Apache Spark SQL自适应执行实践

本文作者：汪愈舟 俞育才 郭晨钊 程浩（英特尔），李元健（百度）

Spark SQL是Apache Spark最广泛使用的一个组件，它提供了非常友好的接口来分布式处理结构化数据，在很多应用领域都有成功的生产实践，但是在超大规模集群和数据集上，Spark SQL仍然遇到不少易用性和可扩展性的挑战。为了应对这些挑战，英特尔大数据技术团队和百度大数据基础架构部工程师在Spark
社区版本的基础上，改进并实现了自适应执行引擎。本文首先讨论Spark SQL在大规模数据集上遇到的挑战，然后介绍自适应执行的背景和基本架构，以及自适应执行如何应对Spark
SQL这些问题，最后我们将比较自适应执行和现有的社区版本Spark SQL在100 TB 规模TPC-DS基准测试碰到的挑战和性能差异，以及自适应执行在Baidu Big SQL平台的使用情况。

挑战1：关于shuffle partition数

在Spark SQL中， shufflepartition数可以通过参数spark.sql.shuffle.partition来设置，默认值是200。这个参数决定了SQL作业每个reduce阶段任务数量，对整个查询性能有很大影响。假设一个查询运行前申请了E个Executor，每个Executor包含C个core（并发执行线程数），那么该作业在运行时可以并行执行的任务数就等于E x C个，或者说该作业的并发数是E x C。假设shuffle partition个数为P，除了map
stage的任务数和原始数据的文件数量以及大小相关，后续的每个reduce
stage的任务数都是P。由于Spark作业调度是抢占式的，E x C个并发任务执行单元会抢占执行P个任务，“能者多劳”，直至所有任务完成，则进入到下一个Stage。但这个过程中，如果有任务因为处理数据量过大（例如：数据倾斜导致大量数据被划分到同一个reducer partition）或者其它原因造成该任务执行时间过长，一方面会导致整个stage执行时间变长，另一方面E x
C个并发执行单元大部分可能都处于空闲等待状态，集群资源整体利用率急剧下降。

那么spark.sql.shuffle.partition参数究竟是多少比较合适？如果设置过小，分配给每一个reduce任务处理的数据量就越多，在内存大小有限的情况下，不得不溢写（spill）到计算节点本地磁盘上。Spill会导致额外的磁盘读写，影响整个SQL查询的性能，更差的情况还可能导致严重的GC问题甚至是OOM。相反，如果shuffle partition设置过大。第一，每一个reduce任务处理的数据量很小并且很快结束，进而导致Spark任务调度负担变大。第二，每一个mapper任务必须把自己的shuffle输出数据分成P个hash bucket，即确定数据属于哪一个reduce partition，当shuffle partition数量太多时，hash bucket里数据量会很小，在作业并发数很大时，reduce任务shuffle拉取数据会造成一定程度的随机小数据读操作，当使用机械硬盘作为shuffle数据临时存取的时候性能下降会更加明显。最后，当最后一个stage保存数据时会写出P个文件，也可能会造成HDFS文件系统中大量的小文件。

从上，shuffle partition的设置既不能太小也不能太大。为了达到最佳的性能，往往需要经多次试验才能确定某个SQL查询最佳的shuffle partition值。然而在生产环境中，往往SQL以定时作业的方式处理不同时间段的数据，数据量大小可能变化很大，我们也无法为每一个SQL查询去做耗时的人工调优，这也意味这些SQL作业很难以最佳的性能方式运行。

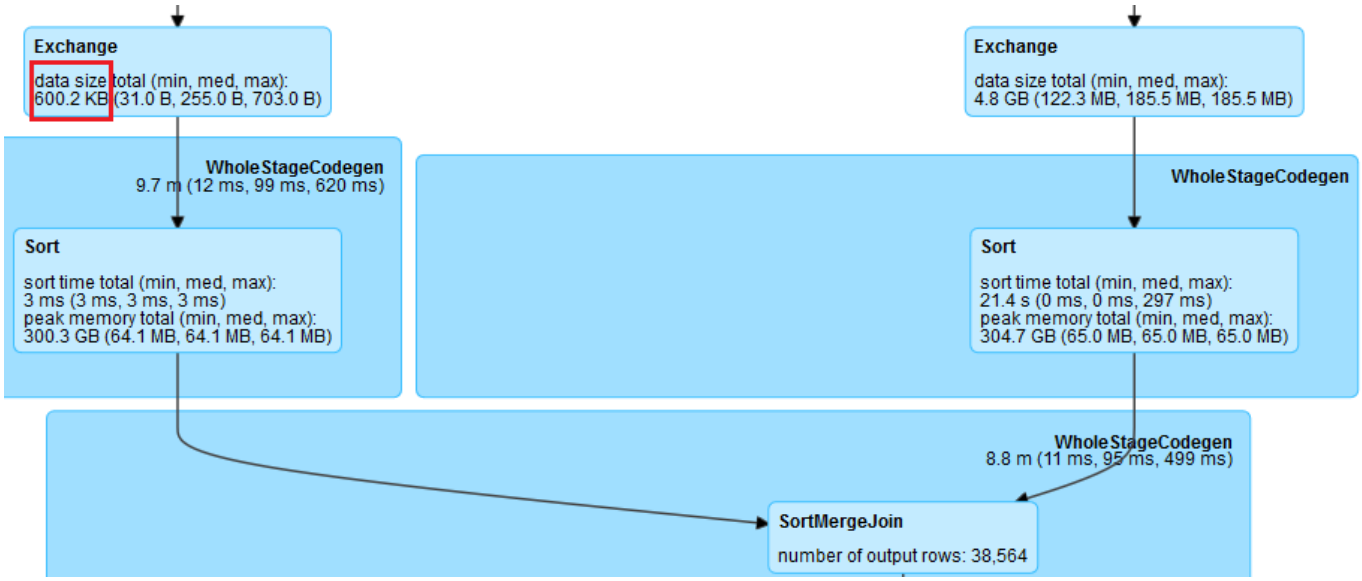
Shuffle partition的另外一个问题是，同一个shuffle partition数设置将应用到所有的stage。Spark在执行一个SQL作业时，会划分成多个stage。通常情况下，每个stage的数据分布和大小可能都不太一样，全局的shuffle
partition设置最多只能对某个或者某些stage最优，没有办法做到全局所有的stage设置最优。

这一系列关于shufflepartition的性能和易用性挑战，促使我们思考新的方法：我们能否根据运行时获取的shuffle数据量信息，例如数据块大小，记录行数等等，自动为每一个stage设置合适的shuffle partition值？

挑战2：Spark SQL最佳执行计划

Spark SQL在执行SQL之前，会将SQL或者Dataset程序解析成逻辑计划，然后经历一系列的优化，最后确定一个可执行的物理计划。最终选择的物理计划的不同对性能有很大的影响。如何选择最佳的执行计划，这便是Spark
SQL的Catalyst优化器的核心工作。Catalyst早期主要是基于规则的优化器（RBO），在Spark 2.2中又加入了基于代价的优化（CBO）。目前执行计划的确定是在计划阶段，一旦确认以后便不再改变。然而在运行期间，当我们获取到更多运行时信息时，我们将有可能得到一个更佳的执行计划。

以join操作为例，在Spark中最常见的策略是BroadcastHashJoin和SortMergeJoin。BroadcastHashJoin属于map side join，其原理是当其中一张表存储空间大小小于broadcast阈值时，Spark选择将这张小表广播到每一个Executor上，然后在map阶段，每一个mapper读取大表的一个分片，并且和整张小表进行join，整个过程中避免了把大表的数据在集群中进行shuffle。而SortMergeJoin在map阶段2张数据表都按相同的分区方式进行shuffle写，reduce阶段每个reducer将两张表属于对应partition的数据拉取到同一个任务中做join。RBO根据数据的大小，尽可能把join操作优化成BroadcastHashJoin。Spark中使用参数spark.sql.autoBroadcastJoinThreshold来控制选择BroadcastHashJoin的阈值，默认是10MB。然而对于复杂的SQL查询，它可能使用中间结果来作为join的输入，在计划阶段，Spark并不能精确地知道join中两表的大小或者会错误地估计它们的大小，以致于错失了使用BroadcastHashJoin策略来优化join执行的机会。但是在运行时，通过从shuffle写得到的信息，我们可以动态地选用BroadcastHashJoin。以下是一个例子，join一边的输入大小只有600K，但Spark仍然规划成SortMergeJoin。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

这促使我们思考第二个问题：我们能否通过运行时收集到的信息，来动态地调整执行计划？

挑战3：数据倾斜

数据倾斜是常见的导致Spark SQL性能变差的问题。数据倾斜是指某一个partition的数据量远远大于其它partition的数据，导致个别任务的运行时间远远大于其它任务，因此拖累了整个SQL的运行时间。在实际SQL作业中，数据倾斜很常见，join key对应的hash bucket总是会出现记录数不太平均的情况，在极端情况下，相同join key对应的记录数特别多，大量的数据必然被分到同一个partition因而造成数据严重倾斜。如图2，可以看到大部分任务3秒左右就完成了，而最慢的任务却花了4分钟，它处理的数据量却是其它任务的若干倍。

Summary Metrics for 5600 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	15 ms	2 s	3 s	5 s	4.0 min
Scheduler Delay	0 ms	2 ms	2 ms	3 ms	1 s
Task Deserialization Time	3 ms	5 ms	6 ms	7 ms	0.4 s
GC Time	0 ms	0 ms	0.2 s	0.3 s	35 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	3 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Shuffle Read Blocked Time	0 ms	44 ms	0.2 s	0.6 s	31 s
Shuffle Read Size / Records	0.0 B / 0	3.5 MB / 374415	4.9 MB / 747052	6.9 MB / 1783859	172.8 MB / 283732661
Shuffle Remote Reads	0.0 B	3.4 MB	4.7 MB	6.6 MB	166.6 MB
Shuffle Write Size / Records	0.0 B / 0	109.0 B / 3	132.0 B / 4	158.0 B / 6	262.0 B / 13
Shuffle spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	13.3 GB
Shuffle spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	68.2 MB

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

目前，处理join时数据倾斜的一些常见手段有：(1)增加shuffle partition数量，期望原本分在同一个partition中的数据可以被分散到多个partition中，但是对于同key的数据没有作用。(2)调大BroadcastHashJoin的阈值，在某些场景下可以把SortMergeJoin转化成BroadcastHashJoin而避免shuffle产生的数据倾斜。(3)手动过滤倾斜的key，并且对这些数据加入随机的前缀，在另一张表中这些key对应的数据也相应的膨胀处理，然后再做join。综上，这些手段都有各自的局限性并且涉及很多的人为处理。基于此，我们思考了第三个问题：Spark能否在运行时自动地处理join中的数据倾斜？

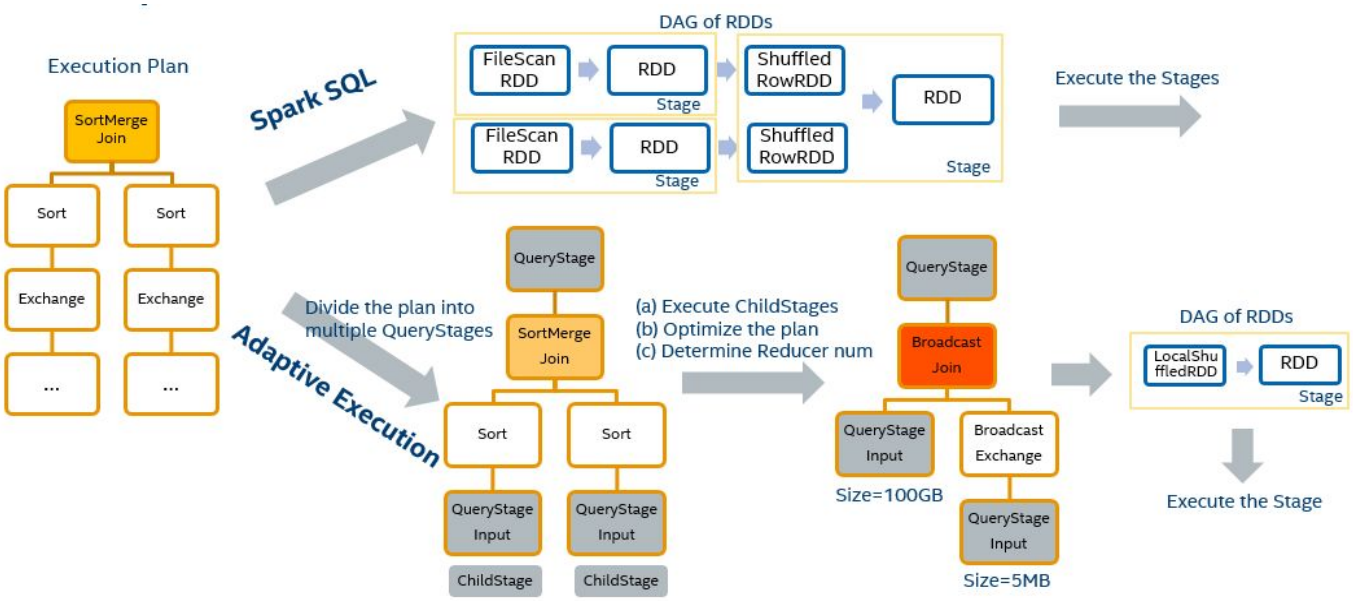
自适应执行背景和简介

早在2015年，Spark社区就提出了自适应执行的基本想法，在Spark的DAGScheduler中增加了提交单个map stage的接口，并且在实现运行时调整shuffle partition数量上做了尝试。但目前该实现有一定的局限性，在某些场景下会引入更多的shuffle，即更多的stage，对于三表在同一个stage中做join等情况也无法很好的处理。所以该功能一直处于实验阶段，配置参数也没有在官方文档中提及。

基于这些社区的工作，英特尔大数据技术团队对自适应执行做了重新的设计，实现了一个更为灵活的自适性执行框架。在这个框架下面，我们可以添加额外的规则，来实现更多的功能。目前，已实现的特性包括：自动设置shuffle partition数，动态调整执行计划，动态处理数据倾斜等等。

自适应执行架构

在Spark SQL中，当Spark确定最后的物理执行计划后，根据每一个operator对RDD的转换定义，它会生成一个RDD的DAG图。之后Spark基于DAG图静态划分stage并且提交执行，所以一旦执行计划确定后，在运行阶段无法再更新。自适应执行的基本思路是在执行计划中事先划分好stage，然后按stage提交执行，在运行时收集当前stage的shuffle统计信息，以此来优化下一个stage的执行计划，然后再提交执行后续的stage。

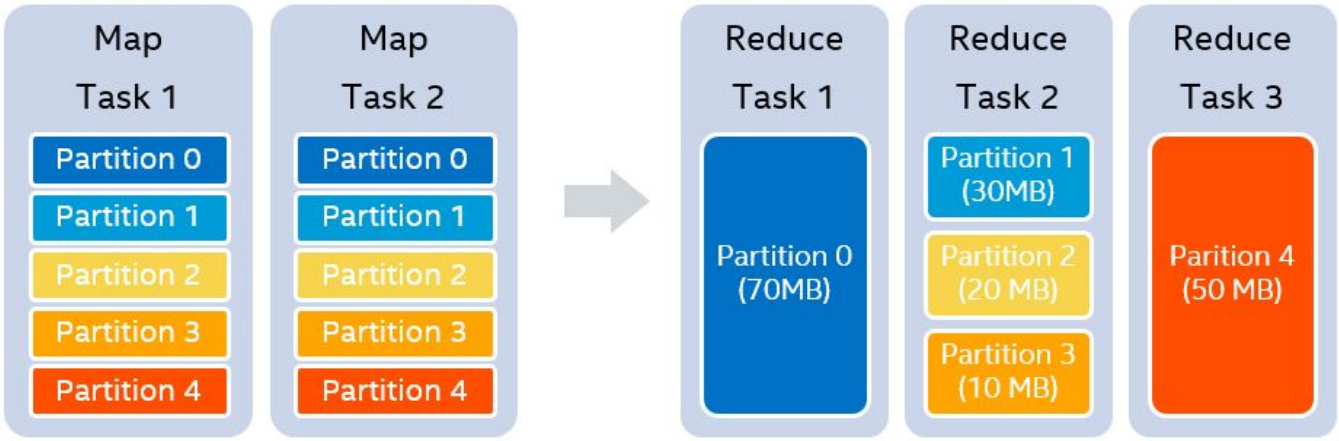


如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

从图3中我们可以看出自适应执行的工作方法，首先以Exchange节点作为分界将执行计划这棵树划分成多个QueryStage（Exchange节点在Spark SQL中代表shuffle）。每一个QueryStage都是一棵独立的子树，也是一个独立的执行单元。在加入QueryStage的同时，我们也加入一个QueryStageInput的叶子节点，作为父亲QueryStage的输入。例如对于图中两表join的执行计划来说我们会创建3个QueryStage。最后一个QueryStage中的执行计划是join本身，它有2个QueryStageInput代表它的输入，分别指向2个孩子的QueryStage。在执行QueryStage时，我们首先提交它的孩子stage，并且收集这些stage运行时的信息。当这些孩子stage运行完毕后，我们可以知道它们的大小等信息，以此来判断QueryStage中的计划是否可以优化更新。例如当我们获知某一张表的大小是5M，它小于broadcast的阈值时，我们可以将SortMergeJoin转化成BroadcastHashJoin来优化当前的执行计划。我们也可以根据孩子stage产生的shuffle数据量，来动态地调整该stage的reducer个数。在完成一系列的优化处理后，最终我们为该QueryStage生成RDD的DAG图，并且提交给DAG Scheduler来执行。

自动设置reducer个数

假设我们设置的shufflepartition个数为5，在map stage结束之后，我们知道每一个partition的大小分别是70MB，30MB，20MB，10MB和50MB。假设我们设置每一个reducer处理的目标数据量是64MB，那么在运行时，我们可以实际使用3个reducer。第一个reducer处理partition 0 (70MB)，第二个reducer处理连续的partition 1 到3，共60MB，第三个reducer处理partition 4 (50MB)，如图4所示。



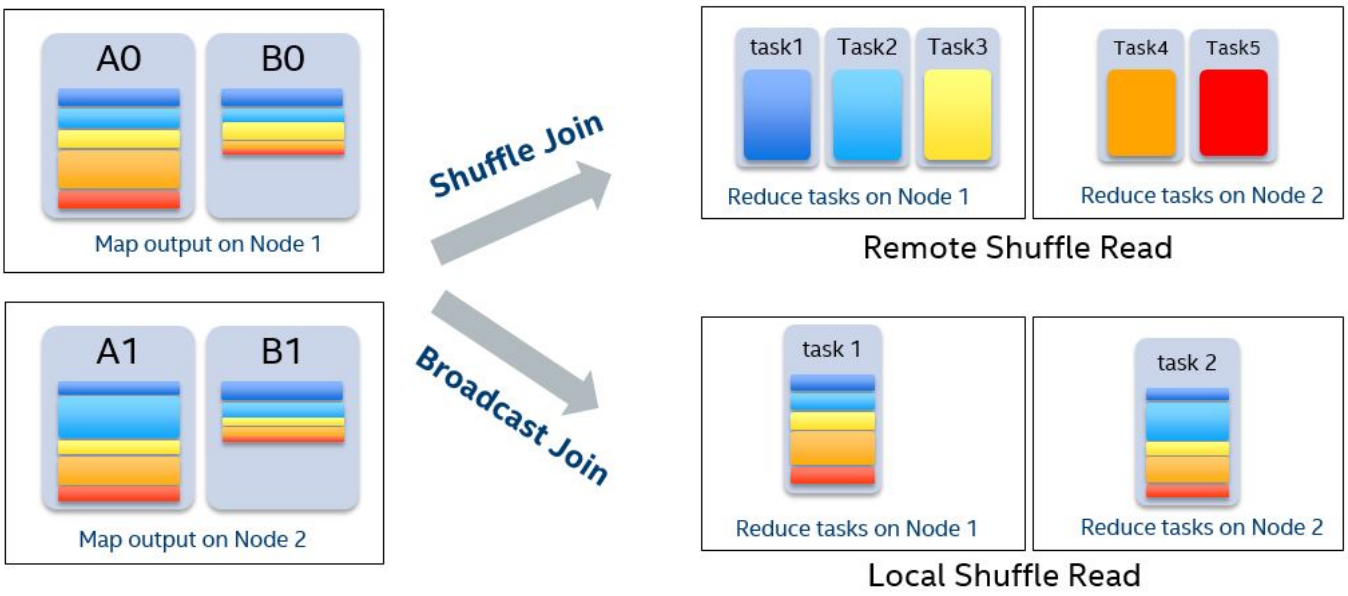
如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

在自适应执行的框架中，因为每个QueryStage都知道自己所有的孩子stage，因此在调整reducer个数时，可以考虑到所有的stage输入。另外，我们也可以将记录条数作为一个reducer处理的目标值。因为shuffle的数据往往都是经过压缩的，有时partition的数据量并不大，但解压后记录条数确远远大于其它partition，造成数据不均。所以同时考虑数据大小和记录条数可以更好地决定reducer的个数。

动态调整执行计划

目前我们支持在运行时动态调整join的策略，在满足条件的情况下，即一张表小于Broadcast阈值，可以将SortMergeJoin转化成BroadcastHashJoin。由于SortMergeJoin和BroadcastHashJoin输出的partition情况并不相同，随意转换可能在下一个stage引入额外的shuffle操作。因此我们在动态调整join策略时，遵循一个规则，即在不引入额外shuffle的前提下才进行转换。

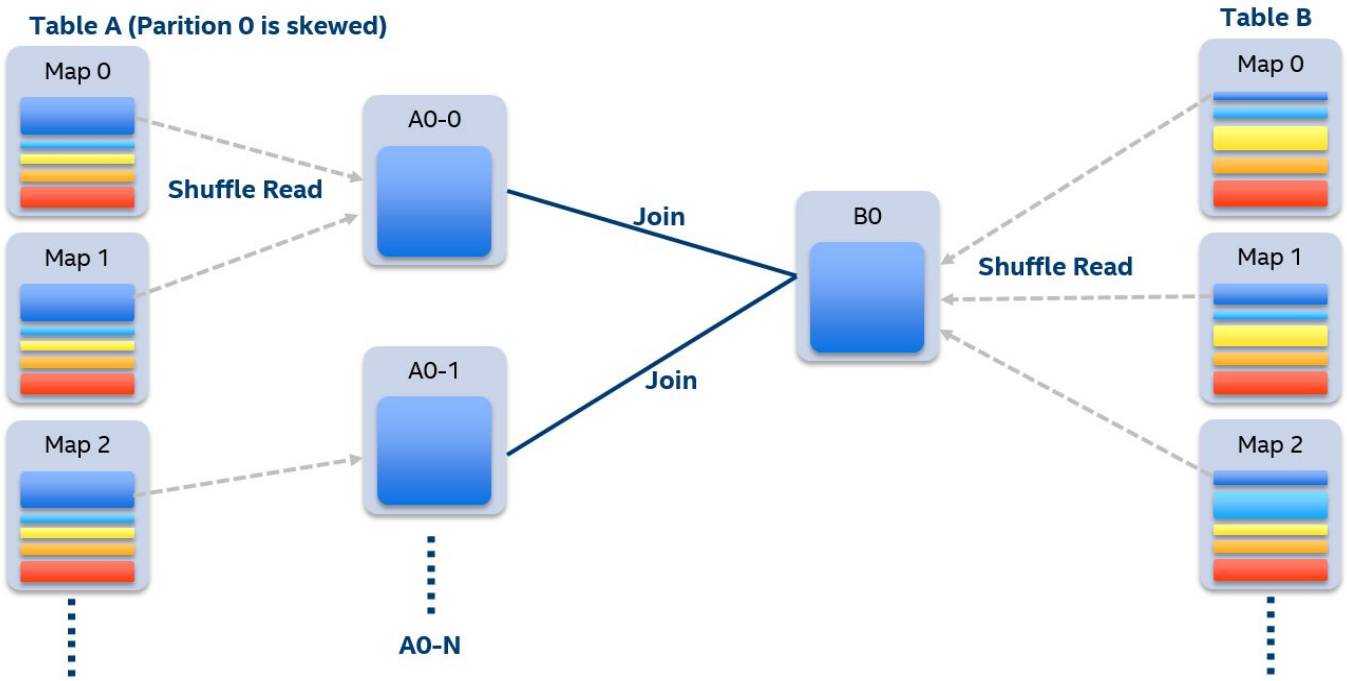
将SortMergeJoin转化成BroadcastHashJoin有哪些好处呢？因为数据已经shuffle写到磁盘上，我们仍然需要shuffle读取这些数据。我们可以看看图5的例子，假设A表和B表join，map阶段2张表各有2个map任务，并且shuffle partition个数为5。如果做SortMergeJoin，在reduce阶段需要启动5个reducer，每个reducer通过网络shuffle读取属于自己的数据。然而，当我们在运行时发现B表可以broadcast，并且将其转换成BroadcastHashJoin之后，我们只需要启动2个reducer，每一个reducer读取一个mapper的整个shuffle output文件。当我们调度这2个reducer任务时，可以优先将其调度在运行mapper的Executor上，因此整个shuffle读变成了本地读取，没有数据通过网络传输。并且读取一个文件这样的顺序读，相比原先shuffle时随机的小文件读，效率也更胜一筹。另外，SortMergeJoin过程中往往会出现不同程度的数据倾斜问题，拖慢整体的运行时间。而转换成BroadcastHashJoin后，数据量一般比较均匀，也就避免了倾斜，我们可以在下文实验结果中看到更具体的信息。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

动态处理数据倾斜

在自适应执行的框架下，我们可以在运行时很容易地检测出有数据倾斜的partition。当执行某个stage时，我们收集该stage每个mapper 的shuffle数据大小和记录条数。如果某一个partition的数据量或者记录条数超过中位数的N倍，并且大于某个预先配置的阈值，我们就认为这是一个数据倾斜的partition，需要进行特殊的处理。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

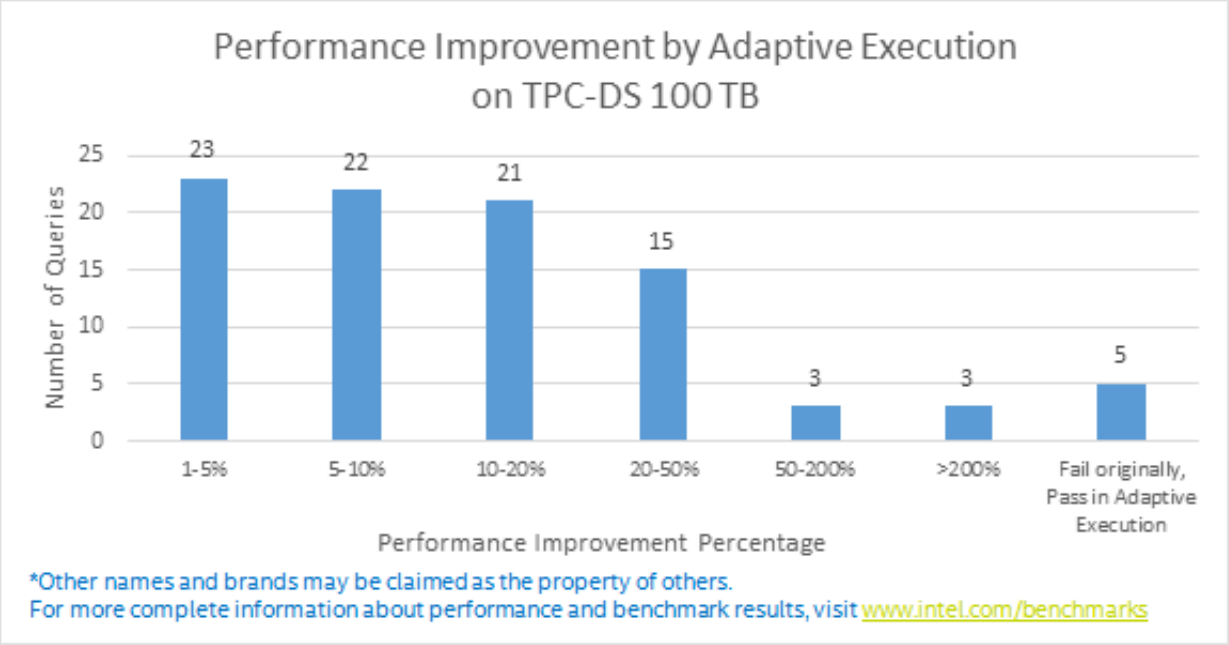
假设我们A表和B表做inner
join，并且A表中第0个partition是一个倾斜的partition。一般情况下，A表和B表中partition 0的数据都会shuffle到同一个reducer中进行处理，由于这个reducer需要通过网络拉取大量的数据并且进行处理，它会成为一个最慢的任务拖慢整体的性能。在自适应执行框架下，一旦我们发现A表的partition
0发生倾斜，我们随后使用N个任务去处理该partition。每个任务只读取若干个mapper的shuffle输出文件，然后读取B表partition 0的数据做join。最后，我们将N个任务join的结果通过Union操作合并起来。为了实现这样的处理，我们对shuffle read的接口也做了改变，允许它只读取部分mapper中某一个partition的数据。在这样的处理中，B表的partition 0会被读取N次，虽然这增加了一定的额外代价，但是通过N个任务处理倾斜数据带来的收益仍然大于这样的代价。如果B表中partition 0也发生倾斜，对于inner join来说我们也可以将B表的partition
0分成若干块，分别与A表的partition
0进行join，最终union起来。但对于其它的join类型例如Left Semi
Join我们暂时不支持将B表的partition 0拆分。

自适应执行和Spark SQL在100TB上的性能比较

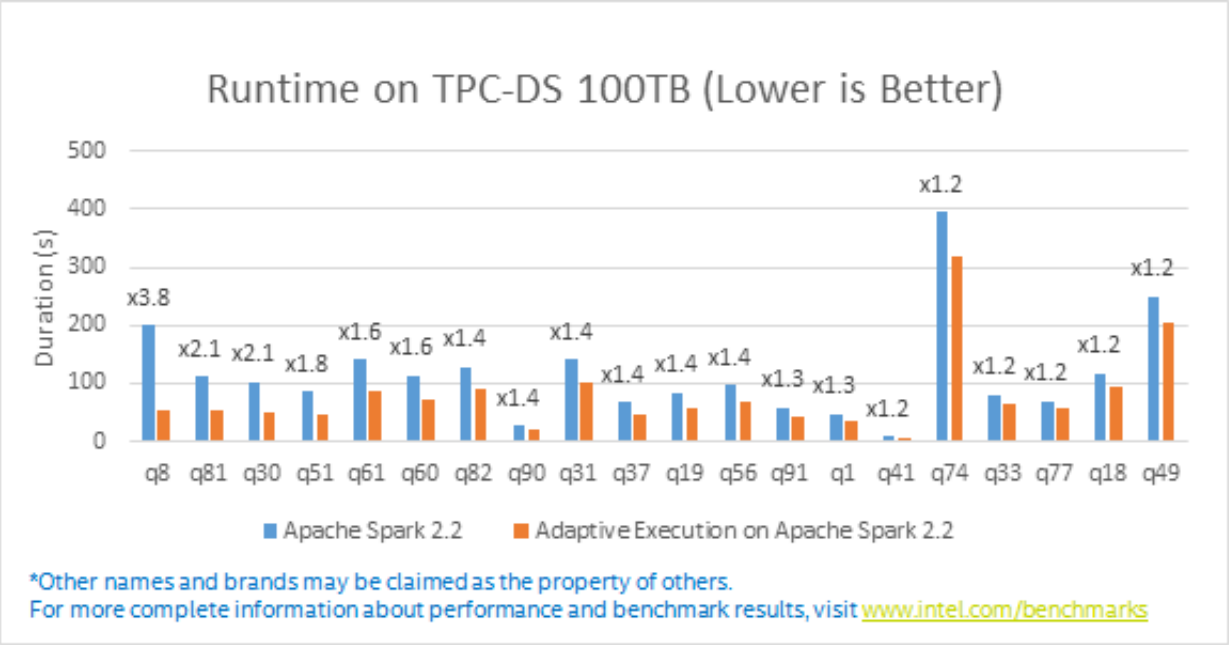
我们使用99台机器搭建了一个集群，使用Spark2.2在TPC-DS
100TB的数据集进行了实验，比较原版Spark和自适应执行的性能。以下是集群的详细信息：

Hardware		BDW
Slave	Node#	98
	CPU	Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz (88 cores)
	Memory	256 GB
	Disk	7× 400 GB SSD
	Network	10 Gigabit Ethernet
Master	CPU	Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz (88 cores)
	Memory	256 GB
	Disk	7× 400 GB SSD
	Network	10 Gigabit Ethernet

实验结果显示，在自适应执行模式下，103条SQL中有92条都得到了明显的性能提升，其中47条SQL的性能提升超过10%，最大的性能提升达到了3.8倍，并且没有出现性能下降的情况。另外在原版Spark中，有5条SQL因为OOM等原因无法顺利运行，在自适应模式下我们也对这些问题做了优化，使得103条SQL在TPC-DS
100TB数据集上全部成功运行。以下是具体的性能提升比例和性能提升最明显的几条SQL。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

通过仔细分析了这些性能提升的SQL，我们可以看到自适应执行带来的好处。首先是自动设置reducer个数，原版Spark使用10976作为shuffle partition数，在自适应执行时，以下SQL的reducer个数自动调整为1064和1079，可以明显看到执行时间上也提升了很多。这正是因为减少了调度的负担和任务启动的时间，以及减少了磁盘IO请求。

原版Spark：

WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details	2017/10/17 11:31:01	18 s	10976/10976			6.9 GB	
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details	2017/10/17 11:30:52	10 s	10976/10976			17.0 GB	6.8 GB

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

自适应执行：

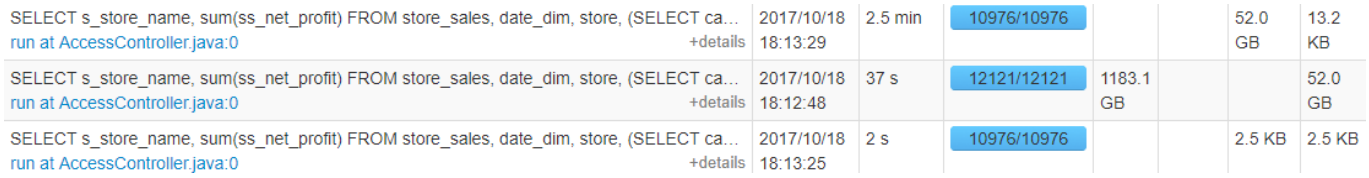
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details	2017/10/18 04:17:22	4 s	1079/1079			5.3 GB	
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at Executors.java:511 +details	2017/10/18 04:17:14	7 s	1084/1084			17.7 GB	5.2 GB

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

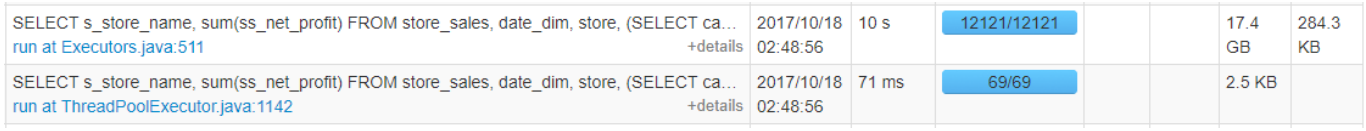
在运行时动态调整执行计划，将SortMergeJoin转化成BroadcastHashJoin在某些SQL中也带来了

很大的提升。例如在以下的例子中，原本使用SortMergeJoin因为数据倾斜等问题花费了2.5分钟。在自适应执行时，因为其中一张表的大小只有2.5k所以在运行时转化成了BroadcastHashJoin，执行时间缩短为10秒。

原版Spark：



自适应执行：



100 TB的挑战及优化

成功运行TPC-DS 100 TB数据集中的所有SQL，对于Apache Spark来说也是一大挑战。虽然Spark SQL官方表示支持TPC-DS所有的SQL，但这是基于小数据集。在100TB这个量级上，Spark暴露出了一些问题导致有些SQL执行效率不高，甚至无法顺利执行。在做实验的过程中，我们在自适应执行框架的基础上，对Spark也做了其它的优化改进，来确保所有SQL在100TB数据集上可以成功运行。以下是一些典型的问题。

统计map端输出数据时driver单点瓶颈的优化（SPARK-22537）

在每个map任务结束后，会有一个表示每个partition大小的数据结构（即下面提到的CompressedMapStatus或HighlyCompressedMapStatus）返回给driver。而在自适应执行中，当一次shuffle的map stage结束后，driver会聚合每个mapper给出的partition大小信息，得到在各个partition上所有mapper输出的数据总大小。该统计由单线程完成，如果mapper的数量是M，shuffle partition的数量为S，那么统计的时间复杂度在O(M x S) ~ O (M x S x log(M x S)) 之间，当CompressedMapStatus被使用时，复杂度为这个区间的下限，当HighlyCompressedMapStatus被使用时，空间有所节省，时间会更长，在几乎所有的partition数据都为空时，复杂度会接近该区间的上限。

在M x S增大时，我们会遇到driver上的单点瓶颈，一个明显的表现是UI上map stage和reduce stage之间的停顿。为了解决这个单点瓶颈，我们将任务尽量均匀地划分给多个线程，线程之间不相交地为scala Array中的不同元素赋聚合值。

在这项优化中，新的spark.shuffle.mapOutput.parallelAggregationThreshold（简称threshold）被引入，用于配置使用多线程聚合的阈值，聚合的并行度由JVM中可用core数和M * S / threshold + 1中的小值决定。

Shuffle读取连续partition时的优化（SPARK-9853）

在自适应执行的模式下，一个reducer可能会从一个mapoutput文件中读取诺干个连续的数据块。目前的实现中，它需要拆分成许多独立的getBlockData调用，每次调用分别从硬盘读取一小块数据，这样就需要很多的磁盘IO。我们对这样的场景做了优化，使得Spark可以一次性地把这些连续数据块都读上来，这样就大大减少了磁盘的IO。在小的基准测试程序中，我们发现shuffle read的性能可以提升3倍。

BroadcastHashJoin中避免不必要的partition读的优化

自适应执行可以为现有的operator提供更多优化的可能。在SortMergeJoin中有一个基本的设计：每个reducetask会先读取左表中的记录，如果左表的
partition为空，则右表中的数据我们无需关注（对于非anti join的情况），这样的设计在左表有一些partition为空时可以节省不必要的右表读取，在SortMergeJoin中这样的实现很自然。

BroadcastHashJoin中不存在按照join key分区的过程，所以缺失了这项优化。然而在自适应执行的一些情况中，利用stage间的精确统计信息，我们可以找回这项优化：如果SortMergeJoin在运行时被转换成了BroadcastHashJoin，且我们能得到各个partition key对应partition的精确大小，则新转换成的BroadcastHashJoin将被告知：无需去读那些小表中为空的partition，因为不会join出任何结果。

Baidu真实产品线试用情况

我们将自适应执行优化应用在Baidu内部基于Spark SQL的即席查询服务BaiduBig SQL之上，做了进一步的落地验证，通过选取单日全天真实用户查询，按照原有执行顺序回放重跑和分析，得到如下几点结论：

- 对于秒级的简单查询，自适应版本的性能提升并不明显，这主要是因为它们的瓶颈和主要耗时集中在了IO上面，而这不是自适应执行的优化点。
- 按照查询复杂度维度考量测试结果发现：查询中迭代次数越多，多表join场景越复杂的情况下自适应执行效果越好。我们简单按照group by, sort, join, 子查询等操作个数来将查询分类，如上关键词大于3的查询有明显的性能提升，优化比从50%~200%不等，主要优化点来源于shuffle的动态并发数调整及join优化。
- 从业务使用角度来分析，前文所述SortMergeJoin转BroadcastHashJoin的优化在Big SQL场景中命中了多种典型的业务SQL模板，试考虑如下计算需求：用户期望从两张不同维度的计费信息中捞取感兴趣的user列表在两个维度的整体计费。收入信息原表大小在百T级别，用户列表只包含对应用户的元信息，大小在10M以内。两张计费信息表字段基本一致，所以我们将两张表与用户列表做inner join后union做进一步分析，SQL表达如下：

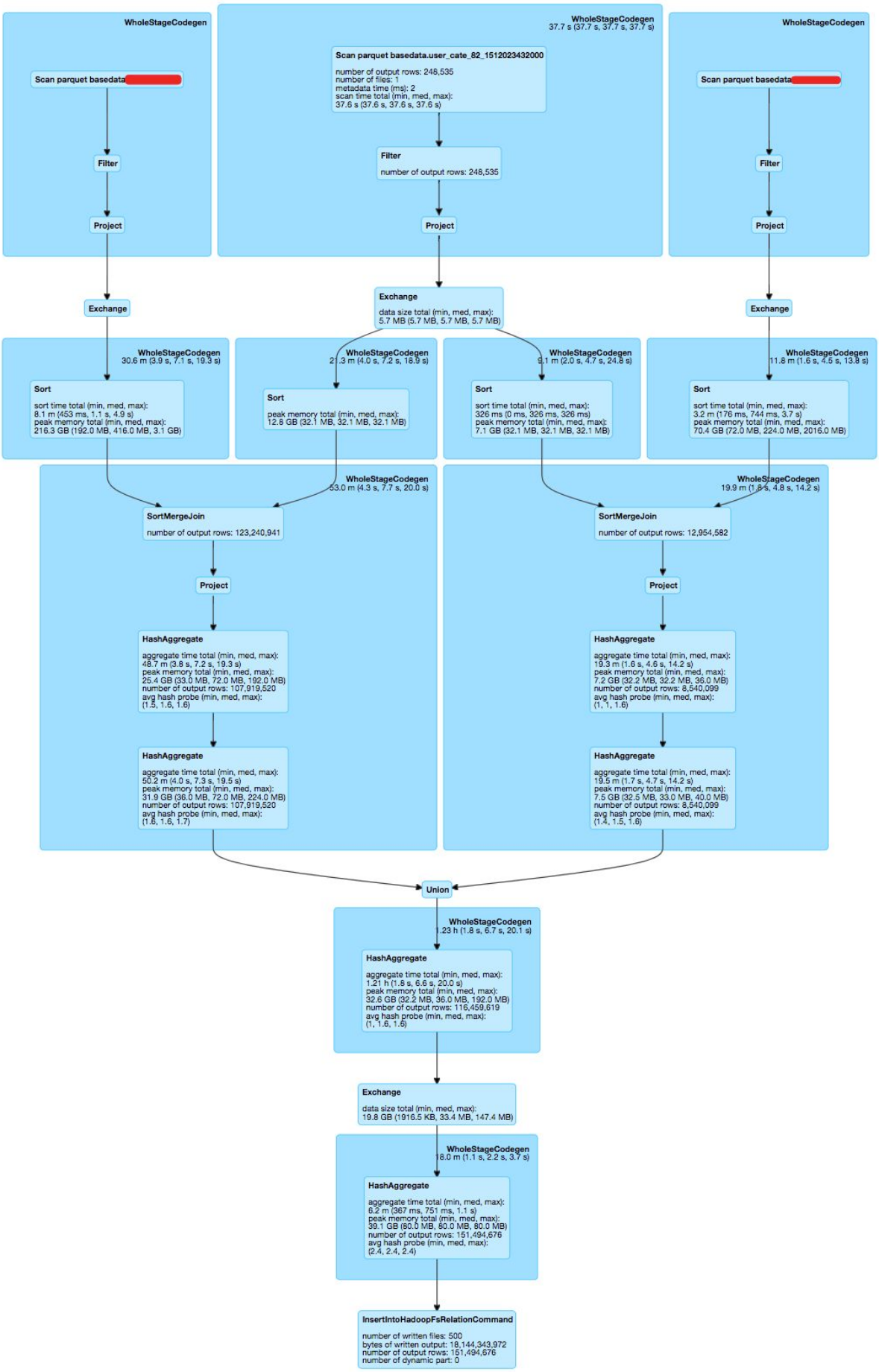
  ```
  select t.c1, t.id, t.c2, t.c3, t.c4,  sum(t.num1), sum(t.num2), sum(t.num3) from (
     select  c1, t1.id as id, c2, c3, c4, sum(num1s) as num1,
  ```

```
    sum(num2) as num2, sum(num3)  as num3
   from basedata.shitu_a t1
   INNER JOIN basedata.user_82_1512023432000 t2
   ON (t1.id = t2.id)
   where  (event_day=20171107)  and flag !=  'true'
   group by c1, t1.id, c2, c3, c4

 union  all

 select  c1, t1.id as id, c2, c3, c4, sum(num1s) as num1,
  sum(num2) as num2, sum(num3)  as num3
 from basedata.shitu_b t1
 INNER JOIN basedata.user_82_1512023432000 t2
 ON (t1.id = t2.id)
 where  (event_day=20171107)  and flag !=  'true'
 group by c1, t1.id, c2, c3, c4
) t group by t.c1, t.id, t.c2, t.c3, c4
```

对应的原版Spark执行计划如下：


WholeStageCodegen
Scan parquet basedata
Filter
Project
Exchange
WholeStageCodegen
37.7 s (37.7 s, 37.7 s, 37.7 s)
Scan parquet basedata.user_cate_82_1512023432000
number of output rows: 248,535
number of files: 1
metadata time (ms): 2
scan time total (min, med, max):
37.6 s (37.6 s, 37.6 s, 37.6 s)
Filter
number of output rows: 248,535
Project
Exchange
data size total (min, med, max):
5.7 MB (5.7 MB, 5.7 MB, 5.7 MB)
WholeStageCodegen
Scan parquet basedata
Filter
Project
Exchange
WholeStageCodegen
30.6 m (3.9 s, 7.1 s, 19.3 s)
Sort
sort time total (min, med, max):
8.1 m (453 ms, 1.1 s, 4.9 s)
peak memory total (min, med, max):
216.3 GB (192.0 MB, 416.0 MB, 3.1 GB)
WholeStageCodegen
21.3 m (4.0 s, 7.2 s, 18.9 s)
Sort
peak memory total (min, med, max):
12.8 GB (32.1 MB, 32.1 MB, 32.1 MB)
WholeStageCodegen
9.1 m (2.0 s, 4.7 s, 24.8 s)
Sort
sort time total (min, med, max):
326 ms (0 ms, 326 ms, 326 ms)
peak memory total (min, med, max):
7.1 GB (32.1 MB, 32.1 MB, 32.1 MB)
WholeStageCodegen
11.8 m (1.6 s, 4.5 s, 13.8 s)
Sort
sort time total (min, med, max):
3.2 m (176 ms, 744 ms, 3.7 s)
peak memory total (min, med, max):
70.4 GB (72.0 MB, 224.0 MB, 2016.0 MB)
WholeStageCodegen
53.0 m (4.3 s, 7.7 s, 20.0 s)
SortMergeJoin
number of output rows: 123,240,941
Project
HashAggregate
aggregate time total (min, med, max):
48.7 m (3.8 s, 7.2 s, 19.3 s)
peak memory total (min, med, max):
25.4 GB (33.0 MB, 72.0 MB, 192.0 MB)
number of output rows: 107,919,520
avg hash probe (min, med, max):
(1.5, 1.6, 1.6)
HashAggregate
aggregate time total (min, med, max):
50.2 m (4.0 s, 7.3 s, 19.5 s)
peak memory total (min, med, max):
31.9 GB (36.0 MB, 72.0 MB, 224.0 MB)
number of output rows: 107,919,520
avg hash probe (min, med, max):
(1.6, 1.6, 1.7)
WholeStageCodegen
19.9 m (1.8 s, 4.8 s, 14.2 s)
SortMergeJoin
number of output rows: 12,954,582
Project
HashAggregate
aggregate time total (min, med, max):
19.3 m (1.6 s, 4.6 s, 14.2 s)
peak memory total (min, med, max):
7.2 GB (32.2 MB, 32.2 MB, 36.0 MB)
number of output rows: 8,540,099
avg hash probe (min, med, max):
(1, 1, 1.6)
HashAggregate
aggregate time total (min, med, max):
19.5 m (1.7 s, 4.7 s, 14.2 s)
peak memory total (min, med, max):
7.5 GB (32.5 MB, 33.0 MB, 40.0 MB)
number of output rows: 8,540,099
avg hash probe (min, med, max):
(1.4, 1.5, 1.6)
Union
WholeStageCodegen
1.23 h (1.8 s, 6.7 s, 20.1 s)
HashAggregate
aggregate time total (min, med, max):
1.21 h (1.8 s, 6.6 s, 20.0 s)
peak memory total (min, med, max):
32.6 GB (32.2 MB, 36.0 MB, 192.0 MB)
number of output rows: 116,459,619
avg hash probe (min, med, max):
(1, 1.6, 1.6)
Exchange
data size total (min, med, max):
19.8 GB (1916.5 KB, 33.4 MB, 147.4 MB)
WholeStageCodegen
18.0 m (1.1 s, 2.2 s, 3.7 s)
HashAggregate
aggregate time total (min, med, max):
6.2 m (367 ms, 751 ms, 1.1 s)
peak memory total (min, med, max):
39.1 GB (80.0 MB, 80.0 MB, 80.0 MB)
number of output rows: 151,494,676
avg hash probe (min, med, max):
(2.4, 2.4, 2.4)
InsertIntoHadoopFsRelationCommand
number of written files: 500
bytes of written output: 18,144,343,972
number of output rows: 151,494,676
number of dynamic part: 0

如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

针对于此类用户场景，可以全部命中自适应执行的join优化逻辑，执行过程中多次SortMergeJoin转为BroadcastHashJoin，减少了中间内存消耗及多轮sort，得到了近200%的性能提升。

结合上述3点，下一步自适应执行在Baidu内部的优化落地工作将进一步集中在大数据量、复杂查询的例行批量作业之上，并考虑与用户查询复杂度关联进行动态的开关控制。对于数千台的大规模集群上运行的复杂查询，自适应执行可以动态调整计算过程中的并行度，可以帮助大幅提升集群的资源利用率。另外，自适应执行可以获取到多轮stage之间更完整的统计信息，下一步我们也考虑将对应数据及Strategy接口开放给Baidu
Spark平台上层用户，针对特殊作业进行进一步的定制化Strategy策略编写。

总结

随着Spark SQL广泛的使用以及业务规模的不断增长，在大规模数据集上遇到的易用性和性能方面的挑战将日益明显。本文讨论了三个典型的问题，包括调整shuffle partition数量，选择最佳执行计划和数据倾斜。这些问题在现有的框架下并不容易解决，而自适应执行可以很好地应对这些问题。我们介绍了自适应执行的基本架构以及解决这些问题的具体方法。最后我们在TPC-DS 100TB数据集上验证了自适应执行的优势，相比较原版Spark SQL，103个SQL查询中，90%的查询都得到了明显的性能提升，最大的提升达到3.8倍，并且原先失败的5个查询在自适应执行下也顺利完成。我们在百度的Big SQL平台也做了进一步的验证，对于复杂的真实查询可以达到2倍的性能提升。总之，自适应执行解决了Spark
SQL在大数据规模上遇到的很多挑战，并且很大程度上改善了Spark SQL的易用性和性能，提高了超大集群中多租户多并发作业情况下集群的资源利用率。将来，我们考虑在自适应执行的框架之下，提供更多运行时可以优化的策略，并且将我们的工作贡献回馈给社区，也希望有更多的朋友可以参与进来，将其进一步完善。

本文转自：《Spark SQL在100TB上的自适应执行实践》