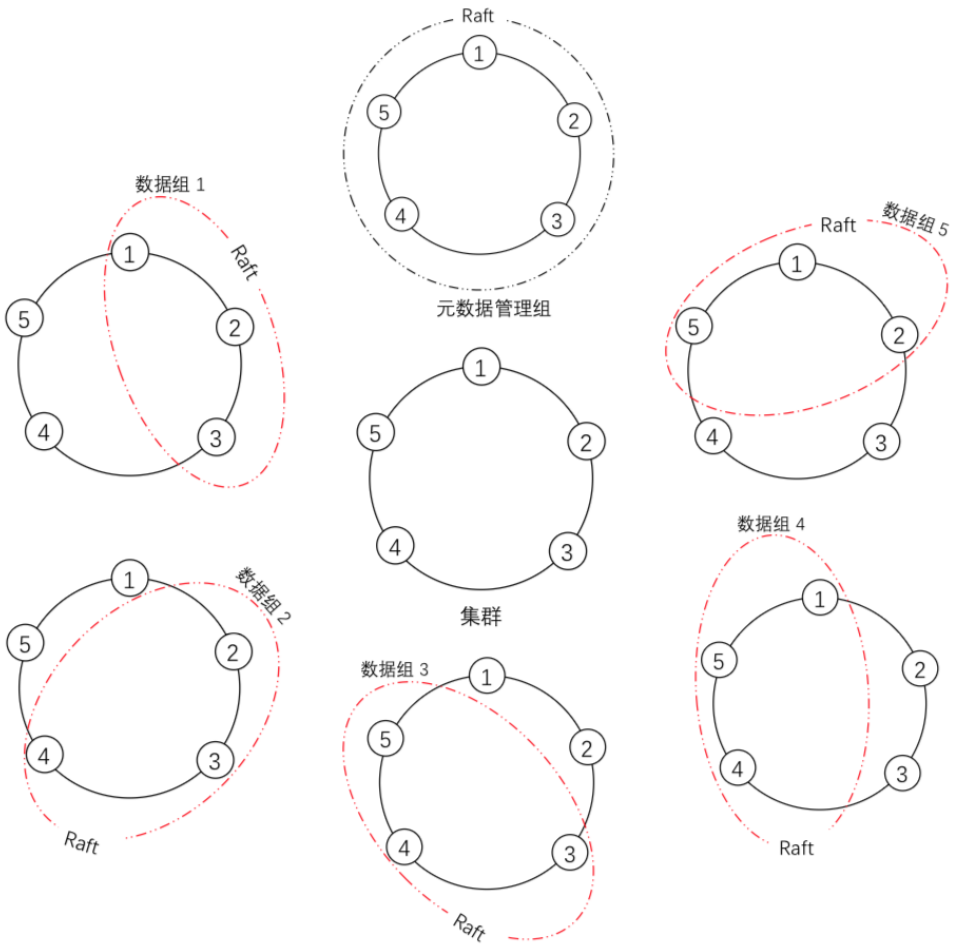
分布式自动创建 schema 功能整理

分布式自动创建 schema 功能

起因

meta组：存储组元数据
data组：时序元数据，时序数据



- 存储组元数据：所有节点均保存。
- 时序数据：由 <存储组，timePartitionId> 的粒度在集群中均匀分布，便于 scale out。
- 时序元数据：
  - 为了防止随着时序数据分区数的增多使得每个节点最终拥有全量时序元数据，现仅让 <存储组，0> 的 data 组保存对应的时序元数据在 MTree 中，其余分区仅使用一个 LRUCache<PartialPath, MeasurementMNode> 来临时保存对应时序的元数据。
  - 当非 0 时间分区的写入请求执行时，一旦单机执行出现 PathNotExisted 等错，分布式会捕捉到该错误然后尝试去 0 时间分区的 data 组拉取对应的 MeasurementMNode 并再次执行。
  - 该设计对性能是否有损耗，不同负载下 Cache 大小应设置为多大暂未有可信依据。

现状

以下几种情况需要自动创建schema：
写入数据时存储组不存在的情况，需要自动创建存储组和时间序列。
写入数据时存储组存在但时间序列不存在的情况，需要自动创建时间序列。
创建时间序列时存储组不存在，需要自动创建存储组。

实现细节

对于写入数据时存储组不存在的情况，分布式自动注册 schema 逻辑较为简单，协调者节点发现存储组不存在，注册存储组并注册对应的时间序列然后再写入即可。

对于写入数据时存储组存在但时间序列不存在的情况。分布式自动注册 schema 的逻辑需要先写入失败，抛出时间序列不存在的异常，数据组Leader检测到写入由于时序不存在而失败，接着在数据组内创建时间序列的 schema 然后再进行写入。

此外对于多副本场景：
对于自动创建 sg 和 ts 时，data 组和 meta 组可能会有顺序性问题。即对于协调者节点在 meta 组创建的 sg 和在对应 data 组创建的 ts，不同节点上的执行顺序可能不一致。对于 meta 组日志先执行的节点，data 组创建 ts 能够成功。反之，对于 data 组日志先执行的节点，其会执行失败并抛出 sg 不存在的错误，则此时需要向 leader 进行一次 sync 以确保 meta 组最新的日志成功 apply，接着再次执行创建 ts 命令。

对于自动创建 ts 并写入数据，由于其均在 data 组，因为其在不同节点上的执行顺序可以由 raft 得到保证，故不会出现顺序性问题。

PartialInsert 功能

起因

部分用户在写入数据时可能会填错某一列的数据类型或者写入不合法的测点等等，此时可以选择直接向用户报错而任何数据都不插入。
开启此参数后，某一列的错误会被记录下来最后回复给用户，但其他正常列的数据仍会被插入。

现状

在开启此参数后，在对应的 executor 中，当某一列发生错误时会标记此列但继续执行。在执行完毕后会检测目前是否有 fail 的列，如有则抛错，否则执行成功。

```java
@Override
public void insert(InsertRowPlan insertRowPlan) throws QueryProcessException {
  try {
    insertRowPlan.setMeasurementMNodes(
        new MeasurementMNode[insertRowPlan.getMeasurements().length]);
    // check whether types are match
    getSeriesSchemas(insertRowPlan);
    insertRowPlan.transferType();
    StorageEngine.getInstance().insert(insertRowPlan);
    if (insertRowPlan.getFailedMeasurements() != null) {
      checkFailedMeasurments(insertRowPlan);
    }
```

insertPlan：

```java
  /** @param index failed measurement index */
  public void markFailedMeasurementInsertion(int index, Exception e) {
    if (measurements[index] == null) {
      return;
    }
    if (failedMeasurements == null) {
      failedMeasurements = new ArrayList<>();
      failedExceptions = new ArrayList<>();
      failedIndices = new ArrayList<>();
    }
    failedMeasurements.add(measurements[index]);
    failedExceptions.add(e);
    failedIndices.add(index);
    measurements[index] = null;
  }
```

insertRowPlan：

```java
  @Override
  public void markFailedMeasurementInsertion(int index, Exception e) {
    if (measurements[index] == null) {
      return;
    }
    super.markFailedMeasurementInsertion(index, e);
    if (failedValues == null) {
      failedValues = new ArrayList<>();
    }
    failedValues.add(values[index]);
    values[index] = null;
  }
```

insertTabletPlan:

```java
  @Override
  public void markFailedMeasurementInsertion(int index, Exception e) {
    if (measurements[index] == null) {
      return;
    }
    super.markFailedMeasurementInsertion(index, e);
    if (failedColumns == null) {
      failedColumns = new ArrayList<>();
    }
    failedColumns.add(columns[index]);
    columns[index] = null;
  }
```

实现细节

在单机中该功能并不复杂，但在分布式中，由于同一个 plan 可能需要在执行失败后拉取数据或同步 leader 后再次被执行，因此重新执行时必须从 failedMeasurement 中全部恢复。

BatchPlan 接口

起因

当前 iotdb 中存在许多批量 plan，例如 CreateMultiTimeSeriesPlan，InsertMultiTabletPlan，InsertRowsOfOneDevicePlan，InsertRowsPlan 等等。这些 plan 一般都包含了多个子 plan。前面已经提到，在分布式中，同一个 plan 可能需要在执行失败后拉取数据或同步 leader 后再次被执行。对于批量 plan 也是如此，比如对于 InsertRowsPlan，其包含多个 InsertRowPlan，在一个 InsertRowPlan 执行失败后，节点会拉取数据或同步 leader 接着再重新执行操作，然而实际上可能其他 InsertRowPlan 均已经执行过，此时再执行一遍会影响性能。

现状

对于批量 plan，在 planExecutor 中执行时会判断某一子 plan 是否已经 fail 或者已经被 apply 过，如果是则不再到存储引擎中去执行。

```
@Override
public void insert(InsertRowsPlan plan) throws QueryProcessException {
  for (int i = 0; i < plan.getInsertRowPlanList().size(); i++) {
    if (plan.getResults().containsKey(i) || plan.isExecuted(i)) {
      continue;
    }
    try {
      insert(plan.getInsertRowPlanList().get(i));
    } catch (QueryProcessException e) {
      plan.getResults().put(i, RpcUtils.getStatus(e.getErrorCode(), e.getMessage()));
    }
  }
  if (!plan.getResults().isEmpty()) {
    throw new BatchProcessException(plan.getFailingStatus());
  }
}
```

```
private boolean createMultiTimeSeries(CreateMultiTimeSeriesPlan multiPlan)
    throws BatchProcessException {
  for (int i = 0; i < multiPlan.getPaths().size(); i++) {
    if (multiPlan.getResults().containsKey(i) || multiPlan.isExecuted(i)) {
      continue;
    }
    CreateTimeSeriesPlan plan =
        new CreateTimeSeriesPlan(
            multiPlan.getPaths().get(i),
            multiPlan.getDataTypes().get(i),
            multiPlan.getEncodings().get(i),
            multiPlan.getCompressors().get(i),
            multiPlan.getProps() == null ? null : multiPlan.getProps().get(i),
            multiPlan.getTags() == null ? null : multiPlan.getTags().get(i),
            multiPlan.getAttributes() == null ? null : multiPlan.getAttributes().get(i),
            multiPlan.getAlias() == null ? null : multiPlan.getAlias().get(i));
    try {
      createTimeSeries(plan);
    } catch (QueryProcessException e) {
      multiPlan.getResults().put(i, RpcUtils.getStatus(e.getErrorCode(), e.getMessage()));
    }
  }
  if (!multiPlan.getResults().isEmpty()) {
    throw new BatchProcessException(multiPlan.getFailingStatus());
  }
  return true;
```


实现细节

分布式的重新执行应该保证已经成功执行过且无错误的子 plan 不再执行，已经成功执行过但有元数据错误的子 plan 再次被执行，已经成功执行过但有其他错误的子 plan 不再被执行。因此逻辑较为复杂。

Data组通过PlanExecutor执行的BatchPlan时，执行失败抛出BatchProcessException异常。此时BatchPlan的所有SubPlan的状态均包含在BatchProcessException的FailingStatus中。BatchPlan包含isExecuted成员，用于保存SubPlan的执行情况，保证再次执行BatchPlan时可以跳过成功执行过的subPlan。对于SubPlan的执行后的状态，共有三种情况。

- SUCCESS_STATUS：执行成功
- TIMESERIES_NOT_EXIST：本地不存在时间序列，在开启时间分区时需要尝试进行schema的pull操作，重新执行
- 其他：执行错误，返回

在执行完成后，data leader根据BatchPlan的TSStatus是否为Multiple_ERROR判断是否执行成功。BatchFail后，检测SubStatus，如果存在TIMESERIES_NOT_EXIST，进行自动创建schema操作，之后重新执行。

设计方案

在自动创建schema时，具体情况分为以下几种：

写入数据时不存在存储组

1. 协调者节点拆分物理计划，获得 planGroupMap，即物理计划和数据组的映射。
2. 协调者节点判断到存储组不存在（此时planGroupMap为空），开始自动创建 schema：首先在 meta 组中创建存储组，其次到对应的 data 组中创建好时间序列，接着递归再次重新拆分物理计划，获得非空的 planGroupMap。
3. 协调者节点根据 planGroupMap 转发子计划到对应的数据组中去执行。
4. 对于每个子计划，其在各自的 data 组中进行同步并 apply（注：单副本会直接到 PlanExecutor 中去执行），执行流程如下：
   a. 判断存储组是否存在，若不存在，meta 组进行一次同步，若仍不存在，则抛出 StorageGroupNotSetException。否则存储组存在，执行子计划。
   b. 若执行抛出 PathNotExistException，则说明本地缺少对应的 schema，若开启了时间分区，尝试远程拉取 schema，重新执行该plan。若仍然抛出PathNotExistException，返回data leader，在data组中自动创建时间序列的schema（注意，由于第一次执行可能会导致 plan 中的数据被单机 PlanExecutor 中的 partialinsert 功能置空，因此重新执行前需要进行错误恢复，即使用 recoverFromFailure 方法恢复 InsertPlan）
   c. 若执行出现 BatchProcessException，根据 subPlan 的 subStatus，若不是仅包含 SUCCESS 和 TIMESERIES_NOT_EXIST，则向上返回错误。否则，针对所有 TIMESERIES_NOT_EXIST 的 plan，重复 b 中创建 schema 和拉远程 schema 的判断步骤，接着将所有的子 status 置为原位。
   d. 若执行成功，则返回 OK。
5. 整合所有子计划的结果，如全部成功则直接向客户端返回 OK，否则需要拼接错误的 status 信息确保其都能够回到原位。（注：协调者节点这里不会再尝试自动创建 schema）

写入数据时存在存储组但不存在时间序列

1. 协调者节点拆分物理计划，获得 planGroupMap，即物理计划和数据组的映射。
2. 协调者节点根据 planGroupMap 转发子计划到对应的数据组中去执行。
3. 对于每个子计划，其在各自的 data 组中进行同步并 apply（注：单副本会直接到 PlanExecutor 中去执行），执行流程如下：
   a. 判断存储组是否存在，若不存在，meta 组进行一次同步，若仍不存在，则抛出 StorageGroupNotSetException。否则存储组存在，执行子计划。
   b. 若执行抛出 PathNotExistException，则说明本地缺少对应的 schema，若开启了时间分区，尝试远程拉取 schema，重新执行该plan。若仍然抛出PathNotExistException，返回data leader，在data组中自动创建时间序列的schema（注意，由于第一次执行可能会导致 plan 中的数据被单机 PlanExecutor 中的 partialinsert 功能置空，因此重新执行前需要进行错误恢复，即使用 recoverFromFailure 方法恢复 InsertPlan）
   c. 若执行出现 BatchProcessException，根据 subPlan 的 subStatus，若不是仅包含 SUCCESS 和 TIMESERIES_NOT_EXIST，则向上返回错误。否则，针对所有 TIMESERIES_NOT_EXIST 的 plan，重复 b 中创建 schema 和拉远程 schema 的判断步骤，接着将所有的子 status 置为原位。
   d. 若执行成功，则返回 OK。
4. 整合所有子计划的结果，如全部成功则直接向客户端返回 OK，否则需要拼接错误的 status 信息确保其都能够回到原位。（注：协调者节点这里不会再尝试自动创建 schema）

创建时间序列时不存在存储组

1. 协调者节点拆分物理计划，获得 planGroupMap，即物理计划和数据组的映射。
2. 协调者节点判断到存储组不存在，开始自动创建 schema：首先在 meta 组中创建存储组，接着等待直到本节点的存储组已创建成功，最后到对应的 data 组中创建好时间序列。(注意，批量时间创建 plan 如果出错需要拼接出包含错误信息的完整 status)
3. 返回结果。

创建时间序列时存在存储组但不存在时间序列

1. 协调者节点拆分物理计划，获得 planGroupMap，即物理计划和数据组的映射。
2. 协调者节点根据 planGroupMap 转发子计划到对应的数据组中去执行。(注意，批量时间创建 plan 如果出错需要拼接出包含错误信息的完整 status)
3. 若执行出现 StorageGroupNotSetException，则在 meta 组进行一次同步，恢复 log 后再次进行 apply。
4. 返回结果。