2021年3月分布式查询性能优化总结

任务列表：https://shimo.im/docs/38rgdjpQVRq8grPD

partitionTable 同步次数减少优化

问题

在查询时目前每创建一次 reader 就需要向 meta leader 同步一次以获取最新的 partitionTable，实际上没有必要。

解决思路

每次查询同步一次 meta leader。

结果

精确点查询 latency 减少约 15%

测试结果

https://shimo.im/docs/WDc9HQWxwgJjvhk8

原始数据查询的查询重定向优化

问题

查询时如果数据不在协调者节点上，则该查询的请求获取流程多经历一段 RTT，从而增大 latency，且需要占用协调者节点一定的内存和计算资源。

优缺点分析

- 优点：如果数据不在连接的协调者节点上，则做了重定向之后，该查询后续的 fetch 流程能够少一轮 RTT 从而减少 latency，且能够节省原来的协调者节点为了此次查询所耗费的内存，CPU，网络资源。
- 缺点：查询的第一次 fetch 需要多一轮 RTT 的时间和一次解析 sql 的代价。对于小查询和带有统配符的查询来说，其解析 sql 的代价相对较大，可能做了重定向反而会导致 latency 增加。

解决思路

只对传输数据量最多的原始数据进行优化，一旦协调者节点不存在数据，则将该查询路由到一定存在数据的节点上。

结果

3 节点 1 副本原始数据查询延迟减少了 37%

测试结果

https://github.com/apache/iotdb/pull/2867

分组批量 fetch 多时间序列数据优化

问题

分布式中节点间获取查询数据时每条时间序列都会创建一个 remoteReader，且每条时间序列每次获取 batchData 都需要发送一次 rpc，给 rpc 框架造成了一定压力，且没有很好的利用网络资源。

解决思路

创建 remoteReader 时根据序列所属的不同数据组进行分组，将属于同一数据组的多条时间序列进行批量创建并批量 fetch batchData。
https://shimo.im/docs/QTHQjdGCJkYYV8cJ

结果

查询 latency 减少 30%

测试结果

https://github.com/apache/iotdb/pull/2875

TimeGenerator 批量 fetch 优化

问题

分布式情况下，`TimeGenerator` 类逐个生成符合值过滤条件的 `timestamp` ，然后再将每一个 `timestamp` 逐个交给 Select 中查询时间序列的 `Reader` 进行查询，即有多少 `timestamp` 就调用了多少次 RPC 请求。

目标场景：

1. 带 value filter 的原始数据查询中，select 子句和 where 子句中时间序列不相同的场景。
2. 带 value filter 的聚合查询

解决思路

每次生成定长（fetchsize）的 timestamp 数组，然后发 RPC 进行批量 fetch 获取一组结果，从而将 fetchsize 次 RPC 请求降低到 1次。

收益分析：

1. RPC次数： N -> N/M (M = fetchsize)
2. 副作用：
   a. 每次RPC的大小，1 -> fetchsize
   b. 协调者节点（或单机节点）临时缓存的内存大小增加

结果

带 value filter 的聚合查询，即目标场景 2，时间减少了 98%

带 value filter 的原始数据查询，即目标场景1，但是由于 benchmark 测试中 select 子句和 where 子句中时间序列相同，所以不完全符合目标场景，测试结果无明显降低即可。实际测试结果，时间减少了 **4%**

测试结果

https://shimo.im/docs/RkXYrhhvhDVcHp8C

带 ValueFilter 的聚合查询优化

问题：

问题1：与不带 value filter 的情况不同的是，带 value filter 的聚合查询并没有对**根据时间序列对一个时间序列的多个聚合函数进行分组**，如 s1 有两个聚合函数 `count` 和 `avg` ，则会创建两个 `Reader` 进行读取。

问题2：对于 select 子句和 where 子句中时间序列相同的情况，没有使用 cache 进行优化。

解决思路：

解决思路1：对时间序列进行分组查询，每个时间序列只使用一个 Reader.

解决思路2：将 TimeGenerator 中的数据 cache 从 1 扩大到 fetchsize，然后对于 select 子句和 where 子句中时间序列相同的情况，使用 cache 进行优化。

收益分析

单机和分布式情况下对于这两个问题均没有进行优化。

单机情况下

对于问题1，**假如一个 Reader 的读取损耗为 C，聚合函数的个数为 N，则总损耗为 C * N**

优化后，N 个聚合函数的总损耗降低为一个 Reader 的损耗，即 N * C –> C

对于问题2，**仍假设一个 Reader 的读取损耗为 C，则 select 和 where 子句中分别对时间序列查询的代价为 2 * C。**

优化后，不需要再对 select 子句中的时间序列重复读取，即减少一半的查询代价，2 * C –> C

分布式情况下

对于问题1，每个 Reader 除了读取损耗外，还有 RPC 的传输代价，设为 M，则**优化前的总损耗为 N * (C + M)，优化后的总损耗降低为 C + M**

对于问题2，同样包含 RPC 的传输代价，即 2 * (C + M) –> C + M

结果

因为 benchmark 内没有相应的查询场景，且因为耦合性比较高，这个问题优化的 PR 内容和上面 timeGenerator 的优化写在一起了，所以没有单独进行测试。