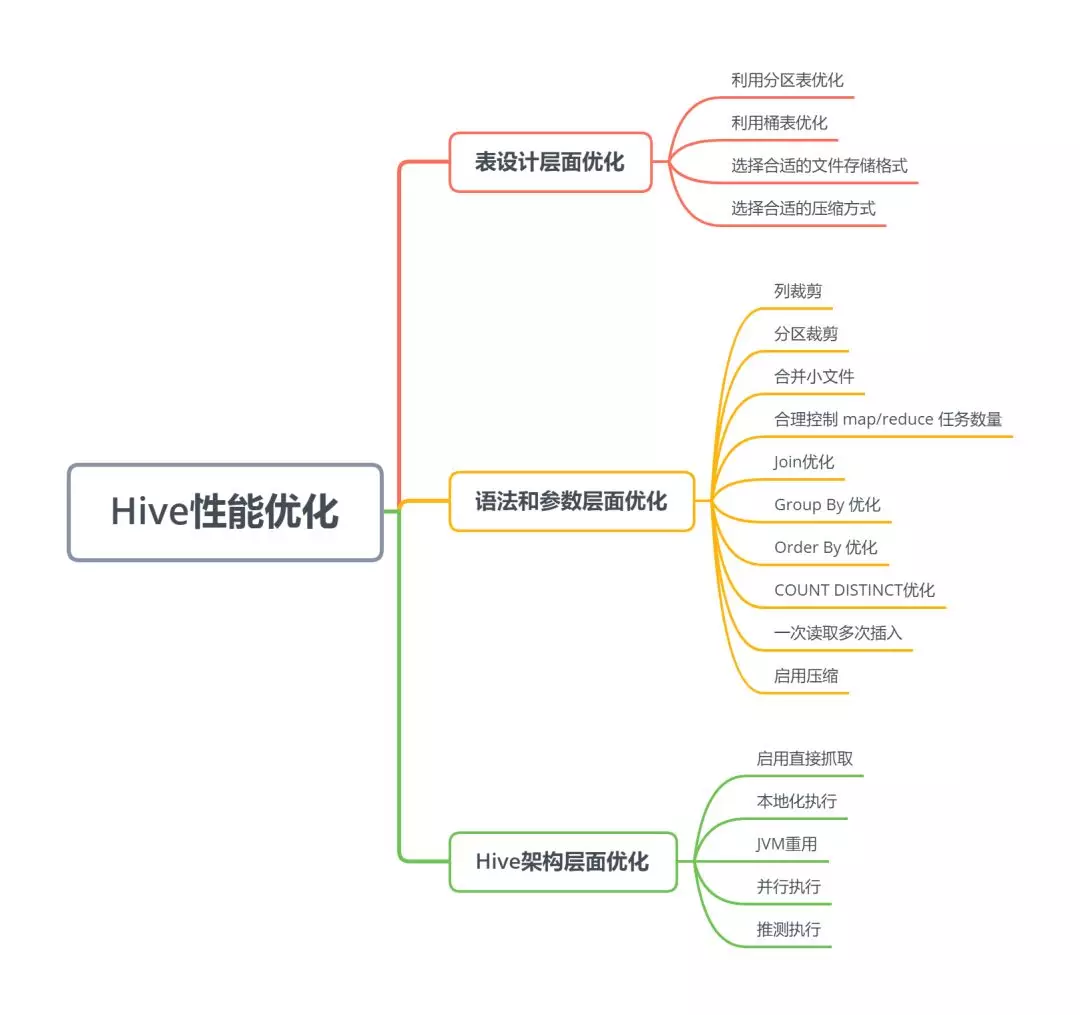
**Hive性能优化篇**

从三个方面对 Hive 常用的一些性能优化进行了总结。

**表设计层面优化**

利用分区表优化

**分区表** 是在某一个或者几个维度上对数据进行分类存储，一个分区对应一个目录。如果筛选条件里有分区字段，那么 Hive 只需要遍历对应分区目录下的文件即可，不需要遍历全局数据，使得处理的数据量大大减少，从而提高查询效率。

当一个 Hive 表的查询大多数情况下，会根据某一个字段进行筛选时，那么非常适合创建为分区表。

利用桶表优化

指定桶的个数后，存储数据时，根据某一个字段进行哈希后，确定存储在哪个桶里，这样做的目的和分区表类似，也是使得筛选时不用全局遍历所有的数据，只需要遍历所在桶就可以了。

选择合适的文件存储格式

Apache Hive 支持 Apache Hadoop 中使用的几种熟悉的文件格式。

**TextFile**默认格式，如果建表时不指定默认为此格式。

存储方式：行存储。

每一行都是一条记录，每行都以换行符\n结尾。数据不做压缩时，磁盘会开销比较大，数据解析开销也比较大。

可结合 **Gzip**、**Bzip2** 等压缩方式一起使用（系统会自动检查，查询时会自动解压），但对于某些压缩算法 hive 不会对数据进行切分，从而无法对数据进行并行操作。

**SequenceFile(**行存储)。

一种Hadoop API 提供的二进制文件，使用方便、可分割、可压缩的特点。

支持三种压缩选择：NONE、RECORD、BLOCK。RECORD压缩率低，一般建议使用BLOCK压缩。

**RCFile**

存储方式：数据按行分块，每块按照列存储 。

* 首先，将数据按行分块，保证同一个record在一个块上，避免读一个记录需要读取多个block。
* 其次，块数据列式存储，有利于数据压缩和快速的列存取。

**ORC(Hive特别优化过)**

存储方式：数据按行分块，每块按照列存储

Hive 提供的新格式，属于 RCFile 的升级版，性能有大幅度提升，而且数据可以压缩存储，压缩快，快速列存取。

**Parquet（Spark对这种格式进行过优化）**

存储方式：列式存储

Parquet 对于大型查询的类型是高效的。对于扫描特定表格中的特定列查询，Parquet特别有用。Parquet一般使用 Snappy、Gzip 压缩。默认 Snappy。

Parquet 支持 Impala 查询引擎（SQL）。

表的文件存储格式尽量采用 **Parquet** 或 **ORC**，不仅降低存储量，还优化了查询，压缩，表关联等性能；

选择合适的压缩方式

Hive 语句最终是转化为 MapReduce 程序来执行的，而 MapReduce 的性能瓶颈在与 **网络IO** 和 **磁盘IO**，要解决性能瓶颈，最主要的是 **减少数据量**，对数据进行压缩是个好方式。压缩虽然是减少了数据量，但是压缩过程要消耗CPU，但是在Hadoop中，往往性能瓶颈不在于CPU，CPU压力并不大，所以压缩充分利用了比较空闲的CPU。

**如何选择压缩方式**

1. 压缩比率
2. 压缩解压速度
3. 是否支持split

支持分割的文件可以并行的有多个 mapper 程序处理大数据文件，大多数文件不支持可分割是因为这些文件只能从头开始读。

**二、语法和参数层面优化（SQL语句相关）**

列裁剪

Hive 在读数据的时候，可以只读取查询中所需要用到的列，而忽略其他的列。这样做可以节省读取开销，中间表存储开销和数据整合开销。Select 列1、列2

set hive.optimize.cp = true; *-- 列裁剪，取数只取查询中需要用到的列，默认为真*

分区裁剪

在查询的过程中只选择需要的分区，可以减少读入的分区数目，减少读入的数据量。

Where 分区列作为条件

set hive.optimize.pruner=true; // 默认为true

合并小文件

**Map 输入合并**

在执行 MapReduce 程序的时候，一般情况是一个文件一个 mapper 来处理。但是如果数据源是大量的小文件，这样岂不是会启动大量的 mapper 任务，这样会浪费大量资源。可以将输入的小文件进行合并，从而减少mapper任务数量。

set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat; *-- Map端输入、合并文件之后按照block的大小分割（默认）*  
set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat; *-- Map端输入，不合并*

**Map/Reduce输出合并**

大量的小文件会给 HDFS 带来压力，影响处理效率。可以通过合并 Map 和 Reduce 的结果文件来消除影响。

set hive.merge.mapfiles=true; *-- 是否合并Map输出文件, 默认值为真*  
set hive.merge.mapredfiles=true; *-- 是否合并Reduce 端输出文件,默认值为假*  
set hive.merge.size.per.task=25610001000; *-- 合并文件的大小,默认值为 256000000*

合理控制 map/reduce 任务数量

**合理控制 mapper 数量**

减少 mapper 数可以通过合并小文件来实现 增加 mapper 数可以通过控制上一个 reduce

默认的 mapper 个数计算方式

输入文件总大小：total\_size  
hdfs 设置的数据块大小：dfs\_block\_size  
default\_mapper\_num = total\_size/dfs\_block\_size

MapReduce 中提供了如下参数来控制 map 任务个数：

1280M 128M 10

set mapred.map.tasks=10;

从字面上看，貌似是可以直接设置 mapper 个数的样子，但是很遗憾不行，这个参数设置只有在大于default\_mapper\_num的时候，才会生效。

那如果我们需要减少 mapper 数量，但是文件大小是固定的，那该怎么办呢?可以通过mapred.min.split.size设置每个任务处理的文件的大小，这个大小只有在大于dfs\_block\_size的时候才会生效

split\_size=max(mapred.min.split.size, dfs\_block\_size)  
split\_num=total\_size/split\_size  
compute\_map\_num = min(split\_num, max(default\_mapper\_num, mapred.map.tasks))

这样就可以减少mapper数量了。

总结一下控制 mapper 个数的方法：

* 如果想增加 mapper 个数，可以设置mapred.map.tasks为一个较大的值
* 如果想减少 mapper 个数，可以设置maperd.min.split.size为一个较大的值
* 如果输入是大量小文件，想减少 mapper 个数，可以通过设置hive.input.format合并小文件

如果想要调整 mapper 个数，在调整之前，需要确定处理的文件大概大小以及文件的存在形式（是大量小文件，还是单个大文件），然后再设置合适的参数。

**合理控制reducer数量**

如果 reducer 数量过多，一个 reducer 会产生一个结果文件，这样就会生成很多小文件，那么如果这些结果文件会作为下一个 job 的输入，则会出现小文件需要进行合并的问题，而且启动和初始化 reducer 需要耗费和资源。

如果 reducer 数量过少，这样一个 reducer 就需要处理大量的数据，并且还有可能会出现数据倾斜的问题，使得整个查询耗时长。默认情况下，hive 分配的 reducer 个数由下列参数决定:

* 参数1：hive.exec.reducers.bytes.per.reducer(默认1G)
* 参数2：hive.exec.reducers.max(默认为1009)

reducer的计算公式为:

N = min(参数2， 总输入数据量/参数1)

可以通过改变上述两个参数的值来控制reducer的数量。也可以通过

set mapred.map.tasks=10;

直接控制reducer个数，如果设置了该参数，上面两个参数就会忽略。

Join优化

**优先过滤数据**

尽量减少每个阶段的数据量，对于分区表能用上分区字段的尽量使用，同时只选择后面需要使用到的列，最大限度的减少参与 join 的数据量。

**小表 join 大表原则**

小表 join 大表的时应遵守小表 join 大表原则，原因是 join 操作的 reduce 阶段，位于 join 左边的表内容会被加载进内存，**将条目少的表放在左边**，可以有效减少发生内存溢出的几率(OOM : out of memory)。join 中执行顺序是从左到右生成 Job，应该保证连续查询中的表的大小从左到右是依次增加的。

**使用相同的连接键**

在 hive 中，当对 3 个或更多张表进行 join 时，如果 **on 条件使用相同字段**，那么它们会合并为一个 MapReduce Job，利用这种特性，可以将相同的 join on 的放入一个 job 来节省执行时间。

**启用 mapjoin**

mapjoin 是将 join 双方比较小的表直接分发到各个map进程的内存中，在 map 进程中进行 join 操作，这样就不用进行 reduce 步骤，从而提高了速度。只有 join 操作才能启用 mapjoin。

set hive.auto.convert.join = true; *-- 是否根据输入小表的大小，自动将reduce端的common join 转化为map join，将小表刷入内存中。*  
set hive.mapjoin.smalltable.filesize = 2500000; *-- 刷入内存表的大小(字节)*  
set hive.mapjoin.maxsize=1000000; *-- Map Join所处理的最大的行数。超过此行数，Map Join进程会异常退出*

*Hint方式设置mapjoin*

select /\*+MAPJOIN(smallTableTwo)\*/ idOne, idTwo, value FROM

( select /\*+MAPJOIN(smallTableOne)\*/ idOne, idTwo, value FROM

bigTable JOIN smallTableOne on (bigTable.idOne = smallTableOne.idOne)

) firstjoin

JOIN

smallTableTwo ON (firstjoin.idTwo = smallTableTwo.idTwo)

**Hint方式影响SQL解析器优化器生成的执行计划，会影响到SQL优化器。**

**CBO：基于成本的SQL/HQL优化器。**

**尽量原子操作**

尽量避免一个SQL包含复杂的逻辑，可以使用中间表来完成复杂的逻辑。

**桶表 mapjoin**

当两个分桶表 join 时，如果 join on的是分桶字段，小表的分桶数是大表的倍数时，可以启用 mapjoin 来提高效率。

set hive.optimize.bucketmapjoin = true; *-- 启用桶表 map join*

Group By 优化

默认情况下，Map阶段同一个Key的数据会分发到一个Reduce上，当一个Key的数据过大时会产生 **数据倾斜**。进行group by操作时可以从以下两个方面进行优化：

**1. Map端部分聚合**

事实上并不是所有的聚合操作都需要在 Reduce 部分进行，很多聚合操作都可以先在 Map 端进行部分聚合，然后在 Reduce 端的得出最终结果。

set hive.map.aggr=true; *-- 开启Map端聚合参数设置*  
set hive.grouby.mapaggr.checkinterval=100000; *-- 在Map端进行聚合操作的条目数目*

**2. 有数据倾斜时进行负载均衡**

set hive.groupby.skewindata = true; *-- 有数据倾斜的时候进行负载均衡（默认是false）*

当选项设定为 true 时，生成的查询计划有两个MapReduce 任务。在第一个 MapReduce 任务中，map 的输出结果会随机分布到 reduce 中，每个 reduce 做部分聚合操作，并输出结果，这样处理的结果是相同的group by key有可能分发到不同的 reduce 中，从而达到负载均衡的目的；第二个 MapReduce 任务再根据预处理的数据结果按照group by key分布到各个 reduce 中，最后完成最终的聚合操作。

Order By 优化

order by只能是在一个reduce进程中进行，所以如果对一个大数据集进行order by，会导致一个reduce进程中处理的数据相当大，造成查询执行缓慢。

* 在最终结果上进行order by，不要在中间的大数据集上进行排序。如果最终结果较少，可以在一个reduce上进行排序时，那么就在最后的结果集上进行order by。
* 如果是去排序后的前N条数据，可以使用distribute by和sort by在各个reduce上进行排序后前N条，然后再对各个reduce的结果集合合并后在一个reduce中全局排序，再取前N条，因为参与全局排序的order by的数据量最多是reduce个数 \* N，所以执行效率很高。

COUNT DISTINCT优化（hive3.0已经内部优化过）

*-- 优化前（只有一个reduce，先去重再count负担比较大）：*  
select count(distinct id) from tablename;  
*-- 优化后（启动两个job，一个job负责子查询(可以有多个reduce)，另一个job负责count(1))：*  
select count(1) from (select distinct id from tablename) tmp;

在进行去重操作的时候尽量的使用group by来操作。

一次读取多次插入

有些场景是从一张表读取数据后，要多次利用，这时可以使用multi insert语法：

from sale\_detail  
 insert overwrite table sale\_detail\_multi partition (sale\_date='2010', region='china' )  
 select shop\_name, customer\_id, total\_price where .....  
 insert overwrite table sale\_detail\_multi partition (sale\_date='2011', region='china' )  
 select shop\_name, customer\_id, total\_price where .....;

说明：

* 一般情况下，单个SQL中最多可以写128路输出，超过128路，则报语法错误。
* 在一个multi insert中：
  + 对于分区表，同一个目标分区不允许出现多次。
  + 对于未分区表，该表不能出现多次。
* 对于同一张分区表的不同分区，不能同时有insert overwrite和insert into操作，否则报错返回。

启用压缩

**map 输出压缩**

set mapreduce.map.output.compress=true;  
set mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.SnappyCodec;

**中间数据压缩**

中间数据压缩就是对 hive 查询的多个 job 之间的数据进行压缩。最好是选择一个节省CPU耗时的压缩方式。可以采用snappy压缩算法，该算法的压缩和解压效率都非常高。

set hive.exec.compress.intermediate=true;  
set hive.intermediate.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;  
set hive.intermediate.compression.type=BLOCK;

**结果数据压缩**

最终的结果数据（Reducer输出数据）也是可以进行压缩的，可以选择一个压缩效果比较好的，可以减少数据的大小和数据的磁盘读写时间；注：常用的gzip，snappy压缩算法是不支持并行处理的，如果数据源是gzip/snappy压缩文件大文件，这样只会有有个mapper来处理这个文件，会严重影响查询效率。所以如果结果数据需要作为其他查询任务的数据源，可以选择支持splitable的LZO算法，这样既能对结果文件进行压缩，还可以并行的处理，这样就可以大大的提高job执行的速度了。关于如何给Hadoop集群安装LZO压缩库可以查看[**这篇文章**](https://mp.weixin.qq.com/s?__biz=MzU5OTM5NjQ2NA==&mid=2247483676&idx=1&sn=2a14972e97bc6c25647e962c12ce3e77&chksm=feb4d803c9c35115c66017e077fdc4b613515d3b93206d62400213bd5ab79ae46f125333db15&scene=21#wechat_redirect)。

set hive.exec.compress.output=true;  
set mapreduce.output.fileoutputformat.compress=true;  
set mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.GzipCodec;  
set mapreduce.output.fileoutputformat.compress.type=BLOCK;

Hadoop集群支持一下算法：

* org.apache.hadoop.io.compress.DefaultCodec
* org.apache.hadoop.io.compress.GzipCodec
* org.apache.hadoop.io.compress.BZip2Codec
* org.apache.hadoop.io.compress.DeflateCodec
* org.apache.hadoop.io.compress.SnappyCodec
* org.apache.hadoop.io.compress.Lz4Codec
* com.hadoop.compression.lzo.LzoCodec
* com.hadoop.compression.lzo.LzopCodec

**三、Hive架构层面优化**

启用直接抓取

Hive 从 HDFS 中读取数据，有两种方式：启用 MapReduce 读取、直接抓取。

直接抓取数据比 MapReduce 方式读取数据要快的多，但是只有少数操作可以使用直接抓取方式。

可以通过hive.fetch.task.conversion参数来配置在什么情况下采用直接抓取方式：

* **minimal**：只有 select \* 、在分区字段上 where 过滤、有 limit 这三种场景下才启用直接抓取方式。
* **more**：在 select、where 筛选、limit 时，都启用直接抓取方式。

--会话级别设置  
set hive.fetch.task.conversion=more; *-- 启用fetch more模式*

*在hive-site.xml中配置，永久生效*

本地化执行

Hive在集群上查询时，默认是在集群上多台机器上运行，需要多个机器进行协调运行，这种方式很好的解决了大数据量的查询问题。但是在Hive查询处理的数据量比较小的时候，其实没有必要启动分布式模式去执行，因为以分布式方式执行设计到跨网络传输、多节点协调等，并且消耗资源。对于小数据集，可以通过本地模式，在单台机器上处理所有任务，执行时间明显被缩短。

set hive.exec.mode.local.auto=true; *-- 打开hive自动判断是否启动本地模式的开关*  
set hive.exec.mode.local.auto.input.files.max=4; *-- map任务数最大值*  
set hive.exec.mode.local.auto.inputbytes.max=134217728; *-- map输入文件最大大小*

JVM重用

Hive 语句最终会转换为一系列的 MapReduce 任务，每一个MapReduce 任务是由一系列的Map Task 和 Reduce Task 组成的，默认情况下，MapReduce 中一个 Map Task 或者 Reduce Task 就会启动一个 JVM 进程，一个 Task 执行完毕后，JVM进程就会退出。这样如果任务花费时间很短，又要多次启动 JVM 的情况下，JVM的启动、关闭时间会变成一个比较大的消耗，这时，可以通过重用 JVM 来解决。

set mapred.job.reuse.jvm.num.tasks=5;

JVM也是有缺点的，开启JVM重用会一直占用使用到的 task 的插槽，以便进行重用，直到任务完成后才会释放。如果某个不平衡的job中有几个 reduce task 执行的时间要比其他的 reduce task 消耗的时间要多得多的话，那么保留的插槽就会一直空闲却无法被其他的 job 使用，直到所有的 task 都结束了才会释放。

并行执行

有的查询语句，hive会将其转化为一个或多个阶段，包括：MapReduce 阶段、抽样阶段、合并阶段、limit 阶段等。默认情况下，一次只执行一个阶段。但是，如果某些阶段不是互相依赖，是可以并行执行的。多阶段并行是比较耗系统资源的。

set hive.exec.parallel=true; *-- 可以开启并发执行。*  
set hive.exec.parallel.thread.number=16; *-- 同一个sql允许最大并行度，默认为8。*

推测执行

在分布式集群环境下，因为程序Bug（包括Hadoop本身的bug），负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致，有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop采用了推测执行（Speculative Execution）机制，它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

set mapreduce.map.speculative=true;  
set mapreduce.reduce.speculative=true;

建议：

如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的map或者Reduce task的话，那么启动推测执行造成的浪费是非常巨大大。

四、HiveQL执行计划：

<https://www.cnblogs.com/halentest/p/3291076.html>

explain hql : 查看执行计划。

explain EXTENDED 查询