Apache Spark SQL adaptive execution practices

Author: Zhou Yu Wang Yucai more Guo Chen Zhao Cheng Hao (Intel), Liyuan Jian (Baidu)

Spark SQL is a component of Apache Spark most widely used, it provides a very friendly interface for distributed processing structured data, many applications have successful production practices, but on very large scale clusters and data sets, Spark SQL many are still challenged ease of use and scalability. To address these challenges, Intel's big data technology team and Baidu large data infrastructure Engineer at Spark

On the basis of community version, improve and implement adaptive execution engine. This paper discusses the challenges Spark SQL encountered on large data sets, and then introduce the background and the basic architecture of the adaptive execution, as well as how to deal with Spark SQL adaptive execution of these issues, we will compare the final implementation and adaptation of existing Community challenges and difference in performance in the 100 TB version Spark SQL scale TPC- DS benchmark encountered, as well as the implementation of adaptive use Baidu Big SQL platform.

Challenge 1: About the number of shuffle partition

In the Spark SQL, shufflepartition number may be set by the parameter spark.sql.shuffle.partition, the default value is 200. This parameter determines the number of stages each reduce task SQL job, have a great impact on the overall query performance. Suppose a query before running the application of the E Executor, each Executor contains C core (concurrent execution threads), then the number of tasks the job at runtime can be executed in parallel is equal to E x C a, or the concurrent operation the number is E x C. Suppose the number of shuffle partition is P, in addition to map

And the file number of tasks stage number and the size of the raw data related to each reduce subsequent
The number of tasks stage is P. Since Spark Job scheduling is preemptive, E x C concurrent task execution unit P performs tasks will be preempted, "ADM" until all tasks have completed, the process proceeds to the next Stage. But this process, if there are large amount of data processing because the task (e.g.: Data inclined cause massive data into the same reducer partition) or other causes of execution time for the task, on the one hand causes the entire execution time becomes stage long, on the other hand E x C concurrent execution units are idle most likely wait state, a sharp decline in overall utilization of cluster resources.

So spark.sql.shuffle.partition parameters How much is appropriate? If the setting is too small, the more the amount of data assigned to each reduce task processing, memory size in limited circumstances, had to write overflow (spill) to the local disk on the compute nodes. Spill causes additional disk read and write, affect the overall performance of SQL queries, worse situation may also lead to serious problems and even GC OOM. On the contrary, if the shuffle partition is set too high. First, reduce the amount of data of each task processing is small and end soon, leading to Spark scheduling burden becomes large. Second, each task must put his mapper s huffle P output data into a hash bucket, i.e. determines which data belongs to reduce partition, when too many shuffle partition, hash bucket in the amount of data is small, the number of concurrent jobs in when a large, reduce task shuffle pull data will cause a certain degree of small random data read operation, when the use of mechanical hard disk as temporary when the shuffle data access performance degradation will be more apparent. Finally, when the last stage to save data files will be written P, it may also cause HDFS file system in a large number of small files.

From, shuffle partition set neither too small nor too big. In order to achieve the best performance, after numerous tests often require a SQL query to determine the best value shuffle partition. However, in a production environment, often by way of the timing of SQL jobs processing data from different time periods, the amount of data size may vary widely, we can not do the time-consuming queries manually tuning for each SQL, this also means that these SQL work hard to run at peak performance mode.

Another problem Shuffle partition is the same number shuffle partition settings will apply to all of the stage. Spark in the implementation of a SQL job will be divided into multiple stage. Typically, data distribution and size of each stage is probably not the same, global shuffle

partition set up to only one or some of the best stage, there is no way to do all the global stage to set the optimum.

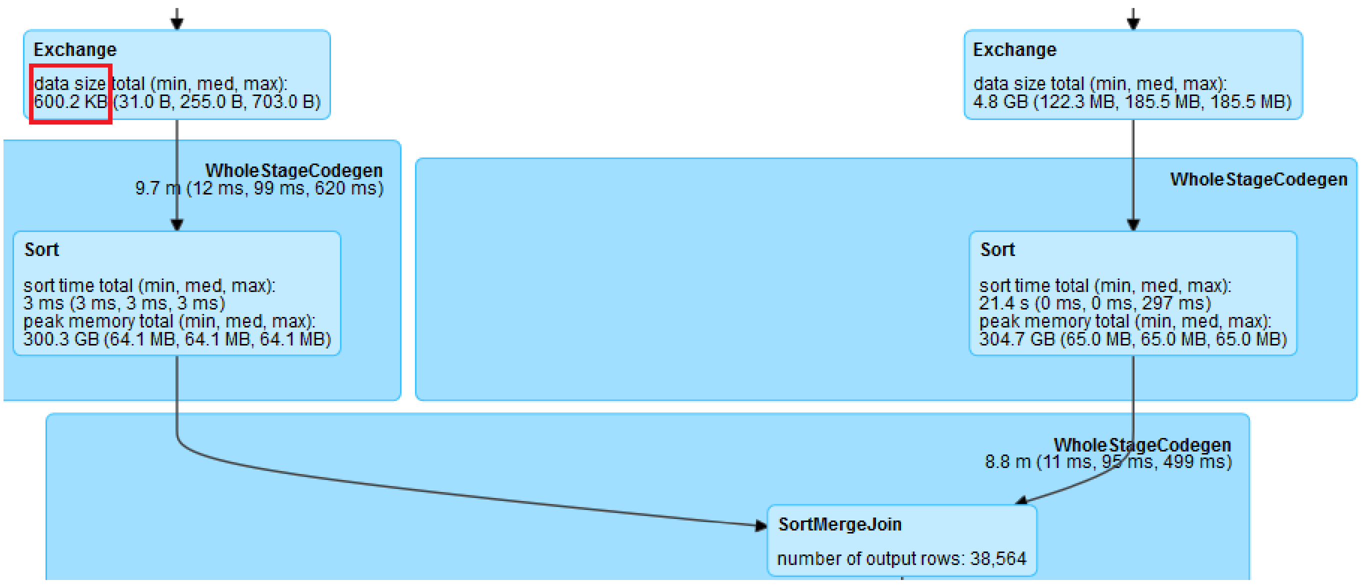
This series of challenges with regard to performance and ease of use shufflepartition, prompting us to think about new ways: shuffle the amount of information we can obtain data according to the runtime, such as block size, record the number of lines, etc., automatically set for each stage suitable sh uffle partition values?

Challenge 2: Spark SQL best execution plan

Spark SQL before executing SQL, SQL or Dataset program will be parsed into logical plan, then go through a series of optimization, physical finalize an executable program. Different physical properties of the final plan chosen has a great influence. How to choose the best execution plan, which is the Spark

The core work of Catalyst SQL optimizer. Early Catalyst primarily rule-based optimizer (RBO), the Spark 2.2 also joined the cost-based optimization (CBO). Determine the implementation plan is currently in the planning phase, once the confirmation will not change. However, during operation, when we get to the more run-time information, we will likely get a better execution plan.

To join the operation, for example, the Spark is the most common strategy is to BroadcastHashJoin and SortMergeJoin. BroadcastHas hJoin belonging map side join, wherein when the principle is that when a broadcast table memory size is less than the threshold value, the Spark Zhang select these tables broadcast to each of the Executor, and the phase map, a read points each mapper large table sheet, and the entire small table and join, the whole process avoids the large table of data is performed in cluster shuffle. Shuffle performed while SortMergeJoin are written in the same manner as in the partition map phase two data tables, reduce the two stage reducer of each table corresponding to data belonging to the same partition pulling a task to do join. RBO The size of the data, as the join operations optimized to BroadcastHashJoin. Spark spark.sql.autoBroadcastJoinThreshold parameters used to control the selection threshold Broadc astHashJoin default is 10MB. However, for complex SQL queries, it is possible to use intermediate results as join input in the planning phase, Spark does not exactly know the size join the two tables or incorrectly estimate their size, so that missed use BroadcastHashJoin strategy to optimize the execution of the opportunity to join. However, at run time, through information obtained from the shuffle to write, we can dynamically choose BroadcastHashJoin. The following is an example of the size of the input side of the join is only 600K, but Spark is still planning to SortMergeJoin.


Exchange
data size total (min, med, max):
600.2 KB (31.0 B, 255.0 B, 703.0 B)
Exchange
data size total (min, med, max):
4.8 GB (122.3 MB, 185.5 MB, 185.5 MB)
WholeStageCodegen
9.7 m (12 ms, 99 ms, 620 ms)
WholeStageCodegen
Sort
sort time total (min, med, max):
3 ms (3 ms, 3 ms, 3 ms)
peak memory total (min, med, max):
300.3 GB (64.1 MB, 64.1 MB, 64.1 MB)
Sort
sort time total (min, med, max):
21.4 s (0 ms, 0 ms, 297 ms)
peak memory total (min, med, max):
304.7 GB (65.0 MB, 65.0 MB, 65.0 MB)
WholeStageCodegen
8.8 m (11 ms, 95 ms, 499 ms)
SortMergeJoin
number of output rows: 38,564

If you want a timely manner

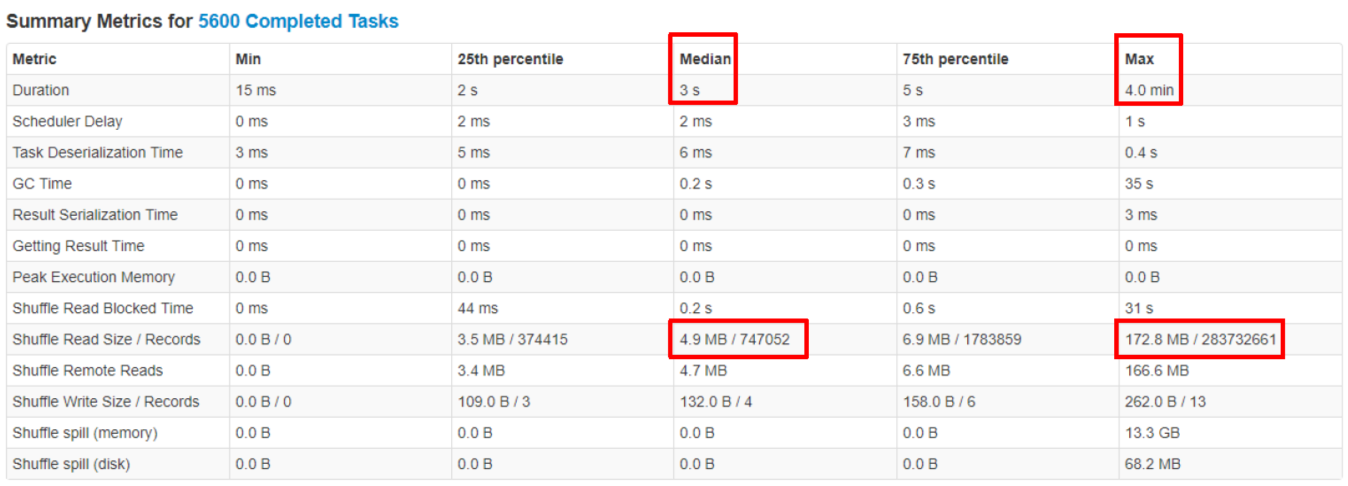
Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

This prompted us to think about the second question: Can we pass the information collected runtime to dynamically adjust the implementation plan?

Challenge 3: Data tilt

Data skew is a common cause of poor performance Spark SQL problem. Inclined amount data refers to data of a certain partition of a partition is much larger than the other data, resulting in individual tasks running time is far greater than other tasks, thus dragging down SQL runtime. SQL job in practice, the tilt data is common, join key corresponding to the hash

bucket will always be less the average number of records of the case, in extreme cases, the same join key corresponding to the number of records particularly large, large amount of data is bound to be assigned to the same partition resulting in serious data skew. 2, can be seen most about 3 seconds to complete the task, while the task of the slowest it took 4 minutes, the amount of data that is several times the processing other tasks.



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

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Currently, some of the common data means join inclined treatment: (1) increasing the number of shuffle partition, desirable to partition the original points of the same data can be dispersed into the plurality of partition, but no effect on the data of the same key. (2) adjusting the threshold value Bro adcastHashJoin, in some scenarios can be converted into BroadcastHashJoin SortMergeJoin avoid data generated shu ffle inclined. (3) manually tilted filter key, and the random data added to the prefix, in another table corresponding to the key data a corresponding expansion process, then do join. Taken together, these instruments have their limitations and to process a lot of people. Based on this, we think the third question: Spark Can run automatically process data join the tilt?

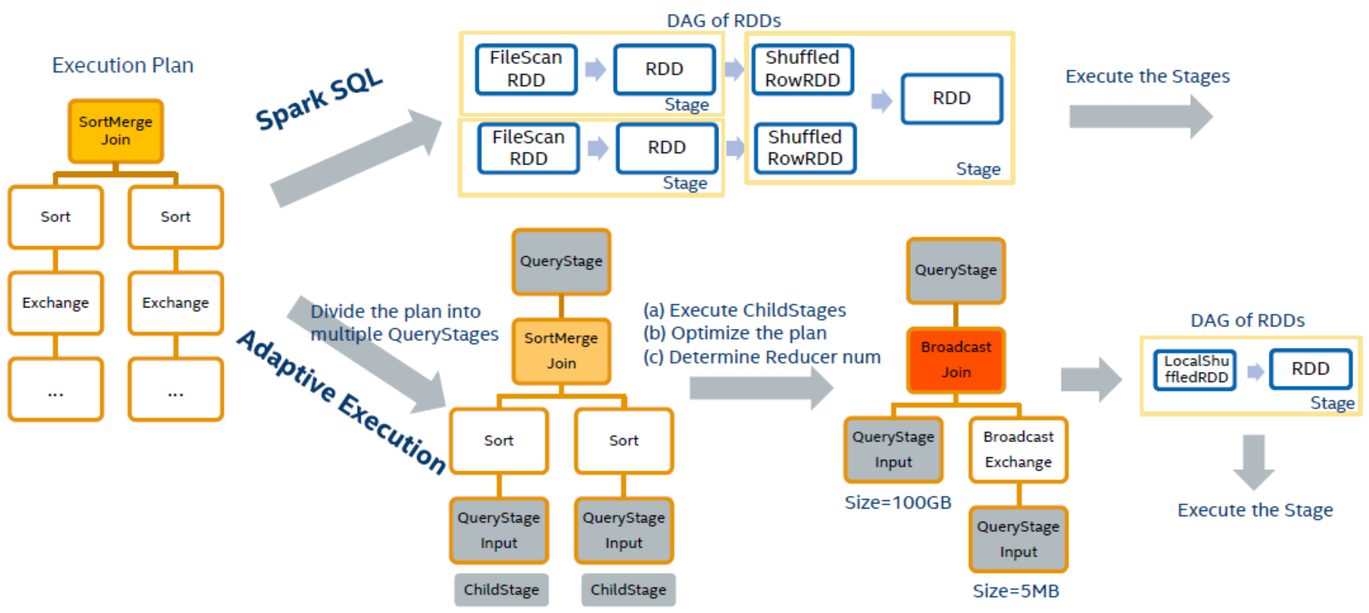
Introduction and implementation of adaptive background

As early as 2015, Spark community proposed the basic idea of adaptive execution, increasing the interface to submit a single map stage in DAGScheduler Spark of, and adjustment to achieve run-time shuffle partition on the number of attempts made. But this implementation has some limitations, in some scenarios will introduce more shuffle, namely more stage, for a three-table join and so on in the same situation to do a stag e is also not a good deal. So this feature has been in the experimental stage, the configuration parameters are not mentioned in the official documents.

Based on the work of these communities, Intel's big data technology team to do a re-implementation of adaptive design to achieve a more flexible self-adaptive execution framework. In this framework the following, we can add additional rules to achieve more. Currently, the properties have been implemented include: shuffle automatically set the number of partition, dynamic adjustment of the implementation plan, dynamic data processing tilt and so on.

Adaptive execution architecture

In the SQL Spark, after the last physical Spark execution plan is determined, according to each of the conversion operator definition of RDD, it generates a DAG RDD of FIG. After the DAG-based Spark static partitioning stage and submitted to the Executive, so once the implementation plan is determined, we can no longer update the operational phase. The basic idea is to perform adaptive pre-division stage in the implementation of the plan is good, and then presented to the press stage, the current stage of the shuffle collect statistics at runtime, in order to optimize the next stage of the implementation plan before submitting subsequent execution stage.



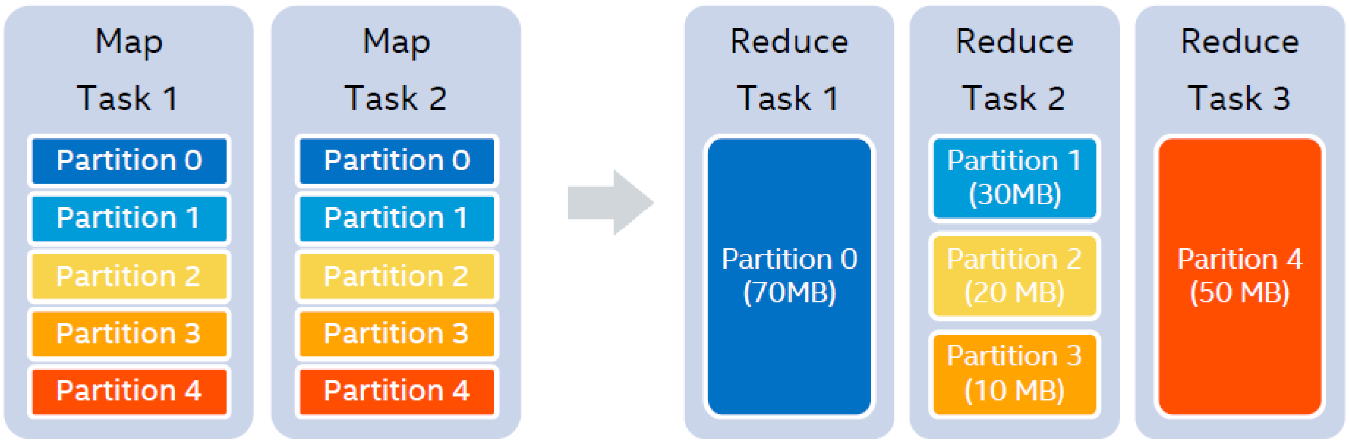
If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

From Figure 3 we can see that the implementation of adaptive methods of work, firstly as a boundary node Exchange will perform the program tree into multiple QueryStage (Exchange node represents a shuffle in the Spark SQL). Each QueryStage is a separate sub-tree, is an independent execution units. While adding QueryStage, we also add a leaf node QuerySta geInput as a father QueryStage input. For example, the execution plan table join the two figures is that we will create three QueryStage. The last execution plan is in QueryStage join itself, it has two QueryStageInput on behalf of its input, pointing to two children QueryStage. When performing QueryStage, we first submitted its children s tage, and collect information on these stage operation. When these kids stage has finished running, we can know their size and other information, in order to determine whether the plan can be optimized QueryStage update. For example, when we know the size of a table is a 5M, it is less than the threshold value broadcast, we can be converted into BroadcastHashJoin SortMergeJoin to optimize the current implementation plan. We can also shuffle the amount of data generated by the child stage, to dynamically adjust the number of reduc er of the stage. After completing a series of optimization, and ultimately we generate the DAG RDD for the QueryStage, and submitted to the DAG Scheduler to perform.

Automatically set the number reducer

Suppose we set shufflepartition number 5, after the map stage, we know the size of each partition are 70MB, 30MB, 20MB, 10MB and 50MB. Suppose we set the target amount of data processed each reducer is 64MB, then at runtime, we can actually use three reducer. Processing a first reducer partition 0 (70MB), the second processing reducer consecutive partition 1 to 3 were 60MB, the third processing reducer partition 4 (50MB), as shown in FIG.



If you want a timely manner

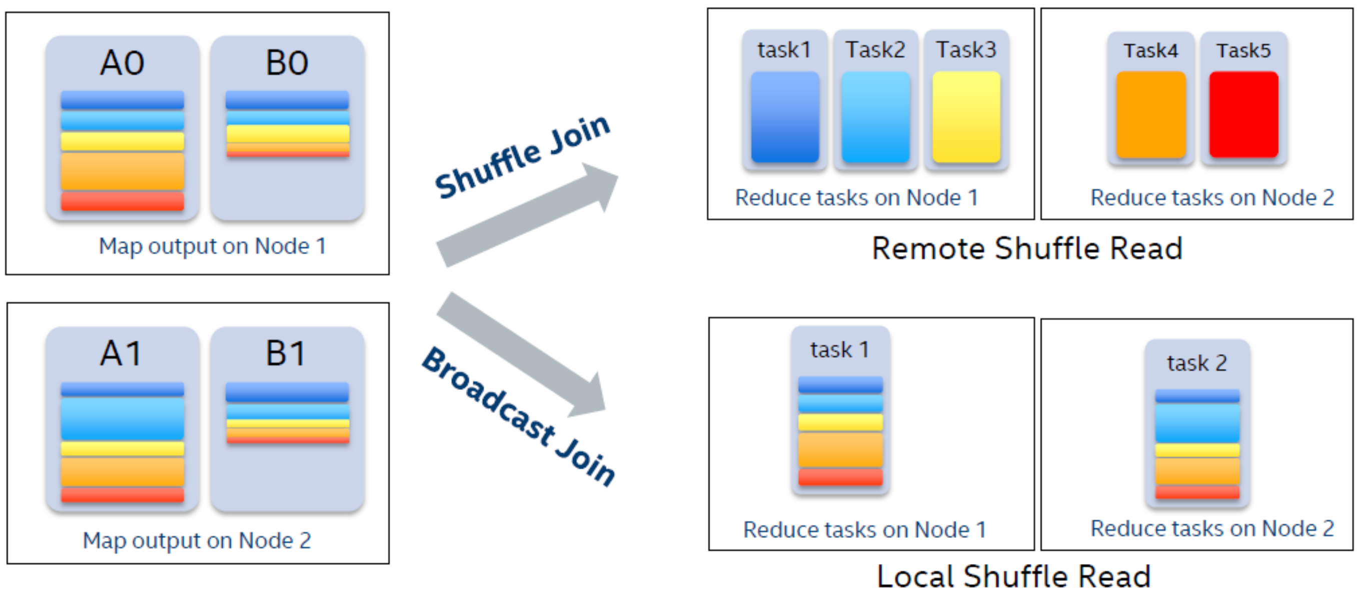
Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

In the framework of the implementation of adaptation, since each QueryStage knows all his children stage, so when adjusting the number of reducer, you can take into account all of the input stage. In addition, we can also record the number of the target as a reducer process. Because shuffle data are often compressed, and sometimes the amount of data partition is not large, but extracting the number of records determined partition is much larger than the other, resulting in uneven data. Therefore, considering the data size and the number of records may be better educer determined number of r.

Dynamic adjustment of the implementation plan

We currently support in run-time dynamic adjustment join strategy, in the case of satisfying the condition that a table is less than the threshold Broadcast, SortMergeJoin can be converted into BroadcastHashJoin. Since the partition and the case SortMergeJoin BroadcastHashJoin output is not the same, the next free conversion stage may introduce additional shuffle operation. Therefore, we join in the dynamic adjustment strategy, follow a rule that was converted without the introduction of additional shuffle.

The SortMergeJoin converted into BroadcastHashJoin What good is it? Because the data has been written to disk shuffle, shuffle we still need to read the data. We can look at the example of FIG. 5, assuming the Join Table A and Table B, Table 2 Stage map map tasks have two, and the number of shuffle partition 5. If you do SortMergeJoin, need to start five reducer reduce in stages, each reducer read their own data over the network shuffle. However, it can be found in Table B Broadcast, and then converts it into BroadcastHashJoin, we only need to start two reducer, a reducer each shuffle output read the entire file when a mapper we run. When we scheduled reducer these two tasks can be scheduled on a priority Executor mapper run, the entire shuffle read into a local reading, no data is transmitted over the network. And reads a file read in this order, a random shuffle small when compared to the original document reading, efficiency is superior. In addition, SortMergeJoin process often will be varying degrees of data skew problem, slow down the overall running time. And after conversion to BroadcastHashJoin, the amount of data is generally more uniform, thus avoiding the inclination, we can see more specific information in the experimental results below.



If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Dynamic processing data skew

In the framework of the adaptive performed, we can easily detect the data when run sloping partition. When performing a s tage, we collect the stage shuffle data size and the number of records of each mapper. If the amount of data a certain number of records or partition than N times the median, and larger than a certain threshold pre-configured, we think that this is a sloping partition data, it requires special handling.

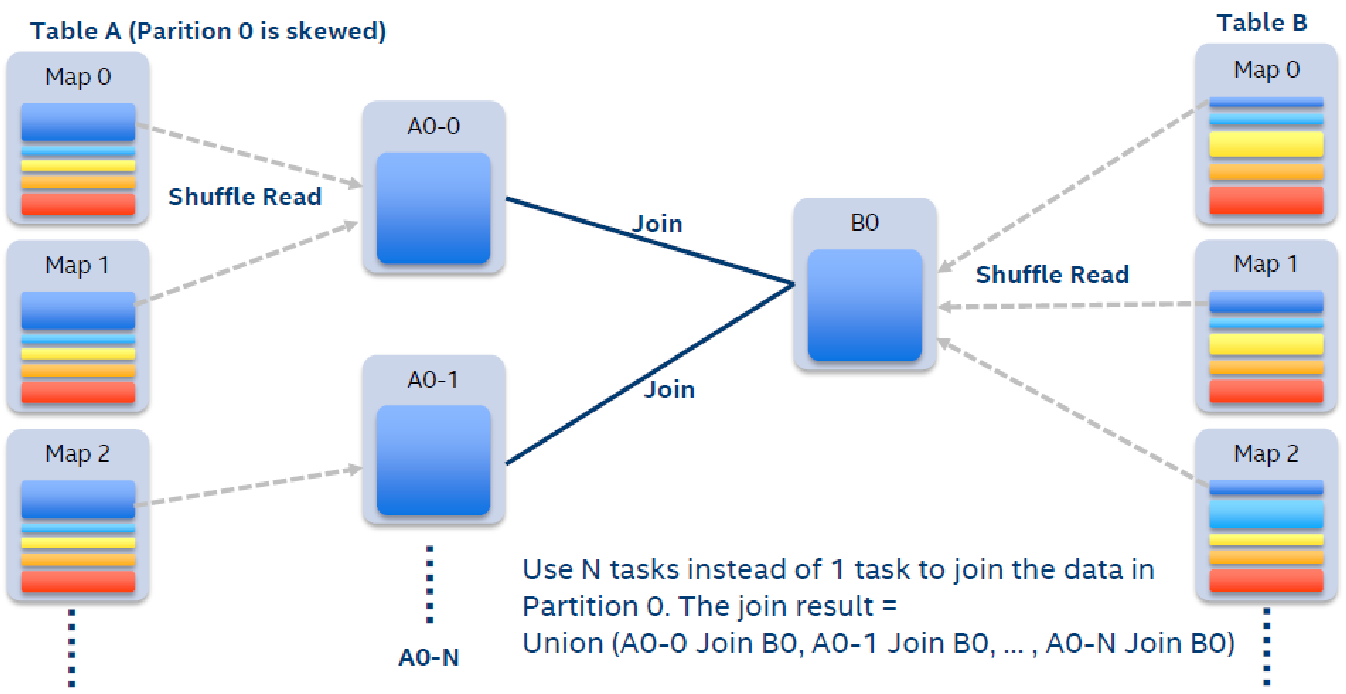


Table A (Parition 0 is skewed)
Map 0
Map 1
Map 2
Shuffle Read
A0-0
A0-1
A0-N
Join
Join
B0
Shuffle Read
Table B
Map 0
Map 1
Map 2
Use N tasks instead of 1 task to join the data in Partition 0. The join result = Union (A0-0 Join B0, A0-1 Join B0, … , A0-N Join B0)

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Suppose we Table A and Table B do inner
join, and Table A is a 0-th partition inclined partition. Generally, A and B tables are tables of data shuffle partition 0 is processed to a reducer of same, since the reducer is necessary to pull a large amount of data across the network and processed, it will be a slow down the overall task of the slowest performance. In the adaptive implementation framework, once we find A partition table

0 tilted, then we use the N tasks to deal with the partition. Each task reads only the number of shuffle mapper output file, and then read the data partition 0 Table B do join. Finally, we will join the results of N tasks by Union merged operation. To implement such a process, we also made shuffle read interface changes, allowing only one data partition section m apper read. In this process, B table partition 0 will be read N times, though it adds a certain extra costs, but revenue from the N-tasking brings tilt data is still greater than this price. If the inclined partition 0 Table B also occur, for inner join is 0 we can partition the table into several blocks B, respectively, and the partition table A

0 of join, the final union together. However, for other types such join Left Semi Join we do not
support partition 0 Table B Split.

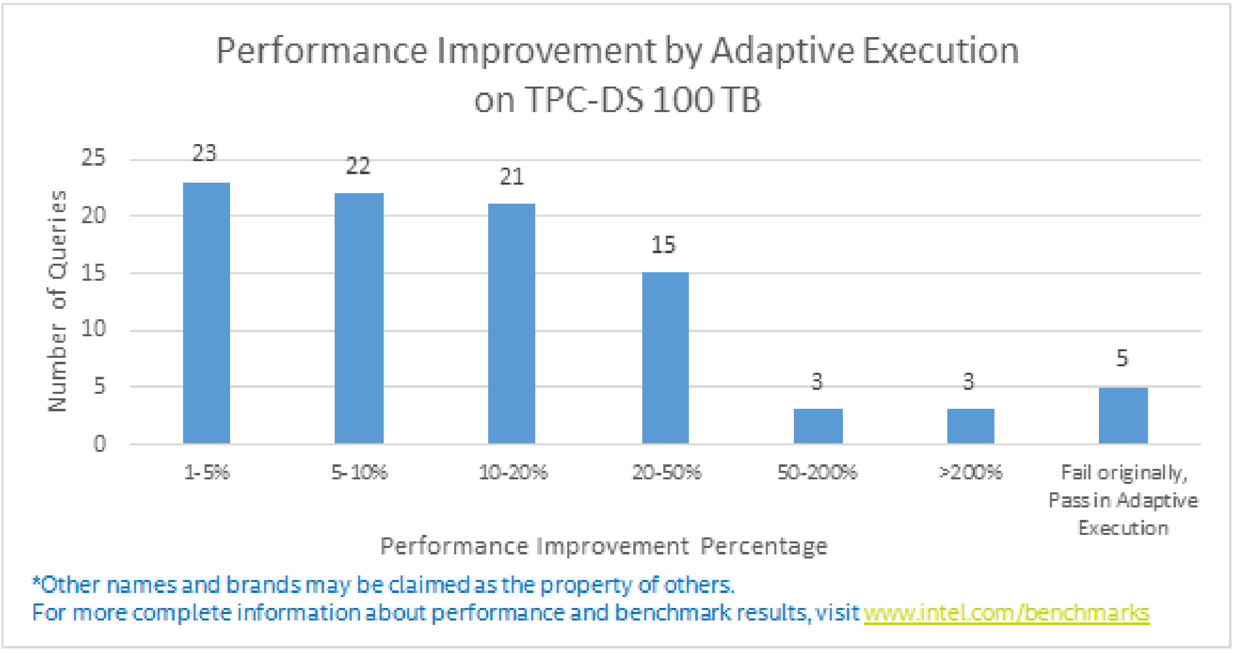
Adaptive Spark SQL execution and performance comparison on the 100TB

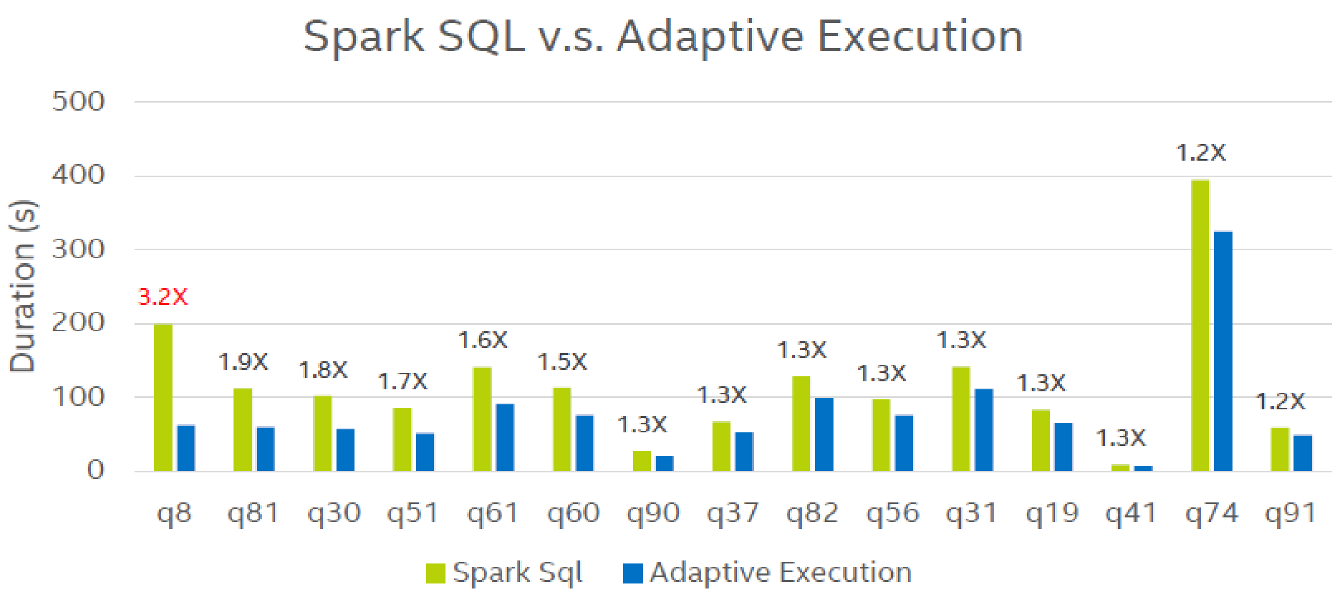
We used to build a 99 machine cluster, using the TPC-DS Spark2.2
100TB experimental data sets, and compare the performance of the original Spark adaptive performed. Here are the details of the cluster:

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
Software		
OS	CentOS* Linux release 6.9	
Kernel	2.6.32-573.22.1.el6.x86_64	
Spark*	Spark* master (2.3) / Spark* master (2.3) with adaptive execution patch	
Hadoop*/HDFS*	hadoop-2.7.3	
JDK	1.8.0_40 (Oracle* Corporation)	

Experimental results show that, in the adaptive execution mode, there are 103 SQL 92 have been significant performance gains, in which 47 S QL performance increase exceeding 10% of the maximum performance reached 3.8 times and no performance degradation. Also in the original Spark, there are five other reasons OOM because SQL can not run smoothly, we are in adaptive mode on these issues has been optimized so that 103 SQL in TPC-DS

100TB of data collection on all run successfully. The following is a specific ratio to enhance performance and improve the performance of the most obvious of several SQL.



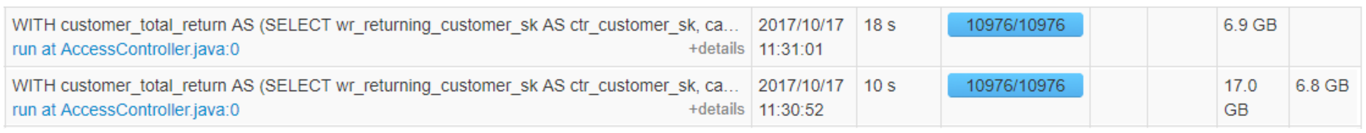


If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

By carefully analyzed the performance of SQL, we can see the benefits brought by the implementation of adaptation. First, the number is automatically set red ucer, using the original Spark Partition number 10976 as shuffle, the adaptive implementation of the following SQL reducer number is automatically adjusted to 1064 and 1079, the execution time can clearly see also improved a lot. This is because of the burden of reducing the time schedule and start the task, and reduces disk IO request.

Original Spark:

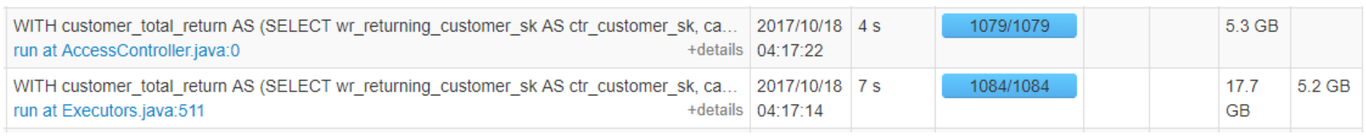


WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details	2017/10/17 11:31:01	18 s	10976/10976			6.9 GB	
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details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

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Adaptive Execution:



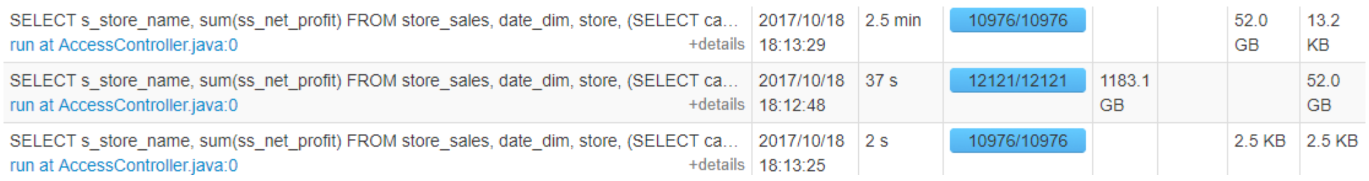
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at AccessController.java:0 +details	2017/10/18 04:17:22	4 s	1079/1079			5.3 GB	
WITH customer_total_return AS (SELECT wr_returning_customer_sk AS ctr_customer_sk, ca... run at Executors.java:511 +details	2017/10/18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

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Dynamic adjustment of the implementation plan at run time, the SortMergeJoin converted into BroadcastHashJoin in some SQL is also a big improvement. For example, in the following examples, since the data originally used SortMergeJoin tilt problem that it takes 2.5 minutes. When the adaptive execution, because the size of a table is only 2.5k so converted at runtime became BroadcastHashJoin, Execution time is shortened to 10 seconds.

Original Spark:

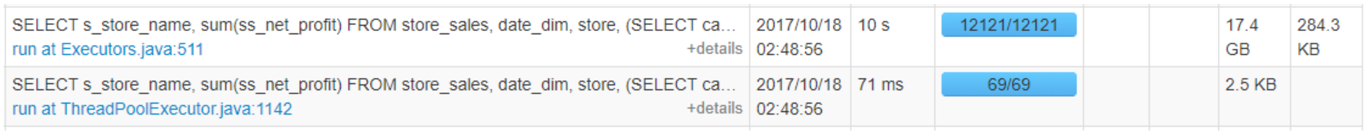


SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ca... run at AccessController.java:0 +details	2017/10/18 18:13:29	2.5 min	10976/10976			52.0 GB	13.2 KB
SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ca... run at AccessController.java:0 +details	2017/10/18 18:12:48	37 s	12121/12121	1183.1 GB			52.0 GB
SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ca... run at AccessController.java:0 +details	2017/10/18 18:13:25	2 s	10976/10976			2.5 KB	2.5 KB

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Adaptive Execution:



SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ca... run at Executors.java:511 +details	2017/10/18 02:48:56	10 s	12121/12121			17.4 GB	284.3 KB
SELECT s_store_name, sum(ss_net_profit) FROM store_sales, date_dim, store, (SELECT ca... run at ThreadPoolExecutor.java:1142 +details	2017/10/18 02:48:56	71 ms	69/69			2.5 KB	

If you want a timely manner

Solution Spark, Hadoop or Hbase related articles, welcome attention to the micro-channel public account: **iteblog_hadoop**

Challenges and optimization of 100 TB

Successful operation of TPC-DS 100 TB data set of all SQL for Apache Spark is also a big challenge. Although SparkS QL official expressed support for the TPC-DS all SQL, but this is based on a small data set. On 100TB of this magnitude, Spark exposed some problems cause some SQL execution efficiency is not high, or even can not be performed smoothly. In the process of doing experiments, we in the framework of the implementation of the adaptation of Spark also made other improvements to optimize, to ensure that all SQL can run successfully on 100TB of data sets. Here are some typical questions.

Statistical map-side driver optimized single bottleneck point (SPARK-22537) output data

Map after each task, there is a partition represents the size of each data structure (i.e., below-mentioned Compressed MapStatus or HighlyCompressedMapStatus) is returned to the driver. In the adaptive implementation, when the end of a shuffle map stage, driver polymerize size information for each partition mapper given, to give a total size of all data on the respective output mapper partition. This is performed at the single thread, if the number of the mapper is M, the number of shuffle partition is S, in O (M x S) ~ O between the (M x S x log (M x S)) then the statistics of the time complexity of when Compre ssedMapStatus is used, the complexity is the lower limit of the interval, when HighlyCompressedMapStatus is used, saving some space, time will be longer, when almost all the partition data are empty, the complexity will be close to the interval The upper limit.

When M x S increases, we will encounter a single point of bottleneck on the driver, an obvious manifestation is the pause between the UI map stage and reduce sta ge. To solve this bottleneck single point, the task will be divided as evenly as possible to the plurality of threads, the threads do not intersect between the value assigned polymerization scala Array different elements.

In this optimization, the new spark.shuffle.mapOutput.parallelAggregationThreshold (referred to as the threshold)

Is introduced, arranged to use multiple threads polymerization threshold, the polymerization degree of parallelism in the JVM and the number of available core M * S / thresho
+ Small value determines 1.

Optimization (SPARK-9853) when reading consecutive partition Shuffle

In the adaptive mode execution, a reducer may read Nogang consecutive data blocks from one file mapoutput. The current implementation, it needs to be split into many separate getBlockData calls, each call to a small piece of data are read from the hard disk, so you need a lot of disk IO. We optimized to such a scenario, Spark can make once these contiguous blocks of data are read up, thus greatly reducing the disk IO. In small benchmarks, we found that the performance can enhance the shuffle read three times.

BroadcastHashJoin avoid unnecessary partition read optimization

Adaptive execution may be possible to provide more optimized for the existing operator. There is a basic design in SortMergeJoin in: Each reducetask will first read the record left the table, if the left table
partition is empty, then the data in the table we do not have the right focus (in the case of non-anti join), this design left some partition table is empty can save unnecessary right table read, in such an implementation in SortMergeJoin very natural.

BroadcastHashJoin does not exist in accordance with the procedure join key partitions, so missing this optimization. However, in some cases, adaptive execution, the use of accurate statistical information between stage, we can retrieve this optimization: is converted to BroadcastHashJoin if SortMergeJoin at run time, and we can get the exact size of each partition key corresponding to the partition, the new convert to BroadcastHashJoin will be told: no need to read the small table in an empty partition, because it would not join any results.

Baidu real product line trial case

We will execute adaptive optimization application within the Baidu on Spark SQL ad hoc query service BaiduBig SQL-based, made further ground verification, by choosing the day the real user queries a single day, and re-run the analysis in accordance with the playback of the original order of execution, We get the following conclusions:

- For simple queries in seconds, the adaptive version of the performance is not obvious, mainly because they are time-consuming bottlenecks and mainly concentrated in the IO above, which is not adaptive optimization point of execution.
- According to the complexity of the query dimensions considered test results showed that: The more times an iterative query, the better multi-table join under the more complex adaptive case scenario execution results. We simply according to the number of group by, sort, join, subqueries operable to classify a query, as query keyword is greater than 3, a significant performance improvement from optimized ratio of from 50% to 200%, and the main optimization points dynamic adjustment of the number of concurrent shuffle and join optimization.
- From a business point of view to use to analyze and optimize the previously described SortMergeJoin turn BroadcastHashJoin hit in the Big SQL scene in a variety of typical business template SQL, consider the demand is calculated as follows: the user desires to gain from the billing information in two different dimensions of sense a list of user interest in the overall billing in two dimensions. Meta-information tables contain the size of the original income information corresponding to the user's only a hundred T level, the list of users, the size of less than 10M. Two accounting information table field basically the same, so we will be two tables with a list of users do inner join union for further analysis, SQL expressed as follows

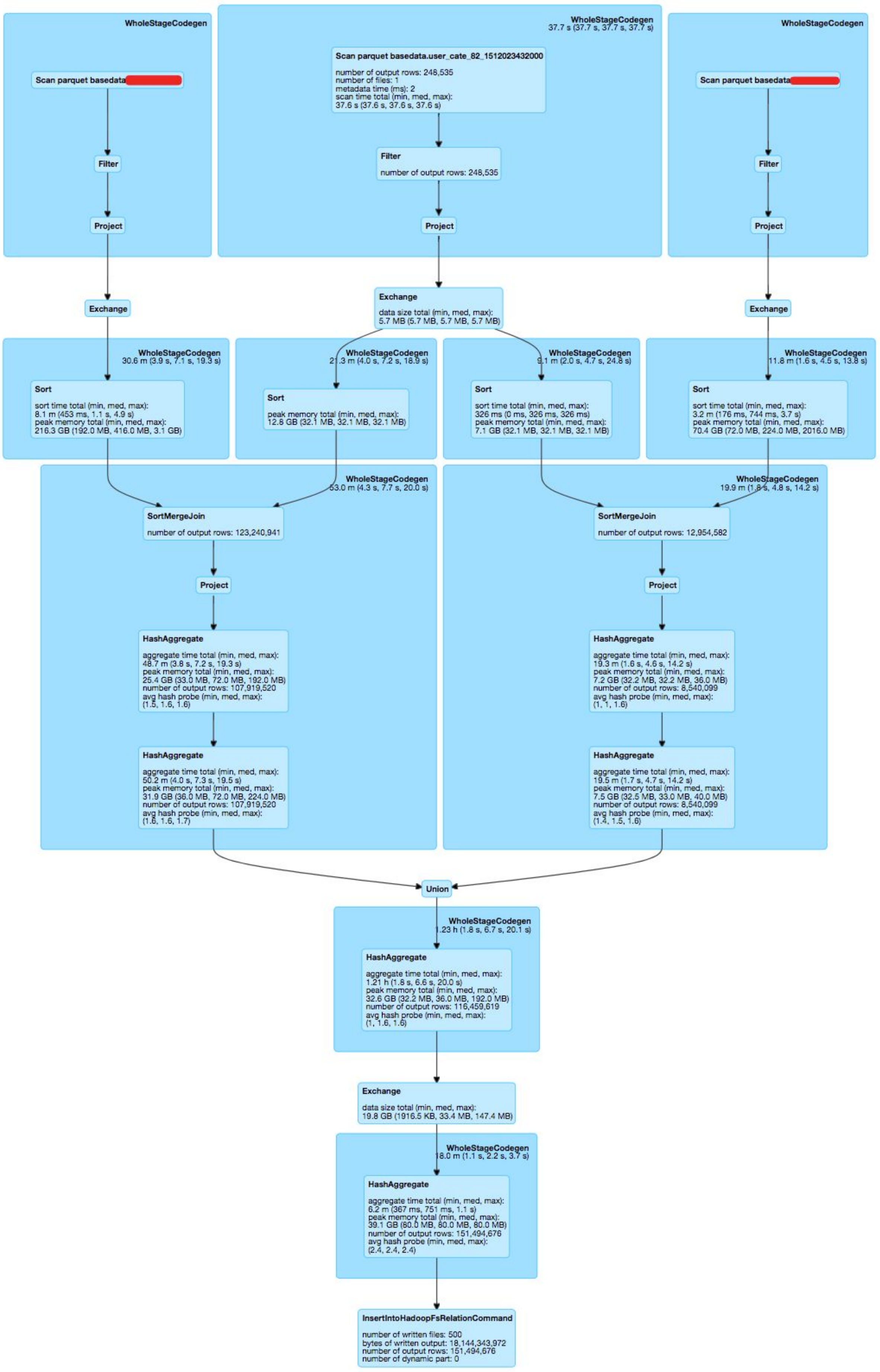
```
select t.c1, t.id, t.c2, t.c3, t.c4, sum (t.num1), sum (t.num2), sum (t.num3) from (
    select c1, t1.id as id, c2, c3, c4, sum (num1s) as num1, sum (num2)
     as num2, sum (num3) as num3
    from basedata.shitu_a t1
    INNER JOIN basedata.user_82_1512023432000 t2 ON (t1.id =
    t2.id)
    where (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
```

Spark corresponding to the original execution plan is as follows:


WholeStageCodegen
Scan parquet basedata[redacted]
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[redacted]
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

Such a scene is directed to the user, all the hits can join optimization logic adaptively performed multiple times during the execution SortMergeJoin into BroadcastHashJoin, reduce memory consumption and intermediate Sort rounds, was nearly 200% performance increase.

Conjunction with the above three points, the adaptive optimization is performed next landing Baidu work within the routine will be further concentrated batch job on large amounts of data and complex queries, and to consider the dynamic switch control complexity associated with the user query. For complex queries running on large clusters of thousands, adaptive execution can dynamically adjust the degree of parallelism calculation process that can help resource utilization increased dramatically cluster. In addition, the adaptive execution can obtain a more complete stage between rounds of statistical information, the next step we will also consider the corresponding data interfaces open to Baidu and Strategy
Spark topsides user for further customization Strategy write a policy for a particular job.

to sum up

With Spark SQL widespread use and the growing size of the business, ease of use and performance challenges encountered in the large-scale data sets will become increasingly apparent. This article discusses three typical problems, including adjusting the number of shuffle partition, choose the best execution plan and data skew. These issues within the existing framework is not easy to solve, and the implementation of adaptation can be a good deal with these issues. We introduce the basic architecture and specific methods to solve these problems adaptive execution. Finally, we verified on the TPC-DS 100TB datasets advantage of adaptive execution, compared to the original Spark SQL, 103 of SQL queries, 90% of queries have been significant performance gains, the biggest upgrade to 3.8 times, and the failure of the original five in the adaptive query execution also successfully completed. We Baidu's Big SQL platform also made further validation for complex queries can be true up to 2 times the performance. In short, the adaptive execution to solve the Spark

SQL encountered many challenges on a large scale data, and largely improved ease of use and performance Spark SQL, improved multi-tenant under multiple concurrent job situation cluster resource utilization super cluster. In the future, we consider under the framework of the implementation of adaptation, can provide a more optimized run-time policy, and our work will contribute back to the community, but also hope that more friends can participate, will be further improved.

This switched: "Spark SQL 100TB on the implementation of adaptive practices"