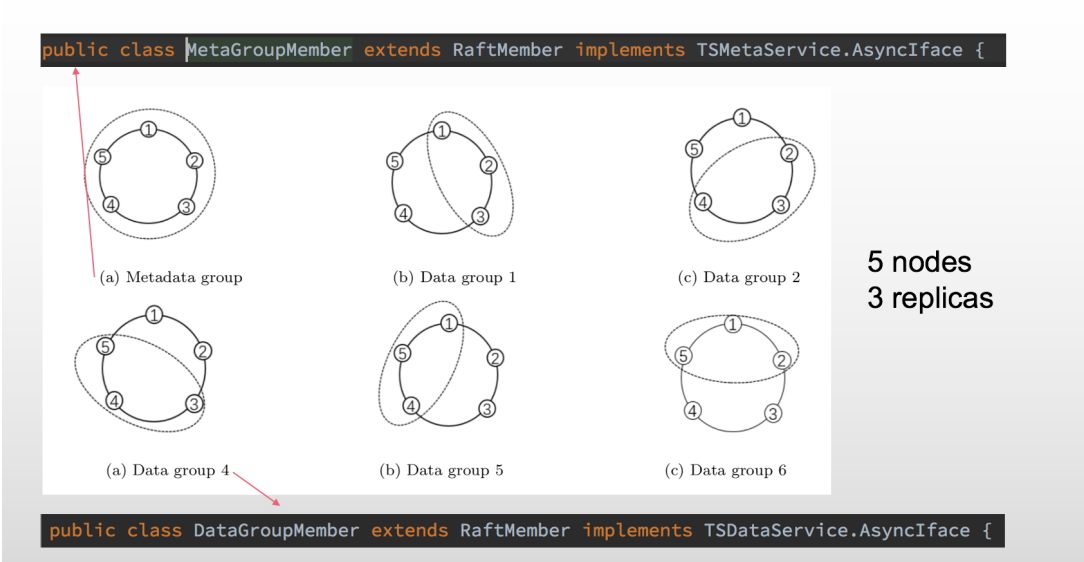
IoTDB Multi-Raft-For-One-DataGroup ...

任务目标

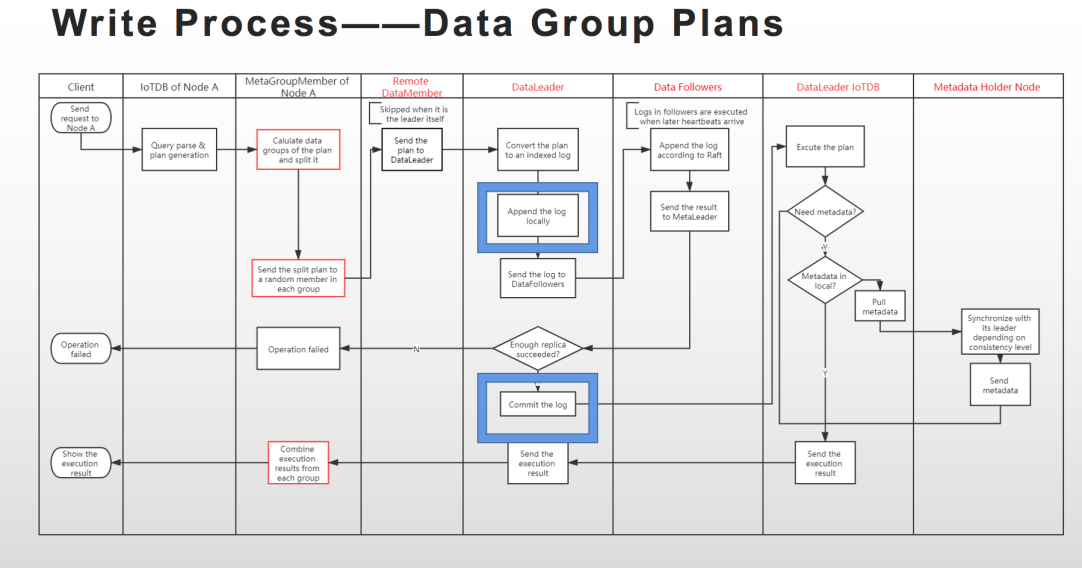
背景

目前分布式 IoTDB 采用了一致性哈希的方式来将数据分片，其架构如下图所示：

比如对于 5 节点 3 副本的集群，将会存在 1 个元数据组和 5 个数据组，其中元数据组只管理存储组等元数据，数据组会存储时间序列以及对应的元数据。当前，每个数据组为一个 raft 组，保证了数据的高可用。



对于每条 raft 日志，其在整个 raft 框架的提交过程中需要拿两次锁，第一次是编号过程，第二次是提交过程。此两个操作拿锁都是必要不可缩减的。（注：将日志 apply 到状态机的操作可以不拿锁，这也是 raft 的异步 apply 优化，其可以提升单 raft 组的吞吐量。此优化目前已实现）



然而在某些场景下，比如在单节点单副本或者两节点两副本的集群中，整个集群只存在 1 或者 2个 raft 组。此时如果并发数较高（200 以上），那么大量的客户端都会耗费大量的时间抢锁。从而对整体性能产生较大影响。比如在 TPCx-IoT 200 客户端的场景下，单节点单副本相比单机版性能下降了 1/3。通过 profile 发现大部分时间都花费在了抢锁上。此外，即使并发不高，多 raft 组的吞吐量相比单 raft 组的吞吐量也会有一定提升，具体可查看此测试文档。

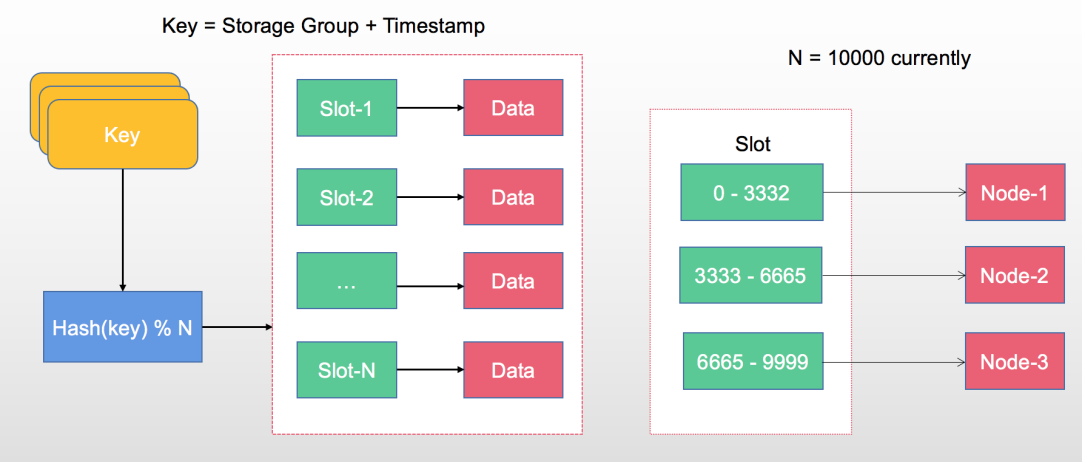
因此，需要考虑高并发情况下如何减少 n 节点 n 副本分布式相比单机版的性能下降问题。

目标

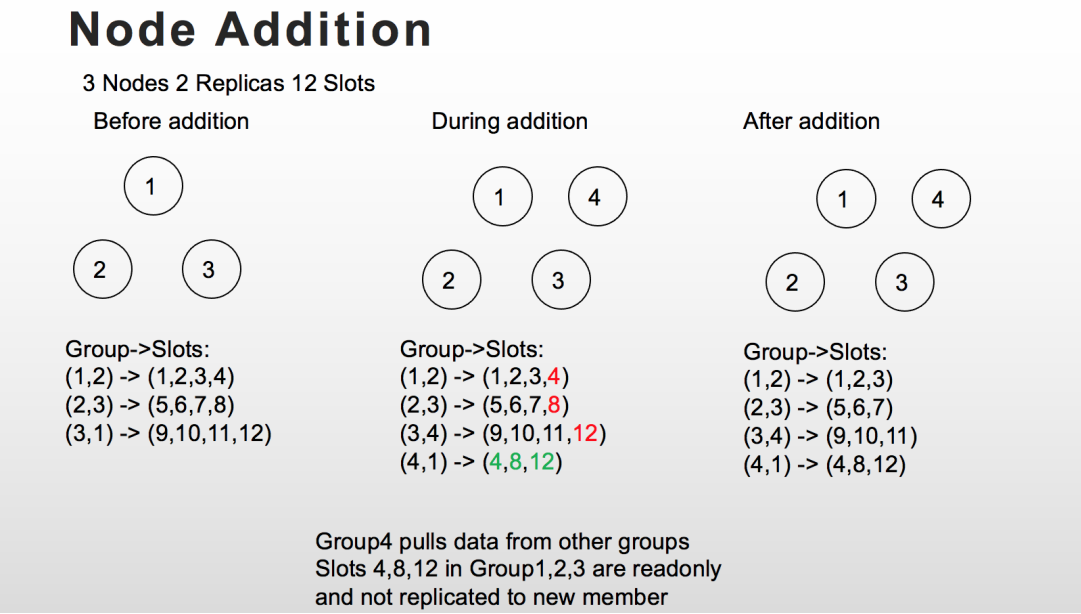
让 2 节点 2 副本的性能能够稳定达到相同配置单机的 0.85 倍以上。

相关现有实现

数据分区



目前的数据分区策略采用了 Redis Slots 的方式。即预先定义整个集群有 10000 个槽，然后将其均分到节点上，并且在节点增删时依然将其平均分到集群中。比如如果集群一共有 12 个槽，则对于 3 节点集群，其每个节点平均拥有 4 个槽。此时如果添加了一个节点，每个节点拥有的槽数就成了 3 个。示意图如下：



对于每一条数据，其会通过 sg+timestamp 来做哈希并映射到对应的槽上，由于每个节点都维护了槽和节点的对应关系，因此数据就能被分配到对应的节点上。由于槽的数目较多，因此一般情况可以认为每个节点上分配的数据相对均匀。

请求路由

目前对于读写请求，集群内部都是根据 node 来进行路由的。比如对于一个写请求，具体可分为以下 4 步：

1. 协调者节点收到客户端的请求。
2. 协调者节点根据读写请求的 sg 和 time 值映射 slot 并根据全局同步的 partitiontable 来找到对应的 partitionGroup；
3. 协调者节点判断自己是否属于此 partitionGroup，如果是，则转发此请求到此 partitionGroup 所属 raft 组的 leader 节点（可能是本节点）。如果不是，则向该 partitionGroup 的节点按序发送请求，一旦收到正确回应即可（被发往对应节点之后可能还要进行进一步转发）。
4. 汇总结果，返回给客户端。

节点如何区分 raft 组

每个节点都会在副本数个 partitionGroup 当中，当前每个 partitionGroup 都是一个 raft 组。每个节点都维护了一个 node->raft 组的 map（在 DataClusterServer 类中）。目前集群间通信的几乎所有 rpc 都会带上 header 这个变量，其是一个 partitionGroup 的唯一标识，每个节点在收到请求时会根据此 header 来判断应该属于哪个 raft 组，从而到对应的 raft 组中去查询或者写入或者更新心跳。

详细设计

需求分析和功能定义

基于以上背景，需要考虑高并发情况下如何减少分布式相比单机版的性能下降问题。因此，最直白的设计即是对于一个数据组，其不再与一个 raft 组强绑定，而是与多个 raft 组绑定。比如可以使得两节点两副本的 raft 组个数从 2 变为 4。这样对于分布式的底层实例来说，负载和之前相同；对于外部的请求来说，其竞争 raft 的锁的个数从 2 变为了 4；对于分布式来说，其 raft 组数变多了，吞吐量可能也会有一定提升。当然，由于一个 raft 组的固定线程个数在 10 个以上，所以增加 raft 组个数也会带来额外的线程切换损耗，

综上，一个数据组绑定 raft 组的个数需要是一个可配置项，这样就能够根据不同的负载来选择合适的参数，从而达到最理想的性能。

当然，更理想的实现应该是与实际负载相匹配，能够动态的去扩展 raft 组的个数而不是提前预定义。不过，这样的设计与实现会更为复杂，可以在快速实现此功能之后再进一步设计。

模块设计

基于以上背景，可以得知目前节点间发送请求是都是通过 node 来标识的。即一个节点收到请求后会根据此 node 来找到对应 partitionGroup 的 raft 组，显然这样的标识方式对于相同的 partitionGroup 是不适用的，即如果想要让一个数据组对应多个 raft 组，这样的标识是不可行的。

因此需要通过 node + id 来唯一标识一个 raft 组。

更具体的，对于每条数据，其会首先被映射到一个逻辑分区 slot，然后会根据全局统一的逻辑分区 slot 与物理节点 node 的映射来进行路由。对于映射到逻辑分区 slot 的步骤，此功能的变更不需要对其进行改动；对于逻辑分区 slot 到物理节点 node 的映射，之前是将所有 slot 平分到所有节点上，然后每个节点以其为 header 构成一个 raft 组来管理属于这些逻辑 slot 的数据。现在此功能加入后变成了首先将所有 slot 平分到所有节点上，然后每个节点将属于其的 slot 再按照一个可配置的 raft 组个数均分，然后每个节点以其为 header 构成多个 raft 组来管理属于这些逻辑 slot 的数据。

rpc 改动

改动前：

```
// leader -> follower
struct AppendEntryRequest {
  1: required long term // leader's
  2: required Node leader
  3: required long prevLogIndex
  4: required long prevLogTerm
  5: required long leaderCommit
  6: required binary entry // data
  7: optional Node header
}
...
```

改动后：

```
// leader -> follower
struct AppendEntryRequest {
  1: required long term // leader's
  2: required Node leader
  3: required long prevLogIndex
  4: required long prevLogTerm
  5: required long leaderCommit
  6: required binary entry // data
  7: optional Node header
  8: optional int raftId
}
...
```

之前每个节点是通过 node 来区分不同的 raft 组，从而根据 rpc 请求中的 header 变量来路由到本节点的不同 raft 组去处理。现在每个节点是通过 node + id 来区分不同的 raft 组，因此可以为每个 rpc 请求添加一个 raftId 变量来路由到本节点的不同 raft 组去处理。

当然，为了可扩展性，可以将 header 和 raftId 抽出来单独组成一个 rpc 中的 struct 结构体，并将这个 struct 作为 raft 组在集群中的唯一标识，之后如果再次更改标识方式，则只需修改此 struct 即可。

partitionGroup 改动

改动前：

```
public class PartitionGroup extends ArrayList<Node> {

  public PartitionGroup() {
  }

  public PartitionGroup(Node... nodes) {
    this.addAll(Arrays.asList(nodes));
  }

  public PartitionGroup(PartitionGroup other) {
    super(other);
  }

  @Override
  public boolean equals(Object o) {
    return super.equals(o);
  }

  @Override
  public int hashCode() {
    return super.hashCode();
  }

  public Node getHeader() {
    return get(0);
  }
}
```

改动后：

```
public class PartitionGroup extends ArrayList<Node> {

      private int id;

      public PartitionGroup() {
      }

      public PartitionGroup(Collection<Node> nodes) {
        this.addAll(nodes);
```

```
      }

      public PartitionGroup(int id, Node... nodes) {
        this.addAll(Arrays.asList(nodes));
        this.id = id;
      }

      public PartitionGroup(PartitionGroup other) {
        super(other);
        this.id = other.getId();
      }

      @Override
      public boolean equals(Object o) {
        if (this == o) {
          return true;
        }
        if (o == null || getClass() != o.getClass()) {
          return false;
        }
        PartitionGroup group = (PartitionGroup) o;
        return Objects.equals(id, group.getId()) &&
            super.equals(group);
      }

      @Override
      public int hashCode() {
        return Objects.hash(id, getHeader());
      }


      public Node getHeader() {
        return get(0);
      }

      public int getId() {
        return id;
      }

    }
```

为每个 partitionGroup 添加一个 id 来标识同数据组但不同 raft 组的请求。

partitionTable 接口改动

改动前：

```
public interface PartitionTable {
  /**
   * Given the storageGroupName and the timestamp, return the list of
nodes on which the storage
```

```
   * group and the corresponding time interval is managed.
   *
   * @param storageGroupName
   * @param timestamp
   * @return
   */
  PartitionGroup route(String storageGroupName, long timestamp);
  /**
   * Given the storageGroupName and the timestamp, return the header node of the partitionGroup by
   * which the storage group and the corresponding time interval is managed.
   *
   * @param storageGroupName
   * @param timestamp
   * @return
   */
  Node routeToHeaderByTime(String storageGroupName, long timestamp);
  ...
}
```

改动后：

```
public interface PartitionTable {
  /**
   * Given the storageGroupName and the timestamp, return the list of nodes on which the storage
   * group and the corresponding time interval is managed.
   *
   * @param storageGroupName
   * @param timestamp
   * @return
   */
  PartitionGroup route(String storageGroupName, long timestamp);
  /**
   * Given the storageGroupName and the timestamp, return the header node of the partitionGroup by
   * which the storage group and the corresponding time interval is managed.
   *
   * @param storageGroupName
   * @param timestamp
   * @return
   */
   Pair<Node, Integer>  routeToHeaderByTime(String storageGroupName, long timestamp);
  ...
}
```

对于查询数据物理分区的函数，有两种返回方式，分别是带有 id 的 partitionGroup 或 node 与 raftId 的 pair。

代价分析

对于一个 raft 组，其有一些线程是与负载有关的，以下仅仅列出仅与 raft 组绑定而非与实际负载绑定的线程：

```
heartBeatService 1 个 心跳线程
commitLogPool 1 个 follower 异步 apply 心跳信息的线程
appendLogThreadPool 0 或 副本数 - 1 个线程用来并行发送 appendEntries 请求。
serialToParallelPool 副本数 - 1 个线程用来序列化日志。
deleteLogFuture 1 个 logmanager 检查删除内存日志线程
checkLogApplierFuture 1 个 logmanager 检查日志 applyIndex 线程
persistLogDeleteExecutorService 1 个 logmanager 检查删除磁盘日志线程
DispatcherThread 1 个日志分发线程，开启 dispatcher 模式时才会有。
```

多 raft 组能够一定程度上提升对网络 pipeline 的利用率，但是也会带来额外的线程切换损耗。比如在此测试结果中可以看到：对于两节点两副本，两 raft 组最优性能是单 raft 组最优性能的两倍，即此时增大 raft 组数是一个完全线性横向扩展的增长曲线。当然，如果节点共拥有 200 个 raft 组，由于管理的数据量没变，但带来了大量的线程切换消耗，因此此时再增多 raft 组带来的性能提升应该微乎其微，甚至出现下降。

总之，随着 raft 组数的增多，系统整体吞吐量应理论上应该先升高再降低。

测试验证

测试目标

测试两节点两副本分布式相比单机版的性能下降幅度，其不高于 15 %。

测试方案

测试系统

TPCx-IoT 客户端

被测系统

单机版 IoTDB 和两节点两副本分布式 IoTDB。

测试环境

服务端节点：AWS i3.metal 实例：64 核 CPU 512G 内存 1.5T SSD*8 50Gb 网卡
客户端节点：AWS c5.metal 实例：96 核 CPU 192G 内存 无 SSD 25Gb 网卡

负载描述

- 负载情况：16 个 电厂（IoTDB 存储组，可调整），每个电厂 10 个 线程（IoTDB session 客户端，不可调整），每个电厂 160 个设备，每个设备一个测点，每个测点为大小 1kB 的字符串（ Snappy 压缩比 4.8 左右）。
- 总点数：TPCx-IoT 分为四个阶段：warmup->run->clear->warmup->run。其中每个阶段的点数为 300 000 000 点（300GB 数据，可调整）。

- 数据分布：总点数由所有电厂均分，时间戳从开始测试的本地时间戳开始，每个电厂独立生成 100ms 间隔的数据点，每个数据点属于一个随机设备。
- 读写负载：每一万次写有 5 次时间范围逻辑查询。每个逻辑查询对应两个物理查询，其区间均为 5 s，第一段是最后写入设备的 5s 区间，第二段是从测试开始到最后写入设备时间 100 s 前的任意 5s 区间。
- 乱序情况：warmup 阶段无乱序，run 阶段存在乱序。（warmup 写入的时间戳大于 run 阶段起始的时间戳）

参考资料

Raft 论文
Ozone如何利用Multi-Raft优化写入吞吐量
Apache Ratis中的multi-raft实现原理
字节跳动自研强一致在线KV与表格存储实践