tuning maxFilesPerTrigger

### What will happen when not setting maxFilesPerTrigger?

- For Delta: Default option is 1000 files. Each file is ~200 MB.  
For Message and other File-based input: Default option is unlimited.
- Leads to a massive mini-batch!
- When you have shuffle operations → Spill.

SPARK-AI SUMMIT

#Datateams #SparkAISummit

接下来的demo中，我们演示maxFilesPerTrigger的调参。这里我们有两个目标值。第一个目标值是我们希望每一个shuffle partition size在100~200MB。这个值是怎么来的，某种程度上，某一个集群配置，或者说某一种集群配置下，这个值都是不一致的。需要大家按照自己的集群大小，包括memory，配置进行一个预估和调整。第二个，我们需要shuffle partition和core的值是相等的。

## Input Parameters: Tuning maxFilesPerTrigger

### Base it on shuffle partition size

- Rule of thumb 1: Optimal shuffle partition size ~100-200 MB
- Rule of thumb 2: Set shuffle partitions equal to # of cores = 20.
- Use Spark UI to tune maxFilesPerTrigger until you get ~100-200 MB per partition.
- **Note:** Size on disk is not a good proxy for size in memory

Reason is that file size is different from the size in cluster memory

SPARK-AI SUMMIT

#Datateams #SparkAISummit

可以看到在当前demo中，我们通过maxFilesPerTrigger调整到6个files，这个时候就没有shuffle再发生了。同时，我们的Processed Records/Seconds比原来的这样一个比例调整上升了30%。

## Tuning maxFilesPerTrigger: Result

### Significant performance improvement by removing spill

- `maxFilesPerTrigger` tuned to 6 files.
- Shuffle partitions tuned to 20.
- Processed Records/Seconds increased by 30%

SPARK-AI SUMMIT

#Datateams #SparkAISummit

屏蔽了shuffle spill这样一个耗时操作之后，我们的调优工作就结束了吗，其实并不是。我们还可以做进一步的调优。正如刚才背景介绍所说，我们的一条动态流和一条静态的DataFrame进行join，但是我们的静态DataFrame其实完全意义上来讲是可以做broadcast hash join。

## Sort Merge Join vs Broadcast Hash Join

### We are not done yet!

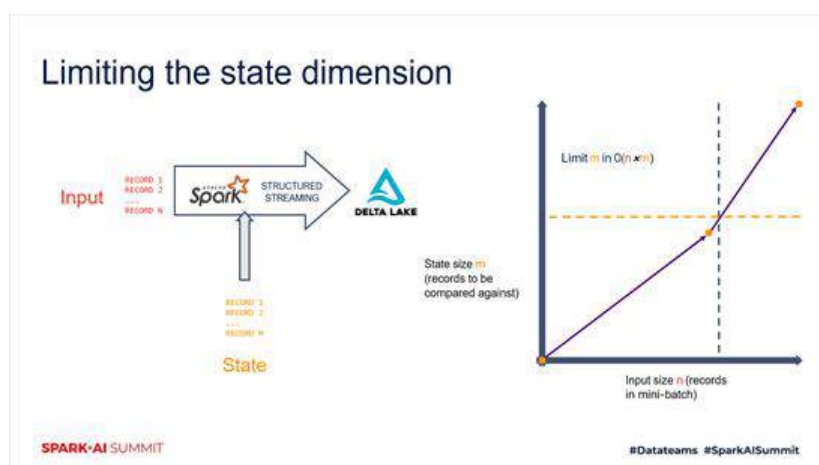
- Currently we use a Sort Merge Join.
- Our static DF is small enough to broadcast it.
- Leads to 70% increased throughput!
- Can also increase `maxFilesPerTrigger`  
Because of no more risk of Shuffle Spill (shuffles were removed)

SPARK-AI SUMMIT

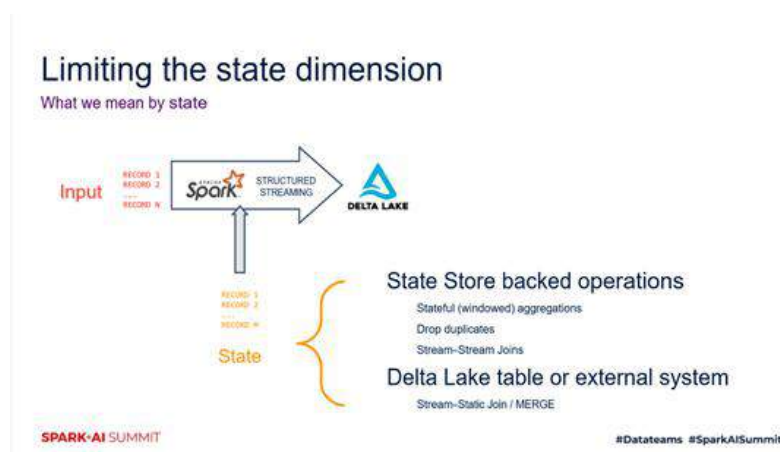
#Datateams #SparkAISummit

### 三、状态参数

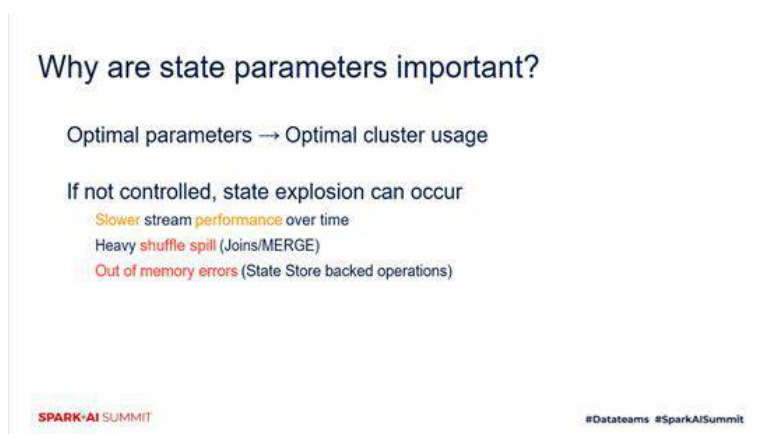
这一部分是关于状态参数。在我们刚才一开始提到的影响streaming的两个维度，一个是input size，另一个是state size。我们需要进一步的对state size有一个很好的限制。否则，我们每一个input，需要在state store当中去做查找，或者说做匹配，做一系列的两个batch之间的状态互通的操作的时候，就会导致一系列的性能问题。



这里的state主要包括两个部分：第一部分是State Store backed operations，第二部分是Delta Lake table or external system。



状态参数的重要性如下图所示。因为每一个batch都需要对state当中的操作进行查询以及按需更新的步骤。无限增长的state肯定会让你的作业越跑越慢，并且到最后消费者跟不上生产者。与此同时，它也会带来我们刚才遇到的问题，比方说shuffle spill或常见的out of memory的问题。



下一个demo中，我们主要会给大家演示的是这两类的state store。第一种与operation强相关。比方说，按需选择watermarking是我们需要保留的history的一个水位线。也就是说，我们当前的系统考虑多久之前的数据就算过期了，我们就可以不用再考虑。另外一个就是哪一种state store我们正在被使用。第二种完全跟具体操作无关，需要去看我们的query的predicate，具体的query实现是什么样的。然后看我们每一次参与计算的batch对应的量，以及state的一个修改情况。

### What parameters are we talking about?

State Store specific	State Store agnostic (Stream-Static Join / MERGE)
<ul style="list-style-type: none"> <li>How much history to compare against (watermarking)</li> <li>What state store backend to use (RocksDB / Default)</li> </ul>	<ul style="list-style-type: none"> <li>How much history to compare against (query predicate)</li> </ul>

SPARK-AI SUMMIT #Datateams #SparkAISummit

接下来看一下状态参数的例子。第一部分和第二部分跟上文中的例子保持一致，没有什么变化。这个例子着重于第3部分，Aggregate sales per item per hour。

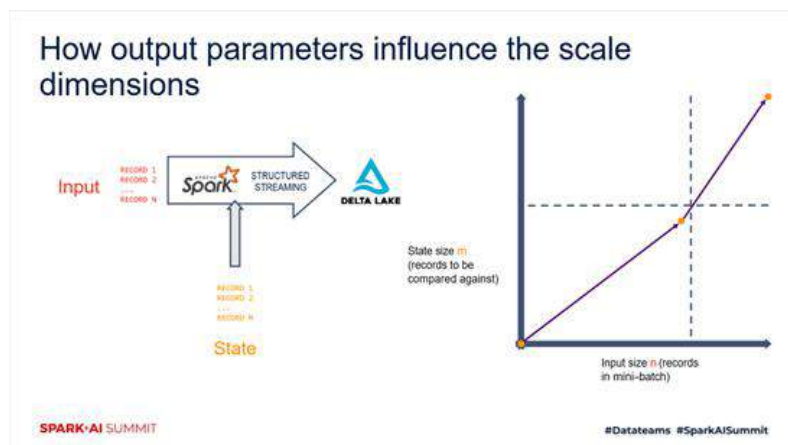
### State parameters example

<ul style="list-style-type: none"> <li>Extending the earlier code sample with stateful aggregation</li> <li>E.g. Calculating the number of sales per item category per hour</li> <li>Two types of state dimension here:               <ol style="list-style-type: none"> <li>Static side of the stream-static join (items)</li> <li>State Store backed operation (windowed stateful aggregation)</li> </ol> </li> </ul>	<ol style="list-style-type: none"> <li>1. Main input stream               <pre>salesSDF = {   spark     .readStream     .format("delta")     .table("sales") }</pre> </li> <li>2. Join item category lookup               <pre>itemSalesSDF = {   salesSDF     .join(spark.table("items"), "item_id") }</pre> </li> <li>3. Aggregate sales per item per hour               <pre>itemSalesPerHourSDF = {   itemSalesSDF     .groupBy(window(..., "1 hour"),              "item_category")     .sum("revenue") }</pre> </li> </ol>
---	--

SPARK-AI SUMMIT #Datateams #SparkAISummit

## 四、输出参数

接下来我们看一下输出参数，相对于前几种场景比较特殊。在上文中提到的input state对structured streaming的影响的二维象限里面，其实并没有output。Structured streaming框架本身并不会受到输出参数的影响。但是它更多的影响其实是在下游。对于streaming系统来讲，我们期望整个端到端的延迟也好，吞吐也好，达到一个整体最优化。



我们可以看到，如果输出参数不是一个很合适的值，最常见的问题就是小文件问题。我们会spill，或者说我们会写出大量的这种小的文件，导致下游的读取文件的操作会变得很慢。与此同时，以这样的方式去写出的时候，其实对框架本身也是一种开销。

### Why are output parameters important?

- Streaming jobs tend to create **many small files**
- Reading a folder with many small files **is slow**
- Degrading performance for downstream jobs / self-joins

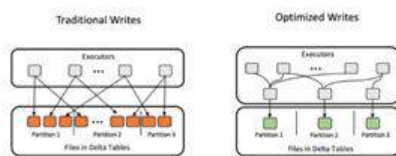
SPARK-AI SUMMIT #Datateams #SparkAISummit

其实商业系统当中对于输出参数是有一系列的优化的。比方说，Delta Lake系统当中对于output有一个Auto-Optimize的功能，它是默认打开的一个feature。Delta Lake在每一个batch写出的时候，他都会根据自己的合适的partition，以及file size进行一个自动的调优的输出。



## What Output parameters are we talking about?

- Manually using repartition
- Delta Lake: Auto-Optimize



<https://docs.databricks.com/delta/optimizations/auto-optimize.html>

SPARK-AI SUMMIT

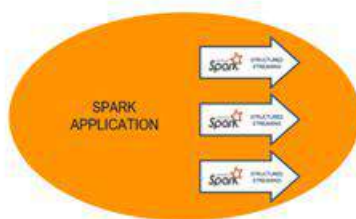
#DataTeams #SparkAISummit

## 五、部署

最后，对于我们在线下调整好streaming的参数，如何保证一系列的性能指标在上线之后也有同样好的性能。首先需要考虑的就是关于driver节点的部署。常见的情况来讲，一开始觉得非重要的streaming流，我们会将它线下的batch job与其他的streaming job进行一个混合部署，用同一个spark cluster。但是这种情况下，我们就会发现不同的作业会互相影响，尤其是driver节点会有一个对应的性能影响。所以在一开始部署，或者说在集群治理的角度，我们需要关注对应的streaming流是否可以和其他的作业进行混合部署。

## Multiple streams per Spark cluster

- Some small streams do not warrant their own cluster
- Packing them together in one Spark application might be a good option, but then they share driver process which has **performance impact**



SPARK-AI SUMMIT

#DataTeams #SparkAISummit

与此同时，有可能的弹性伸缩的需求。我们之前期望core和shuffle partition尽量一致，来尽可能的用到系统cluster当中的所有资源。但是如果说你后期会有弹性的部署需求，比方说，我希望有临时的一个集群扩展，对于streaming的cluster。这个时候，其实是有一个限制在里面。就是说如果你的shuffle partition一开始设置的比原有的core要小，这种情况下，集群的扩展对你原有的系统是没有用的。因为你的shuffle partition对应的参数写在checkpoint file当中去了，所以不会增加额外的。

## Temporary changes to load (elasticity)

- Temporary scaling up a streaming cluster to handle backlog
- Can only scale out until **#cores <= #shuffle partitions**

SPARK·AI SUMMIT

#Datateams #SparkAISummit

下一点有关于capacity planning。因为shuffle partition是一个fixed的checkpoint里边对应的一个值。我们如果说打破了对应的这样一个限制，真正重启作业的时候需要将checkpoint清掉。最后，我们需要在一开始的时候就考虑state恢复的机制。

## Permanent changes to load (capacity planning)

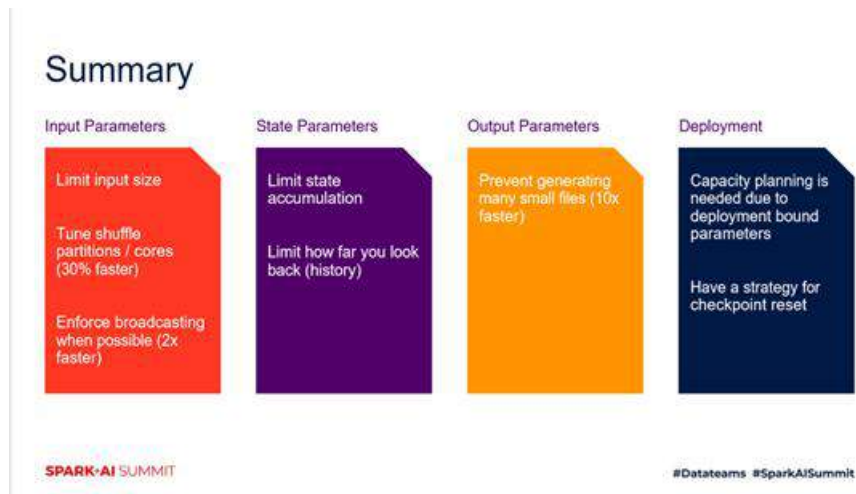
- Permanent load increase warrants capacity planning
- Requires checkpoint wipe-out **since shuffle partitions is fixed per checkpoint location!**
- Think of strategy to recover state (if necessary)

SPARK·AI SUMMIT

#Datateams #SparkAISummit



我们一共讲了四个方向，对于streaming性能有一定影响，如下图所示，分别是输入参数，状态参数，输出参数，以及部署。



# 使用Spark Streaming SQL进行PV/UV统计

简介： PV/UV统计是流式分析一个常见的场景。通过PV可以对访问的网站做流量或热点分析，例如广告主可以通过PV值预估投放广告网页所带来的流量以及广告收入。另外一些场景需要对访问的用户作分析，比如分析用户的网页点击行为，此时就需要对UV做统计。

作者：关文选，花名云魄，阿里云E-MapReduce 高级开发工程师，专注于流式计算，Spark Contributor

## 1. 背景介绍

PV/UV统计是流式分析一个常见的场景。通过PV可以对访问的网站做流量或热点分析，例如广告主可以通过PV值预估投放广告网页所带来的流量以及广告收入。另外一些场景需要对访问的用户作分析，比如分析用户的网页点击行为，此时就需要对UV做统计。

使用Spark Streaming SQL，并结合Redis可以很方便进行PV/UV的统计。本文将介绍通过Streaming SQL消费Loghub中存储的用户访问信息，对过去1分钟内的数据进行PV/UV统计，将结果存入Redis中。

## 2. 准备工作

- 创建E-MapReduce 3.23.0以上版本的Hadoop集群。
- 下载并编译E-MapReduce-SDK包。

编译完后, assembly/target目录下会生成emr-datasources\_shaded\_\${version}.jar，其中\${version}为sdk的版本。

```
git clone git@github.com:aliyun/aliyun-emapreduce-sdk.git
```

```
cd aliyun-emapreduce-sdk
```

```
git checkout -b master-2.x origin/master-2.x
```

```
mvn clean package -DskipTests
```

- 数据源

本文采用Loghub作为数据源，有关日志采集、日志解析请参考[日志服务](#)。

### 3. 统计PV/UV

一般场景下需要将统计出的PV/UV以及相应的统计时间存入Redis。其他一些业务场景中，也会只保存最新结果，用新的结果不断覆盖更新旧的数据。以下首先介绍第一种情况的操作流程。

#### 3.1 启动客户端

命令行启动streaming-sql客户端

```
streaming-sql --master yarn-client --num-executors 2 --executor-memory 2g --
executor-cores 2 --jars emr-datasources_shaded_2.11-${version}.jar --driver-class-path
emr-datasources_shaded_2.11-${version}.jar
```

也可以创建SQL语句文件，通过streaming-sql -f的方式运行。

#### 3.2 定义数据表

数据源表定义如下

```
CREATE TABLE loghub_source(user_ip STRING, __time__ TIMESTAMP)
USING loghub
OPTIONS(
  sls.project=${sls.project},
  sls.store=${sls.store},
  access.key.id=${access.key.id},
  access.key.secret=${access.key.secret},
  endpoint=${endpoint});
```

其中，数据源表包含user\_ip和\_\_time\_\_两个字段，分别代表用户的IP地址和loghub上的时间列。

OPTIONS中配置项的值根据实际配置。

结果表定义如下

```
CREATE TABLE redis_sink
USING redis
OPTIONS(
  table='statistic_info',
  host=${redis_host},
  key.column='user_ip');
```

其中，user\_ip对应数据中的用户IP字段，配置项\${redis\_host}的值根据实际配置。

### 3.3 创建流作业

```
CREATE SCAN loghub_scan
ON loghub_source
USING STREAM
OPTIONS(
watermark.column='__time__',
watermark.delayThreshold='10 second');
```

```
CREATE STREAM job
OPTIONS(
checkpointLocation=${checkpoint_location})
INSERT INTO redis_sink
SELECT COUNT(user_ip) AS pv, approx_count_distinct( user_ip) AS uv, window.end AS
interval
FROM loghub_scan
GROUP BY TUMBLING(__time__, interval 1 minute), window;
```

### 3.4 查看统计结果

```
emr-worker-1:6379> KEYS *
1) "statistic_info:2019-10-14 17:42:00.0"
2) "statistic_info:2019-10-14 17:43:00.0"
3) "_spark:statistic_info:schema"
emr-worker-1:6379> HGETALL "statistic_info:2019-10-14 17:42:00.0"
1) "uv"
2) "3"
3) "pv"
4) "86"
emr-worker-1:6379> HGETALL "statistic_info:2019-10-14 17:43:00.0"
1) "pv"
2) "59"
3) "uv"
4) "3"
emr-worker-1:6379> █
```

可以看到，每隔一分钟都会生成一条数据，key的形式为表名:interval，value为pv和uv的值。

### 3.5 实现覆盖更新

将结果表的配置项key.column修改为一个固定的值，例如定义如下：

```
CREATE TABLE redis_sink
USING redis
OPTIONS(
  table='statistic_info',
  host=${redis_host},
  key.column='statistic_type');
```

创建流作业的SQL改为：

```
CREATE STREAM job
OPTIONS(
  checkpointLocation='/tmp/spark-test/checkpoint')
INSERT INTO redis_sink
SELECT "PV_UV" as statistic_type,COUNT(user_ip) AS pv, approx_count_distinct( user_ip)
AS uv, window.end AS interval
FROM loghub_scan
GROUP BY TUMBLING(__time__, interval 1 minute), window;
```

最终的统计结果如下图所示

```
emr-worker-1:6379> KEYS *
1) "statistic_info:PV_UV"
2) "_spark:statistic_info:schema"
emr-worker-1:6379> HGETALL "statistic_info:PV_UV"
1) "uv"
2) "3"
3) "interval"
4) "2019-10-14 17:53:00.0"
5) "pv"
6) "20"
emr-worker-1:6379> HGETALL "statistic_info:PV_UV"
1) "uv"
2) "3"
3) "interval"
4) "2019-10-14 17:54:00.0"
5) "pv"
6) "58"
emr-worker-1:6379> KEYS *
1) "statistic_info:PV_UV"
2) "_spark:statistic_info:schema"
emr-worker-1:6379> █
```

可以看到，Redis中值保留了一个值，这个值每分钟都被更新，value包含pv、uv和interval的值。

## 4. 总结

本文简要介绍了使用Streaming SQL结合Redis实现流式处理中统计PV/UV的需求。后续文章，我将介绍Spark Streaming SQL的更多内容

# 自适应查询执行AQE：在运行时加速 SparkSQL

简介：SPARK+AI SUMMIT 2020中文精华版线上峰会将会带领大家一起回顾2020年的SPARK又产生了怎样的最佳实践，技术上取得了哪些突破，以及周边的生态发展。本文是阿里巴巴云智能平台事业部王道远关于Spark3.0中自适应查询执行（AQE）的相关介绍。以下由Spark+AI Summit中文精华版峰会的精彩内容整理。

## 一、自适应查询执行AQE简介

关于自适应查询执行，在数据库领域早有充分研究。在Spark社区，最早在Spark 1.6版本就已经提出发展自适应执行（Adaptive Query Execution，下文简称AQE）；到了Spark 2.x时代，Intel大数据团队进行了相应的原型开发和实践；到了Spark 3.0时代，Databricks和Intel一起为社区贡献了新的AQE。

什么是AQE呢？简单来说就是根据在运行时统计信息（runtime statistics）在查询执行的过程中进行动态（Dynamic）的查询优化。那么我们为什么需要AQE呢？在Spark 2.x时代，为了选择最佳执行计划，我们引入了CBO（Cost-based optimization），但是在一些场景下，效果非常不好，缺点明显，比如：

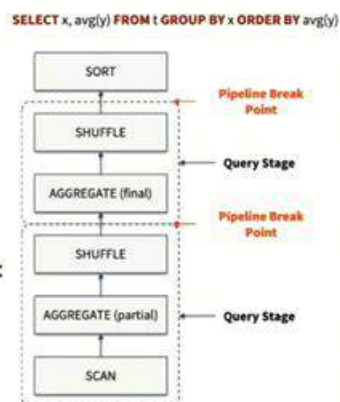
- 统计信息过期或者缺失导致估计错误；
- 收集统计信息代价较大（比如column histograms）；
- 某些谓词使用自定义UDF导致无法预估；
- 手动指定执行hint跟不上数据变化。

而在Spark 3.0时代，AQE完全基于精确的运行时时统计信息进行优化，引入了一个基本的概念Query Stages，并且以Query Stage为粒度，进行运行时的优化，其工作原理如下所示：



## Query Stages

- 由 Shuffle 和 broadcast exchange 把查询执行计划分为多个 query stage
- query stage 执行完成时获取中间结果。
- Query stage 边界是运行时优化的最佳时机:
  - 天然的执行间歇
  - 分区大小、数据大小等统计信息已经产生



- 整个AQE的工作原理以及流程为：运行没有依赖的stage；
- 在一个stage完成时再依据新的统计信息优化剩余部分；
- 执行其他已经满足依赖的stage；
- 重复步骤（2）（3）直至所有stage执行完成。

## 二、Spark 3.0中主要的AQE特性

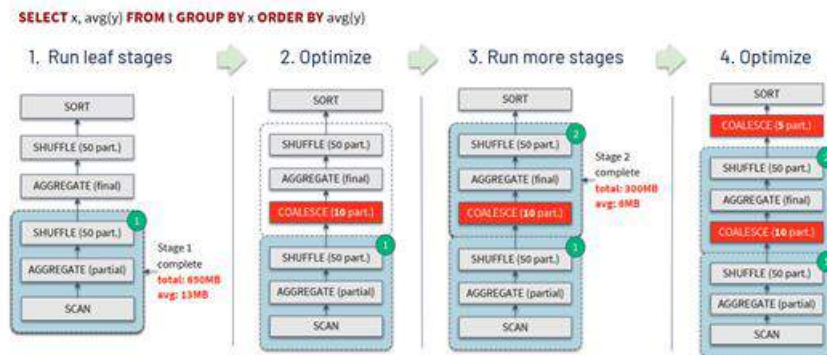
Spark 3.0中主要的AQE特性包括：

- 动态合并shuffle分区；
- 动态转换join策略；
- 动态优化join中的数据倾斜。

### （一）动态合并shuffle分区

Shuffle分区数量和大小对查询性能很关键。在Spark 3.0以前，Shuffle分区是一个固定值，存在着明显的缺点，如果分区过小会导致I/O低效、调度开销和任务启动开销，但是如果分区过大又会带来GC压力和溢写硬盘等问题。另一方面，在Spark 3.0之前，整个查询执行过程中使用统一的分区数，而在查询执行的不同阶段，数据规模会发生明显变化，如果保持统一的分区数，则大大降低了效率。基于以上，动态合并Shuffle分区是非常必要的。

AQE解决上面问题的具体做法是设置较大的初始分区数来满足整个查询执行过程中最大的分区数，并且在每个Query stage结束的时候按需自动合并分区，其具体的流程如下图所示：

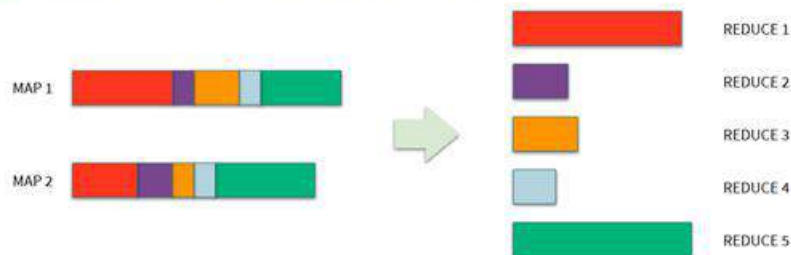


具体来说，动态合并Shuffle分区的原理如下：

对于普通的Shuffle来说，没有自动合并的过程，每个MAP读取Shuffle后，会根据指定分区数进行分区，比如下图为5：

普通的 shuffle - 没有自动合并

- 根据指定分区数进行分区，下图为 5



进行上图所示的分区后发现，REDUCE1和REDUCE5要处理的数据量明显高于其余三个REDUCE，而我们理想的情况下是每个REDUCE处理的数据量是相当的，所以AQE进行了动态合并分区，将相邻的小分区2, 3, 4进行合并，输出三个REDUCE，大大提高了后续的效率，如下图所示：

AQE 合并的 shuffle

- 合并相邻的小分区 2, 3, 4

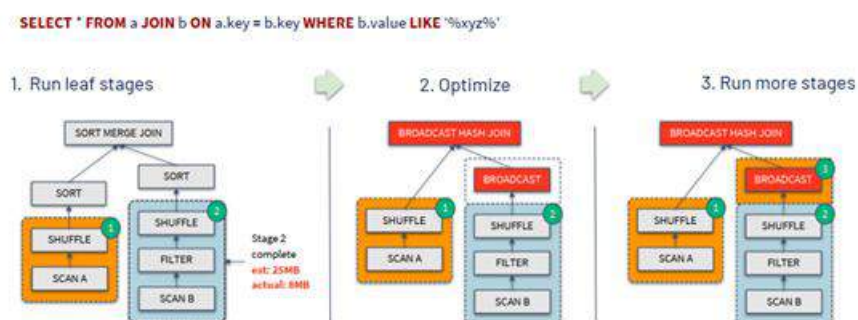


## （二）动态优化join中的数据倾斜

在Spark中，我们希望当Join的某一边可以完全放入内存时，Spark选择Broadcast Hash Join，但是实际上会出现预估可能不够准确，导致本来可以优化为BHJ的没有被优化的情况，原因也很多，比如；

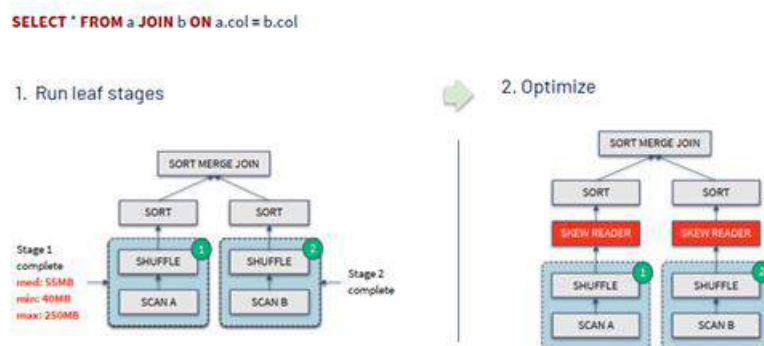
- 统计信息不够准确；
- 子查询太复杂；
- 黑盒的谓词，比如自定义UDF。

对于以上问题，AQE的解决方法就是使用运行时数据大小重新选择执行计划，其整个流程与原理如下图所示：



## （三）动态优化join中的数据倾斜

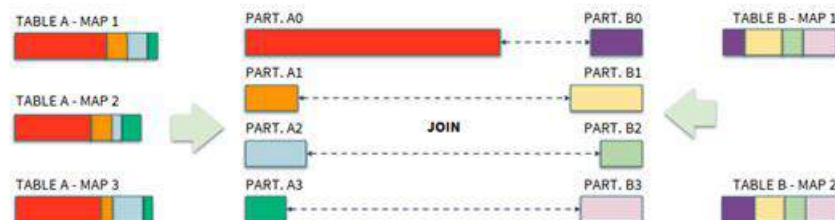
在Join中的数据倾斜会导致一系列的问题，比如性能下降、某一个task影响整个stage的运行等，处理数据量比较大的partitions时候还可能会出现溢写磁盘的情况。AQE针对上述问题使用运行时的统计信息自动优化查询执行，动态的发现倾斜数据的数量，并且把倾斜的分区分成更小的子分区来处理。其做法如下图所示：



具体来说其原理如下：

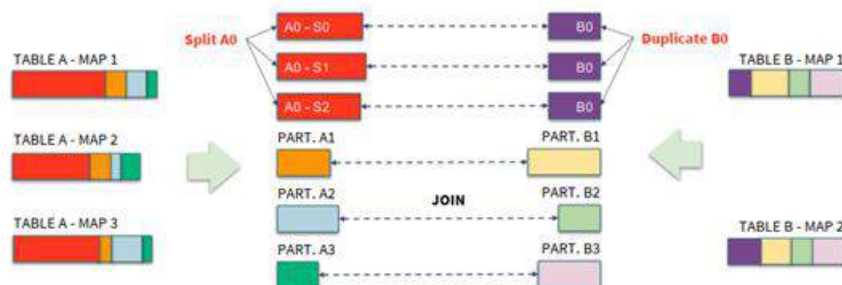
对于普通的sort merge join来说，没有倾斜优化，可能会造成某个Shuffle分区的数据数量明显高于其他分区，如下图中的PART.A0，这种情况会造成A0和B0的这个Join执行速度明显慢于其他的Join。

普通的 sort merge join – 没有倾斜优化:



有了AQE之后，根据数据倾斜优化后的sort merge join，使用skew Shuffle reader，如下图所示将A0分成三个子分区，并将对应的B0复制三份，整个Join任务的运行效率大大提升。

根据数据倾斜优化后的 sort merge join – 使用 skew shuffle reader:



### 三、TPC-DS性能测试

进行TPC-DS性能测试的集群配置如下图所示：

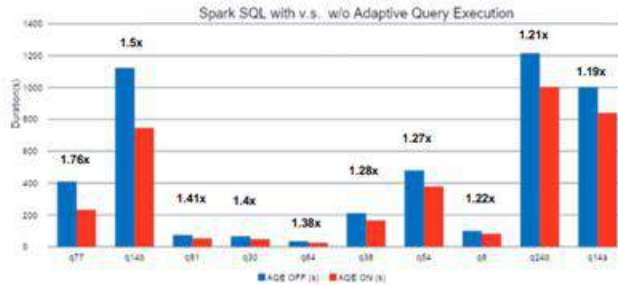
#### TPC-DS 性能 (3TB) – 集群配置

Hardware		BDW
Slave	Node#	5
	CPU	Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz (96cores)
	Memory	384 GB
	Disk	7× 1 TB SSD
	Network	10 Gigabit Ethernet
Master	CPU	Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz (96cores)
	Memory	384 GB
	Disk	7× 1 TB SSD
	Network	10 Gigabit Ethernet
Software		
OS	Fedora release 29	
Kernel	4.20.6-200.fc29.x86_64	
Spark*	Spark master (commit ID: 0b6aae422ba37a13531e98c8801589f5f3cb28e0)	
Hadoop*/HDFS*	hadoop-2.7.5	
JDK	1.8.0_110 (Oracle* Corporation)	

测试结果显示, 2条Query获得了1.5倍的性能提升, 37条Query获得了1.1倍的性能提升。

## TPC-DS 性能(3TB)- 结果

- Over 1.5x speedup on 2 queries; over 1.1x speedup on 37 queries



下面两张图是关于分区合并和Join策略的性能测试结果, 可以看出AQE对于性能的提升还是非常明显的。

## TPC-DS 性能(3TB)- 分区合并

- 调度和 task 启动开销更小
- 硬盘 IO 请求更少
- 写硬盘的数据更少, 因为聚合的数据更多。

Partitions Number 1000 (Q8 without AQE)

SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ...)	2020/05/13	9 s	1000/1000	74.0 MB	532.5 KB
processLine at CUDriver.java:336	<details>	20:01:03			
SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ...)	2020/05/13	11 s	1000/1000	221.1 MB	74.0 MB
processLine at CUDriver.java:336	<details>	20:00:52			

Partitions Number changed to 658 and 717 (Q8 with AQE)

SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ...)	2020/05/13	2 s	717/717	53.1 MB	531.6 KB
processLine at CUDriver.java:336	<details>	19:59:57			
SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ...)	2020/05/13	6 s	658/658	221.1 MB	53.1 MB
processLine at CUDriver.java:336	<details>	19:58:38			

## TPC-DS 性能(3TB)-- Join 策略

- Random IO read -> Sequence IO read
- Remote shuffle read -> local shuffle read.

SortMergeJoin (Q14b without AQE)

WITH cross_items AS (SELECT i_item, ss_item, sk FROM item, (SELECT iss_i, brand, st brand ...)	2020/06/05 02:00:12	8.8 min	1000/1000	141.2 GB	9.4 MB
processLine at CUDriver.java:336	<details>				
WITH cross_items AS (SELECT i_item, ss_item, sk FROM item, (SELECT iss_i, brand, st brand ...)	2020/06/05 02:00:12	13 min	1000/1000	141.2 GB	11.0 MB
processLine at CUDriver.java:336	<details>				
WITH cross_items AS (SELECT i_item, ss_item, sk FROM item, (SELECT iss_i, brand, st brand ...)	2020/06/05 01:57:56	2.3 min	3473/3473	56.1 GB	141.2 GB
processLine at CUDriver.java:336	<details>				
Broadcast exchange (urlid:4266466-0266-4038-d635-346989-2018a)	2020/06/05 01:57:05	0.7 s	1000/1000	17.8 MB	
call at FutureTask.java:288	<details>				

Broadcast Hash Join (Q14b with AQE)

WITH cross_items AS (SELECT i_item, ss_item, sk FROM item, (SELECT iss_i, brand, st brand ...)	2020/06/05 00:46:07	7.8 min	3473/3473	141.2 GB	194.2 MB
processLine at CUDriver.java:336	<details>				
WITH cross_items AS (SELECT i_item, ss_item, sk FROM item, (SELECT iss_i, brand, st brand ...)	2020/06/05 00:46:07	3.7 min	3473/3473	141.2 GB	526.3 MB
processLine at CUDriver.java:336	<details>				
Broadcast exchange (urlid:4266466-0266-4038-d635-346989-2018a)	2020/06/05 00:46:06	0.3 s	575/575	4.8 MB	
call at FutureTask.java:288	<details>				
Broadcast exchange (urlid:4266466-0266-4038-d635-346989-2018a)	2020/06/05 00:46:06	0.2 s	738/738	12.4 MB	
call at FutureTask.java:288	<details>				

除了在TPC-DS的测试中AQE表现优秀，在实际生产环境中AQE对于性能的提升也非常优秀，比如某电商公司分享在某些典型的倾斜查询中使用了AQE之后获得了十几倍的性能提升，某互联网巨头使用了AQE之后发现在2个典型的查询中性能分别有了5倍和1.38倍的提升等等。

## 四、Q&A

**Q1：** Shuffle是如何对大量小文件进行优化的？

**A1：** AQE 支持的动态分区合并可以减少 shuffle 后的分区数，如果是 ETL 作业写动态分区表，建议手动添加distribute by partkey 等子句来减少输出文件数量。

**Q2：** AQE是否支持外部的Shuffle Service？

**A2：** 支持，需要 shuffle service 提供基本的统计信息

**Q3：** 如果join的两边的part都比较大，是不是都会拆分？还会broadcast 么？

**A3：** 都比较大的话优化就没啥用了，需要从业务出发进行优化。



# 浅析Hive/Spark SQL读文件时的输入任务划分

简介：本文最后留个思考题给读者们：如何设置参数彻底关闭Spark SQL data source表的文件合并？积极回答问题即可获得社区礼物。

作者：王道远，花名健身，阿里云EMR技术专家，Apache Spark活跃贡献者，主要关注大数据计算优化相关工作。

Hive以及Spark SQL等大数据计算引擎为我们操作存储在HDFS上结构化数据提供了易于上手的SQL接口，大大降低了ETL等操作的门槛，也因此在实际生产中有着广泛的应用。SQL是非过程化语言，我们写SQL的时候并不能控制具体的执行过程，它们依赖执行引擎决定。而Hive和Spark SQL作为Map-Reduce模型的分布式执行引擎，其执行过程首先就涉及到如何将输入数据切分成一个个任务，分配给不同的Map任务。在本文中，我们就来讲解Hive和Spark SQL是如何切分输入路径的。

## Hive

Hive是起步较早的SQL on Hadoop项目，最早也是诞生于Hadoop中，所以输入划分这部分代码与Hadoop相关度非常高。现在Hive普遍使用的输入格式是CombineHiveInputFormat，它继承于HiveInputFormat，而HiveInputFormat实现了Hadoop的InputFormat接口，其中的getSplits

方法用来获取具体的划分结果，划分出一份输入数据被称为一个“Split”。在执行时，每个Split对应到一个map任务。在划分Split时，首先挑出不能合并到一起的目录——比如开启了事务功能的路径。这些不能合并的目录必须单独处理，剩下的路径交给私有方法getCombineSplits，这样Hive的一个map task最多可以处理多个目录下的文件。在实际操作中，我们一般只要通过set mapred.max.split.size=xx;即可控制文件合并的大小。当一个文件过大时，父类的getSplits也会帮我们完成相应的切分工作。

## Spark SQL

Spark的表有两种：DataSource表和Hive表。另外Spark后续版本中DataSource V2也将逐渐流行，目前还在不断发展中，暂时就不在这里讨论。我们知道Spark SQL其实底层是Spark RDD，而RDD执行时，每个map task会处理RDD的一个Partition中的数据（注意这里的Partition是RDD的概念，要和表的Partition进行区分）。因此，Spark SQL作业的任务切分关键在于底层RDD的partition如何切分。

## Data Source表

Spark SQL的DataSource表在最终执行的RDD类为FileScanRDD，由FileSourceScanExec创建出来。在创建这种RDD的时候，具体的Partition直接作为参数传给了构造函数，因此划分输入的方法也在DataSourceScanExec.scala文件中。具体分两步：首先把文件划分为PartitionFile，再将较小的PartitionFile进行合并。

第一步部分代码如下：

```
if (fsRelation.fileFormat.isSplittable(
  fsRelation.sparkSession, fsRelation.options, file.getPath)) {
  (0L until file.getLength by maxSplitBytes).map { offset =>
    val remaining = file.getLength - offset
    val size = if (remaining > maxSplitBytes) maxSplitBytes else remaining
    val hosts = getBlockHosts(blockLocations, offset, size)
    PartitionedFile(
      partition.values, file.getPath.toUri.toString,
      offset, size, partitionDeleteDeltas, hosts)
  }
} else {
  val hosts = getBlockHosts(blockLocations, 0, file.getLength)
  Seq(PartitionedFile(partition.values, file.getPath.toUri.toString,
    0, file.getLength, partitionDeleteDeltas, hosts))
}
```

第二部分代码如下所示：

```
splitFiles.foreach { file =>
  if (currentSize + file.length > maxSplitBytes) {
    closePartition()
  }
  // Add the given file to the current partition.
  currentSize += file.length + openCostInBytes
  currentFiles += file
}
```

这样我们就可以依次遍历第一步切好的块，再按照`maxSplitBytes`进行合并。注意合并文件时还需加上打开文件的预估代价`openCostInBytes`。那么`maxSplitBytes`和`openCostInBytes`这两个关键参数怎么来的呢？

```
val defaultMaxSplitBytes =
  fsRelation.sparkSession.sessionState.conf.filesMaxPartitionBytes
val openCostInBytes = fsRelation.sparkSession.sessionState.conf.filesOpenCostInBytes
val defaultParallelism = fsRelation.sparkSession.sparkContext.defaultParallelism
val totalBytes = selectedPartitions.flatMap(_.files.map(_.getLen + openCostInBytes)).sum
val bytesPerCore = totalBytes / defaultParallelism

val maxSplitBytes = Math.min(defaultMaxSplitBytes, Math.max(openCostInBytes,
  bytesPerCore))
```

不难看出，主要是`spark.sql.files.maxPartitionBytes`、`spark.sql.files.openCostInBytes`、调度器默认并发度以及所有输入文件实际大小所控制。

## Hive表

Spark SQL中的Hive表底层的RDD类为`HadoopRDD`，由`HadoopTableReader`类实现。不过这次，具体的`Partition`划分还是依赖`HadoopRDD`的`getPartitions`方法，具体实现如下：

```
override def getPartitions: Array[Partition] = {
  ...
  try {
    val allInputSplits = getInputFormat(jobConf).getSplits(jobConf, minPartitions)
    val inputSplits = if (ignoreEmptySplits) {
      allInputSplits.filter(_.getLength > 0)
    } else {
      allInputSplits
    }
    val array = new Array[Partition](inputSplits.size)
    for (i <- 0 until inputSplits.size) {
      array(i) = new HadoopPartition(id, i, inputSplits(i))
    }
    array
  } catch {
    ...
  }
}
```

不难看出，在处理Hive表的时候，Spark SQL把任务划分又交给了Hadoop的InputFormat那一套。不过需要注意的是，并不是所有Hive表都归为这一类，Spark SQL会默认对ORC和Parquet的表进行转化，用自己的Data Source实现OrcFileFormat和ParquetFileFormat来把这两种表作为Data Source表来处理。

## 总结

切分输入路径只是大数据处理的第一步，虽然不起眼，但是也绝对不可或缺。低效的文件划分可能会给端到端的执行速度带来巨大的负面影响，更有可能影响到输出作业的文件布局，从而影响到整个数据流水线上所有作业的执行效率。万事开头难，为程序输入选择合适的配置参数，可以有效改善程序执行效率。

留个思考题给读者们：如何设置参数彻底关闭Spark SQL data source表的文件合并？



更多精彩内容扫码关注  
Spark 中文社区微信公众号



钉钉扫码加入  
Spark 中文社区钉钉群



阿里云开发者“藏经阁”  
海量电子书免费下载