POSTED ON DECEMBER 15, 2010 TO CORE DATA

Join Optimization in Apache Hive

Small Table	Big Table	Join Condition	Average Previous Map Join Execution time	Average New Optimized Map Join Execution time	Performance Improvement
75 K rows; 383K file size	130 M rows; 3.5G file size;	1 join key, 2 join value	1032 sec	79 sec	+ 1206%
500 K rows; 2.6M file size	130 M rows; 3.5G file size	1 join key, 2 join value	3991 sec	144 sec	+2671 %
75 K rows; 383K file size	16.7 B rows; 459 G file size	1 join key, 2 join value	4801 sec	325 sec	+ 1377 %

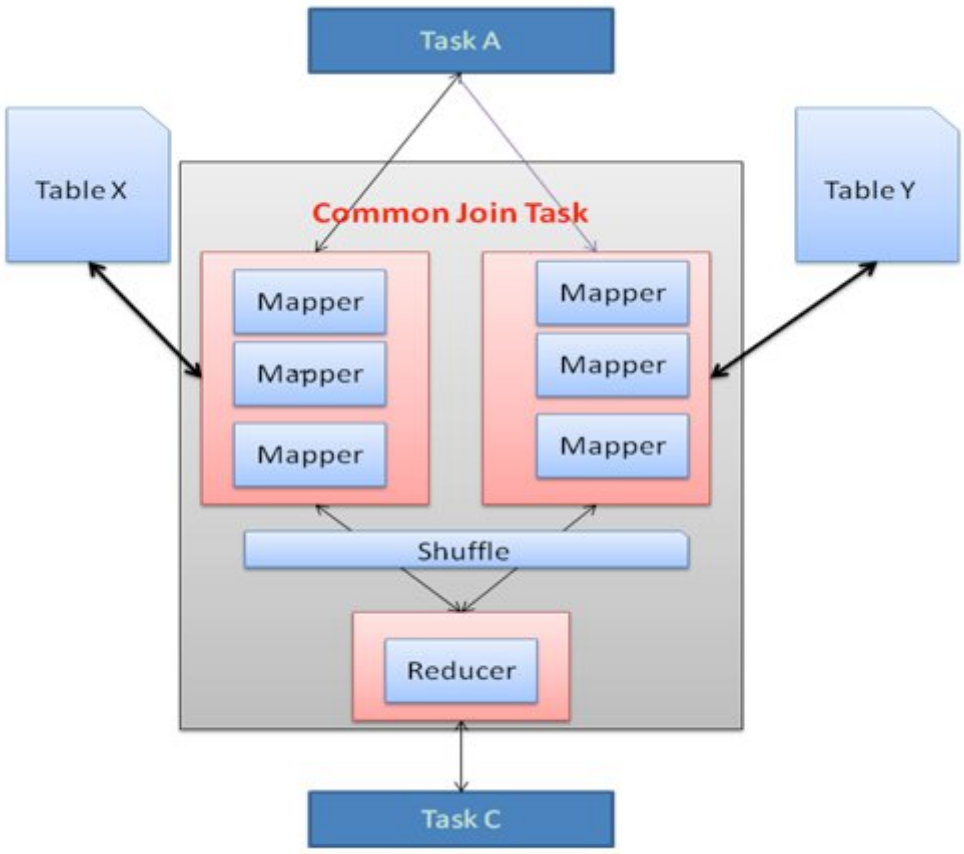
By Liyin Tang

With more than 500 million users sharing a billion pieces of content daily, Facebook stores a vast amount of data, and needs a solid infrastructure to store and retrieve that data. This is why we use Apache Hive and Apache Hadoop so widely at Facebook. Hive is a data warehouse infrastructure built on top of Hadoop that can compile SQL queries as MapReduce jobs and run the jobs in the cluster.

As performant as Hive and Hadoop are, there is always room for improvement. I was so excited that my internship project was to optimize performance of join, a very common SQL operation, in Hive.

How Joins Work Today

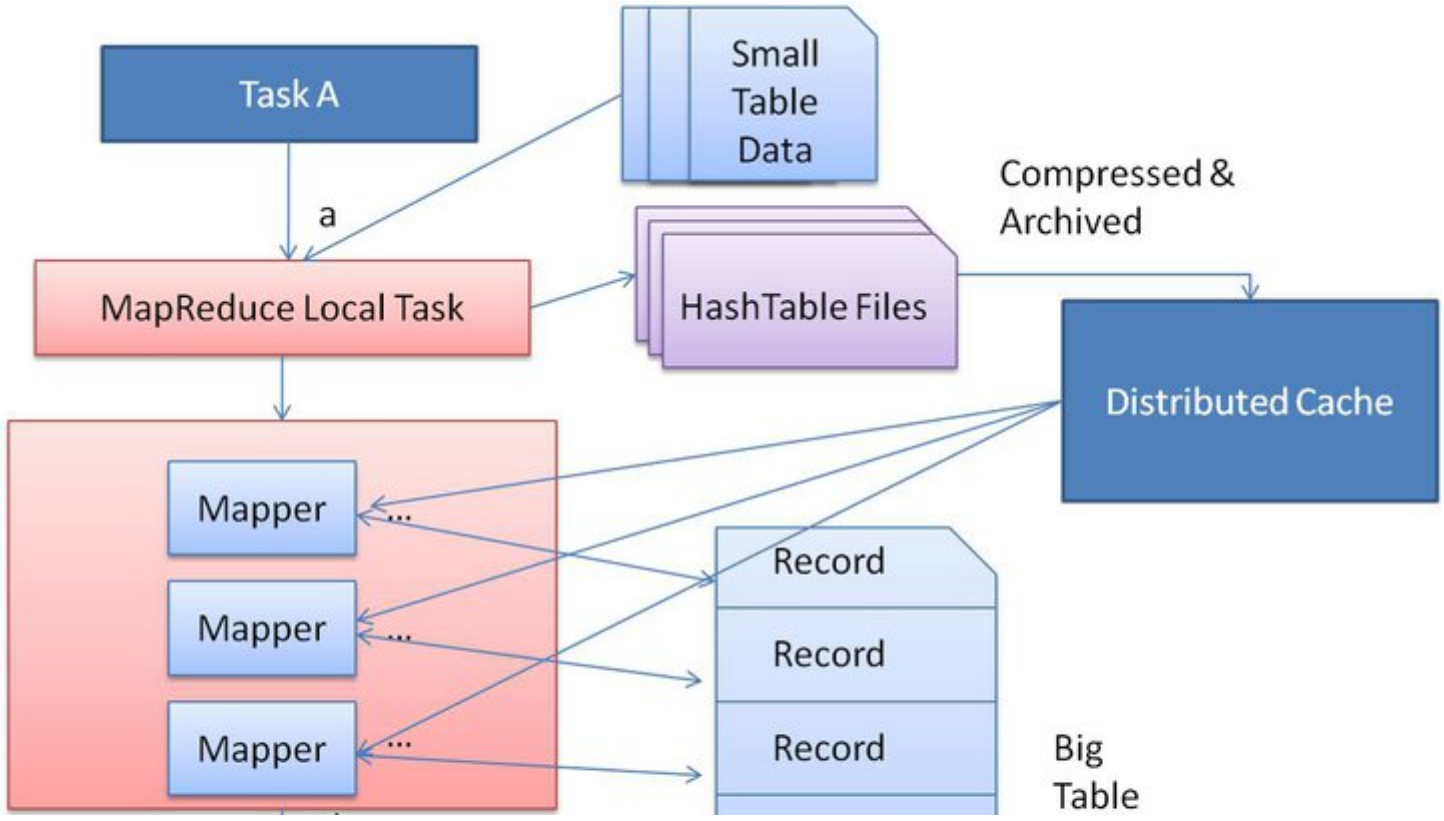
First, let's discuss how join works in Hive. A common join operation will be compiled to a MapReduce task, as shown in figure 1. A common join task involves a map stage and a reduce stage. A mapper reads from join tables and emits the join key and join value pair into an intermediate file. Hadoop sorts and merges these pairs in what's called the shuffle stage. The reducer takes the sorted results as input and does the actual join work. The shuffle stage is really expensive since it needs to sort and merge. Saving the shuffle and reduce stages improves the task performance.



The motivation of map join is to save the shuffle and reduce stages and do the join work only in the map stage. By doing so, when one of the join tables is small enough to fit into the memory, all the mappers can hold the data in memory and do the join work there. So all the join operations can be finished in the map stage. However there are some scaling problems with this type of map join. When thousands of mappers read the small join table from the Hadoop Distributed File System (HDFS) into memory at the same time, the join table easily becomes the performance bottleneck, causing the mappers to time out during the read operations.

Using the Distributed Cache

Hive-1641 solves this scaling problem. The basic idea of optimization is to create a new MapReduce local task just before the original join MapReduce task. This new task reads the small table data from HDFS to an in-memory hash table. After reading, it serializes the in-memory hash table into a hashtable file. In the next stage, when the MapReduce task is launching, it uploads this hashtable file to the Hadoop distributed cache, which populates these files to each mapper's local disk. So all the mappers can load this persistent hashtable file back into memory and do the join work as before. The execution flow of the optimized map join is shown in figure 2. After optimization, the small table needs to be read just once. Also if multiple mappers are running on the same machine, the distributed cache only needs to push one copy of the hashtable file to this machine.

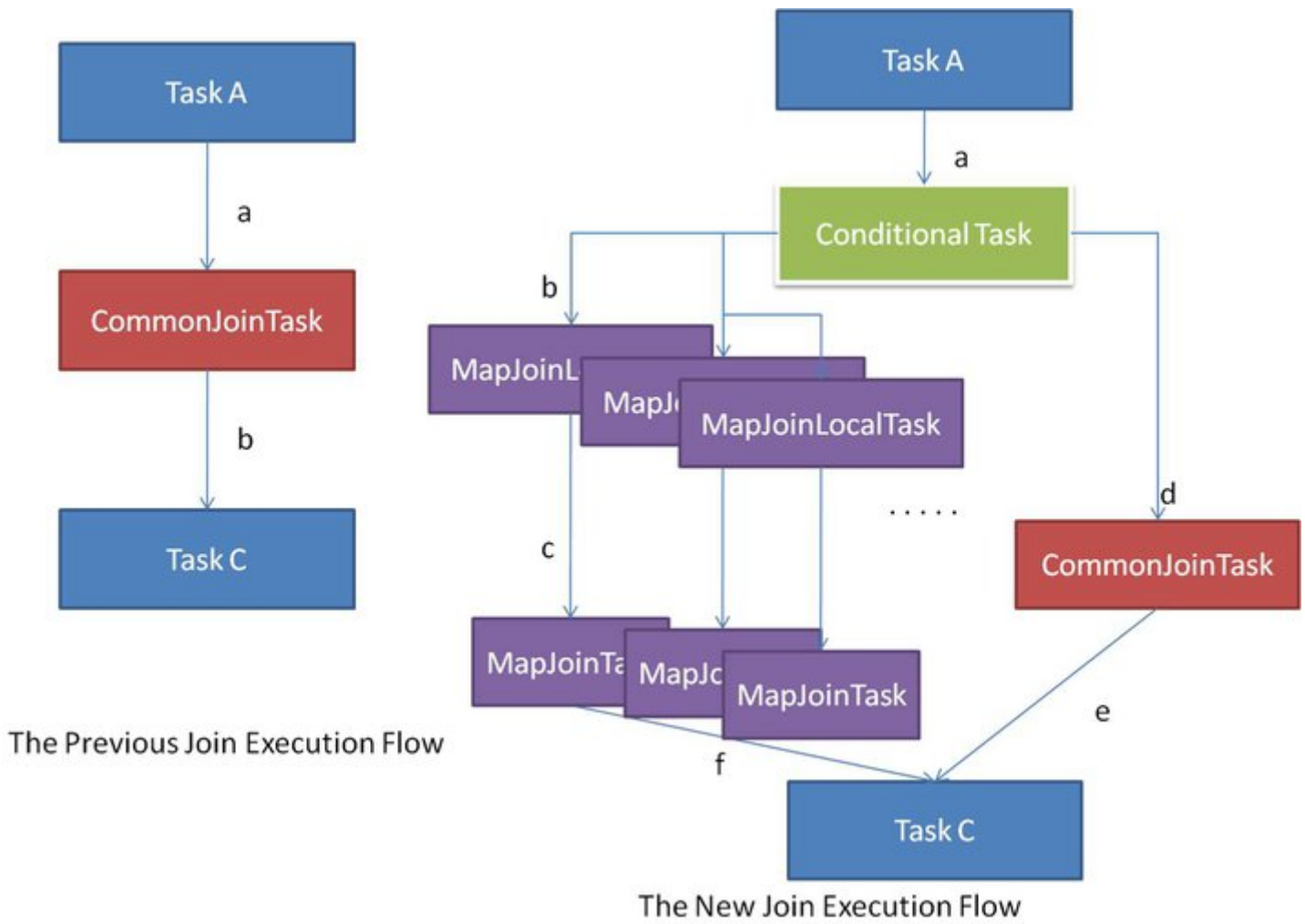


Since map join is faster than the common join, it's better to run the map join whenever possible. Previously, Hive users needed to give a hint in the query to specify the small table. For example, `select /*+mapjoin(a)*/ * from src1 x join src2 y on x.key=y.key;`.

This isn't a good user experience because sometimes the user may give the wrong hint or may not give any hint at all. It's much better to convert the common join into a map join without user hints.

Converting Joins to Map Joins Based on Size

Hive-1642 solves this problem by converting the common join into a map join automatically. For the map join, the query processor should know which input table is the big table. The other input tables are recognized as the small tables during the execution stage, and these tables need to be held in the memory. However, in general, the query processor has no idea of input file size during compilation time because some of the tables may be intermediate tables generated from sub-queries. So the query processor can only figure out the input file size during the execution time.



As shown in figure 3, the left side flow shows the previous common join execution flow, which is very straightforward. On the other side, the right side flow is the new common join execution flow. During compilation time, the query processor generates a conditional task containing a list of tasks; one of these tasks gets resolved to run during execution time. First, the original common join task should be put into the task list. Then the query processor generates a series of map join tasks by assuming each of the input tables may be the big table. For example, `select * from src1 x join src2y on x.key=y.key`. Because both tables src2 and src1 can be the big table, the processor generates two map join tasks, with one assuming src1 is the big table and the other assuming src2 is the big table.

During the execution stage, the conditional task knows the exact file size of each input table, even if the table is an intermediate one. If all the tables are too large to be converted into map join, then just run the common join task as previously. If one of the tables is large and others are small enough to run map join, then the conditional task will pick the corresponding map join local task to run. By this mechanism, it can convert the common join into a map join automatically and dynamically.

Currently, if the total size of small tables is larger than 25MB, then the conditional task will choose the

Now let's see how much of a performance improvement we can get after the map join optimization.

Small Table	Big Table	Join Condition	Average Previous Map Join Execution time	Average New Optimized Map Join Execution time	Performance Improvement
75 K rows; 383K file size	130 M rows; 3.5G file size;	1 join key, 2 join value	1032 sec	79 sec	+ 1206%
500 K rows; 2.6M file size	130 M rows; 3.5G file size	1 join key, 2 join value	3991 sec	144 sec	+2671 %
75 K rows; 383K file size	16.7 B rows; 459 G file size	1 join key, 2 join value	4801 sec	325 sec	+ 1377 %

As shown in table 1, the optimized map join is 12 to 26 times faster than the previous one. Most of map join performance improvement comes from removing the JDBM component.

Also, let's see how much performance improvement we can get if a common join can be converted into map join. All the join operations in the benchmarks can be converted into map join.

Small Table	Big Table	Join Condition	Average Previous Common Join Execution time	Average New Optimized Common Join Execution time	Performance Improvement
75 K rows; 383K file size	130 M rows; 3.5G file size;	1 join key, 2 join value	169 sec	79 sec	+ 114%
500 K rows; 2.6M file size	130 M rows; 3.5G file size	1 join key, 2 join value	246 sec	144 sec	+71 %
75 K rows; 383K file size	16.7 B rows; 459 G file size	1 join key, 2 join value	511 sec	325 sec	+ 57 %
500 K rows; 2.6M file size	16.7 B rows; 459 G file size	1 join key, 2 join value	502 sec	305 sec	+64 %
1M rows; 10M file size	16.7 B rows; 459 G file size	1 join key, 3 join value	653 sec	248 sec	+163 %
1M rows; 10M file size	16.7 B rows; 459 G file size	2 join key, 2 join value	1117sec	536 sec	+108%

From the results shown in table 2, if the new common join can be converted into map join, it will get 57% – 163% performance improvements.

In order to measure the benefits from this project – we would like to track all instances where join operations are converted to map joins (and similarly cases where the map-join optimization fails because the mappers run out of memory). Hive-1792 allows us to capture such tracking information in a generic way. For our Hive deployment, I developed a Hive execution hook to read the tracking information provided by this issue and record it in an internal database. After this work, we can know exactly how many common joins have been converted into map joins everyday and how much CPU time is saved in the cluster.

Looking Ahead

There are a few optimizations remaining to be done. One tricky aspect is setting up the number of replicas for the compressed hashtable files in HDFS. Currently, the number of replicas is initially set to 3 and (asynchronously) increased to 10. But 10 replicas may be insufficient when thousands of nodes need to read this file (and load it into the distributed cache). In addition, the lag time to increase the replication from 3 to 10 is non-deterministic and causes large variations in mapper performance. So we need to figure out the replication factor dynamically based on the number of mappers as well as achieve the higher replication factor in a deterministic and synchronous manner.

size can contain more than 100 million rows. Because of this conservative limit, we may lose many opportunities to convert joins into map joins. A smarter strategy may be to sample the input file to better estimate the number of rows and decide to try the map join strategy based on this estimate.

Finally, the local task can hold 2 million unique key/value pairs in memory, but it consumes 1.47GB of memory. By optimizing the code of hashtable implementations to be more memory efficient, the local task can hold more data in memory, allowing more join queries to be converted into map joins. There are many ideas to explore here, like using hash tables optimized for holding primitive values, which use fewer objects.

Liyin is an intern on our data infrastructure engineering team.

(Originally posted on the Hive wiki.)

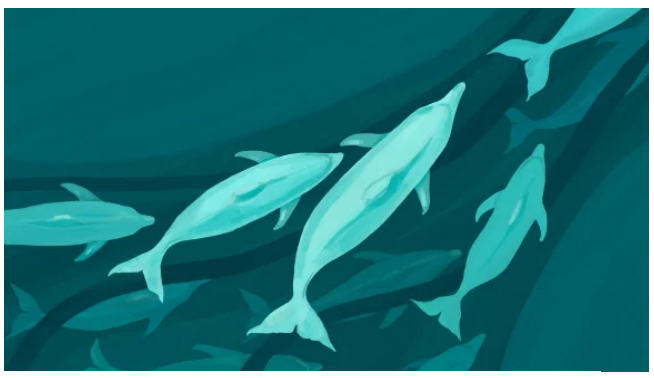
TAGS: INFRA OPEN SOURCE OPTIMIZATION PERFORMANCE

Prev
Visualizing Friendships

Next
A New Year of Facebook Fellowships

Read More in Core Data

View All

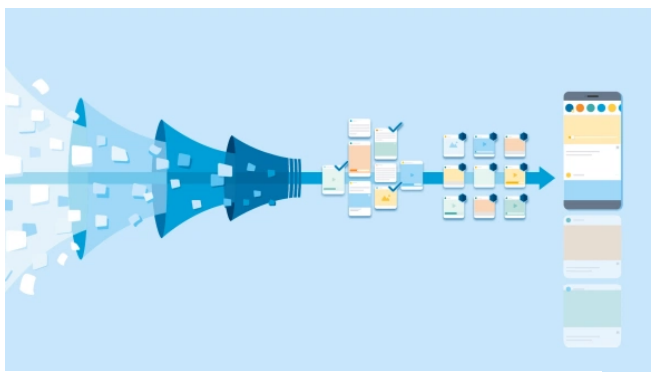


JUL 22, 2021

Migrating Facebook to MySQL 8.0

JUL 9, 2021

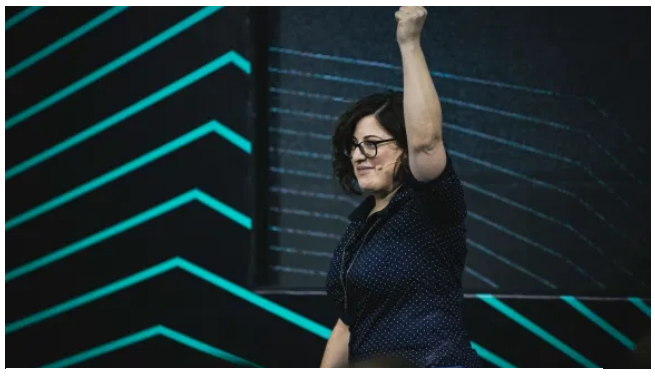
Ribbon filter: Practically smaller than Bloom and Xor



JAN 26, 2021

How machine learning powers Facebook's News Feed ranking algorithm

APR 21, 2020



DEC 20, 2019

OCT 15, 2019

Related Positions

Data Engineering Manager, Facebook Financial (F2)
TEL AVIV, ISRAEL

Product Quality, SDET
MENLO PARK, US

Product Quality, SDET
SEATTLE, US

Data Scientist - Product Analytics
REMOTE, CANADA

Data Scientist Manager, Fraud
TEL AVIV, ISRAEL

See All Jobs

Available Positions

Data Engineering Manager, Facebook Financial (F2)
TEL AVIV, ISRAEL
Product Quality, SDET
MENLO PARK, US
Product Quality, SDET
SEATTLE, US
Data Scientist - Product Analytics
REMOTE, CANADA
Data Scientist Manager, Fraud
TEL AVIV, ISRAEL

See All Jobs

Stay Connected

Facebook Engineering
Like

@fbOpenSource
Follow

facebook research

Facebook Research
Like

Facebook Developers
Like

RSS
Subscribe

Open Source

Facebook believes in building community through open source technology. Explore our latest projects in Artificial Intelligence, Data Infrastructure, Development Tools, Front End, Languages, Platforms, Security, Virtual Reality, and more.

ANDROID
iOS
WEB
BACKEND
HARDWARE

Learn More