Hive SQL和Spark SQL

Hive作为基于Hadoop的一个数据库管理工具，其底层是基于MapReduce实现的，用户写的SQL语句最终会转化为MapReduce任务提交到hadoop运行。不过由于MapReduce速度比较慢（MapReduce计算过程中大量的中间磁盘落地过程会消耗大量I/O，降低运行效率），因此，近几年陆续出来一些新的SQL查询引擎，比如Spark SQL，Hive On Tez，Hive On Spark等。比如现在常用的Spark SQL，是Spark自己研发出来的针对Hive、JSON、Parquet、JDBC、RDD等各种数据源的，基于Spark计算引擎的查询引擎，本身兼容Hive。

HIve SQL优化基本思想

由于Hive SQL会以MapReduce任务提交到hadoop运行，所以性能瓶颈主要在4个地方：

- Map task：select 等数据读取的操作，从磁盘中将数据读入内存
- Reduce task：group by/order by 等聚合排序操作
- Join task：Join 等表关联操作
- Merge task：无对应代码，是小文件合并任务

数据量大对于Hive SQL（或Spark SQL）来说并不是主要挑战，最怕的是数据倾斜、数据冗余、job或I/O过多、MapReduce分配不合理等，如果能将数据比较合理的分配到各个计算节点上，计算压力并不会很大，但是如果很多task都被分配到一个节点上，这样分布式计算的优势就发挥不出来，这种最直接的表现就是计算卡在99%（或其他某个数字）不动，看任务只有一两个task就结束，实则可能得等到天荒地老，同样，如果有很多小文件，很多IO，也会造成性能下降。

当然也不是说数据量可以一味的增大，更多数据会增加磁盘IO（特别是MR），另外还会增加网络IO，所以能减少无用的计算就优先过滤掉。

HIve SQL优化

SQL写法上的优化

map task优化

- **尽早使用where条件**：提前把不需要计算的数据过滤掉，而不是在进行复杂操作后再集中过滤。
- **使用分区裁剪**：Hive不同分区是按照不同目录存放的，指定分区可以访问特定的目录，减少数据量。
- **使用列裁剪**：尽量不要使用select * from ...，指定特定列会只扫描特定列而不扫描全表，提高执行速度，同时select * 会让优化器无法完成索引覆盖扫描这类优化，会影响优化器对执行计划的选择，也会增加网络带宽消耗，更会带来额外的 I/O，内存和 CPU 消耗。
- **相似任务尽量使用多路输出**：相同的计算只需要计算一次，减少重复计算，同时也能减少reduce task
- **减少case when中的when**：表中的文件都需要走一遍when流程，when越多效率就越低，而且在reduce阶段最好做一遍合并压缩操作，否则可能会产生很多文件。

reduce task优化

- **使用 group by 代替 distinct**：因为distinct会把所有任务都分配到一个reduce task中。
- **使用 sort by + distribute by代替 order by**：order by 和 distinct 类似，在reduce阶段都会把所有的任务集中到一个reduce task中计算，使用 sort by 和 distribute by 后MR会根据情况启动多个reduce来排序，不过记得一定要加distribute by，否则map后的数据会随机分配到reducer中，不能保证全局有序。
- **尽量使用union all代替union**：union去重，有shuffle，union all不去重，无shuffle，shuffle会造成数据在集群中传输，并且伴随着读和写，很影响任务的执行性能。如果要去重，可以最后用group by。

join task过程优化

- **避免使用笛卡尔积**：尽量有关联键，hive本身不支持笛卡尔积，需要先用set hive.mapred.mode=nonstrict设为非strict模式。
- **多表join查询时，小表在前，大表在后**，Hive在解析带join的SQL语句时，会默认将最后一个表作为probe table（大表），将前面的表作为build table（小表）并试图将它们读进内存（是否读入内存可以配置）。如果表顺序写反，probe table在前面，有引发OOM的风险。
- **小表超出内存限制，采用多次join**：build table没有小到可以直接读如内存，但是相比probe table又很小，可以将build table拆成几个表，分别join。
- **小表join大表，尽量使用map join**：将build table和probe table在map端直接完成join过程，没有了reduce，效率高很多。
- **多表join时如果允许尽量使用相同的key**：这样会将多个join合并为一个MR job来处理。
- **join时保证关联键类型相同**：如果不同时也适用cast进行转换，否者会导致另外一个类型的key分配到一个reducer上。
- **join的时候如果关联健某一类值较多先过滤**：比如空值、0等，因为这会导致某一个reducer的计算量变得很大，可以单独处理倾斜key。
- **left semi join 代替join判断in和exists**：hive0.13前不支持在where 中使用in嵌套查询是否exists，使用left semi join代替join。

参数配置上的优化

小表join时尽量开启map join

```
set hive.auto.convert.join=true;  -- 版本0.11.0之后，默认是开启状态的，但时不时会把这个配置关闭，所以最好还是手动配置一下
set hive.mapjoin.smalltable.filesize=25000000;  -- 默认是25Mb开启mapjoin，对于稍微超过这大小的，可以适当调大，但不能太大
```

调整map数

如果输入文件是少量大文件，就减少mapper数；如果输入文件是大量大文件，就增大mapper数；如果是大量的小文件就先合并小文件。

```
set mapred.min.split.size=10000;  -- 最小分片大小
set mapred.max.split.size=10000000;  -- 最大分片大小
set mapred.map.tasks=100;  -- 设置map task任务数
map任务数计算规则：map_num = MIN(split_num, MAX(default_num, mapred.map.tasks)),
```

合并小文件

```
set hive.input.format = org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;  -- 输入阶段合并小文件
set hive.merge.mapredfiles=true;  -- 输出阶段小文件合并
set hive.merge.mapfiles=true;  -- 开启map端合并小文件，默认开启
set hive.merge.mapredfiles=true;  -- 开启reduce端合并小文件
set hive.merge.smallfiles.avgsize=16000000;  -- 平均文件大小，默认16M，满足条件则自动合并，只有在开启merge.mapfiles和merge.mapredfiles两个开关才有效
```

启用压缩

压缩可以减少数据量，进而减少磁盘和网络IO。

```
set hive.exec.compress.intermediate=true;  -- 开启输入压缩
set hive.exec.compress.output=true;  -- 开启输出压缩
set sethive.intermediate.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
 -- 使用Snappy压缩
set mapred.output.compreession.codec=org.apache.hadoop.io.compress.GzipCodec;  -- 使用
Gzip压缩
set hive.intermediate.compression.type=BLOCK;  -- 配置压缩对象 快或者记录
```

分桶设置

```
set hive.enforce.bucketing=true;
set hive.enforce.sorting=true;
```

设置合适的数据存储格式

hive默认的存储格式是TextFile,但是这种文件格式不使用压缩，会占用比较大空间，目前支持的存储格式有SequenceFile、RCFile、Avro、ORC、Parquet，这些存储格式基本都会采用压缩方式，而且是列式存储.

```
如指定用orc存储格式

ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.orc.OrcSerde'STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.orc.OrcInputFormat' OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat'
```

并行化执行

每个查询被hive转化成多个阶段，有些阶段关联性不大，则可以并行化执行，减少执行时间，主要针对uoion 操作

```
set hive.exec.parallel=true;  -- 开启并行模式
set hive.exec.parallel.thread.numbe=8;  -- 设置并行执行的线程数
```

本地化执行

本地模式主要针对数据量小，操作不复杂的SQL。

```
set hive.exec.mode.local.auto;  -- 开启本地执行模模式
需要满足的条件：
job的输入数据大小必须小于参数:hive.exec.mode.local.auto.inputbytes.max(默认128MB)
job的map数必须小于参数:hive.exec.mode.local.auto.tasks.max(默认4)
job的reduce数必须为0或者1
```

使用严格模式

严格模式主要是防范用户的不规范操作造成集群压力过大，甚至是不可用的情况，只对三种情况起左右，分别是查询分区表是不指定分区；两表join时产生笛卡尔积；使用了order by 排序但是没有limit关键字。

```
set hive.mapred.mode=strict;  -- 开启严格模式
```

map端预聚合

```
。预聚合的配置项是
set hive.map.aggr=true;  -- group by时，如果先起一个combiner在map端做部分预聚合，使用这个配置项可以有效减少shuffle数据量，默认值true
set hive.groupby.mapaggr.checkinterval=100000;  -- 也可以设置map端预聚合的行数阈值，超过该值就会分拆job，默认值100000
```

倾斜均衡配置项

```
set hive.groupby.skewindata=false;  -- group by时如果某些key对应的数据量过大，就会发生数据倾斜。Hive自带了一个均衡数据倾斜的配置项，默认值false
```

动态分区配置

```
set hive.exec.dynamic.partition=false;  -- 是否开启动态分区功能，默认false关闭
set hive.exec.dynamic.partition.mode=strict;  -- 动态分区的模式，默认strict，表示必须指定至少一个分区为静态分区，nonstrict模式表示允许所有的分区字段都可以使用动态分区
set hive.exec.max.dynamic.partitions.pernode=100;  -- 在每个执行MR的节点上，最大可以创建多少个动态分区,根据实际的数据来设定，比如hour必须大于等于24，day必须大于365
set hive.exec.max.dynamic.partitions=1000;  -- 在所有执行MR的节点上，最大一共可以创建多少个动态分区
set hive.exec.max.created.files=100000;  -- 整个MR Job中，最大可以创建多少个HDFS文件
set hive.error.on.empty.partition=false;  -- 当有空分区生成时，是否抛出异常
```

JVM重用

```
set mapred.job.reuse.jvm.num.tasks=10;  -- 在MR job中，默认是每执行一个task就启动一个JVM。如果task非常小而碎，那么JVM启动和关闭的耗时就会很长。可以通过调节参数这个参数来重用。例如将这个参数设成5，就代表同一个MR job中顺序执行的10个task可以重复使用一个JVM，减少启动和关闭的开销。但它对不同MR job中的task无效。
```

附：Mysql常用调优技巧：

查询

- **避免在字段开头模糊查询**，会导致数据库引擎放弃索引进行全表扫描（like '%数%' 改成like '数%'）
- **避免使用 in 和 not in**，会导致引擎走全表扫描（id IN (2,3) 改为 id BETWEEN 2 AND 3）
- **避免使用 or**，会导致数据库引擎放弃索引进行全表扫描（id = 1 OR id = 3 改成 select ... id = 1 union select ... id = 3）
- **避免进行 null 值的判断**，会导致数据库引擎放弃索引进行全表扫描（score is NULL 改成 score = 0）
- **避免在 where 条件中等号的左侧进行表达式、函数操作**，会导致数据库引擎放弃索引进行全表扫描（score/10 = 9 改成 score = 9 * 10）
- **当数据量大时，避免使用 where 1=1 的条件**（当WHERE 1=1 在代码拼装时进行判断，没 where 条件就去掉 where，有 where 条件就加 and）
- **查询条件不能用 <> 或者 !=**（如确实业务需要使用到不等于符号，需重新评估索引建立，避免在此字段上建立索引）
- **where 条件仅包含复合索引非前置列**（如：复合(联合)索引包含 key_part1，key_part2，key_part3 三列，但 SQL 语句没有包含索引前置列"key_part1"，按照 MySQL 联合索引的最左匹配原则，不会走联合索引。）
- **隐式类型转换造成不使用索引**（如 select col1 from table_name where col_varchar = 123 由于索引对列类型为 varchar，但给定的值为数值，涉及隐式类型转换，造成不能正确走索引。）
- **order by 条件要与 where 中条件一致，否则 order by 不会利用索引进行排序**（select * FROM t order by age 改成 select * FROM t where age > 0 order by age）
- **避免出现 select ***（会让优化器无法完成索引覆盖扫描这类优化，会影响优化器对执行计划的选择，也会增加网络带宽消耗，更会带来额外的 I/O，内存和 CPU 消耗。）
- **避免出现不确定结果的函数**（如 now()、rand()、sysdate()、current_user() 等不确定结果的函数很容易导致主库与从库相应的数据不一致。）
- **多表关联查询时，小表在前，大表在后**（MySQL扫描顺序是从做到右（Oracle相反），将小表放在前面，先扫小表，扫描快效率较高，在扫描后面的大表，或许只扫描大表的前 100 行就符合返回条件并 return 了。）
- **使用表的别名**（使用表的别名并把别名前缀于每个列名上，减少解析的时间并减少那些有列名歧义引起的语法错误。）
- **用 where 字句替换 HAVING 字句**（HAVING 只在检索出所有记录才对结果集进行过滤，而 where 则是在聚合前刷选记录）
- **调整 Where 字句中的连接顺序**（MySQL 采用从左往右，自上而下的顺序解析 where 子句。根据这个原理，应将过滤数据多的条件往前放，最快速度缩小结果集。）

增删改

- **大批量插入数据**（insert into T values(1,2);insert into T values(1,3);insert into T values(1,4);改成insert into T values(1,2),(1,3),(1,4)）(减少 SQL 语句解析的操作，在特定场景可以减少对 DB 连接次数，SQL 语句较短，可以减少网络传输的 IO。)
- **适当使用 commit**（适当使用 commit 可以释放事务占用的资源而减少消耗，释放的模块有事务占用的undo模块，事务在redo log中记录的数据块，减少锁争用影响性能。）
- **避免重复查询更新的数据**（更新一行记录的时间戳，同时希望查询当前记录中存放的时间戳是什么? update t1 set time = now() where col = 1; select time from t1 where col = 1;改成 update t1 set time = now() where col = 1 and @now:=now(); select @now;）
- **对于复杂的查询，可以使用中间临时表暂存数据**
- **MySQL 会对 GROUP BY 分组的所有值进行排序**（ “GROUP BY col1，col2，....;” 查询的方法如同在查询中指定 “ORDER BY col1，col2，...;” ，如果不要排序可以指定 ORDER BY NULL 禁止排序。）
- **使用join来避免嵌套查询**（因为 MySQL 不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。）

- **使用union all 代替 union**（如果没有 all 这个关键词，MySQL 会给临时表加上 distinct 选项，这会导致对整个临时表的数据做唯一性校验，这样做的消耗相当高。）
- **拆分复杂 SQL 为多个小 SQL，避免大事务**（简单的 SQL 容易使用到 MySQL 的 QUERY CACHE；减少锁表时间特别是使用 MyISAM 存储引擎的表；可以使用多核 CPU）
- **使用 truncate 代替 delete**（使用 delete 语句的操作会被记录到 undo 块中，删除记录也记录 binlog。使用 truncate 替代，不会记录可恢复的信息，数据不能被恢复。）

建表

- **在表中建立索引，优先考虑 where、order by 使用到的字段。**
- **尽量使用数字型字段**
- **查询数据量大的表，会造成查询缓慢**
- **用 varchar/nvarchar 代替 char/nchar。**

完～