# 备份恢复

## 背景

TDBackup：数据备份工具

网络存储：分布式网络存储系统，如hadoop, cos等

SST文件：rocksdb存储数据的文件单元，利用key在其中顺序排列的特点做了压缩

Region：TDStore管理数据的逻辑单元，表示 一段数据范围

TDRecover：数据恢复工具

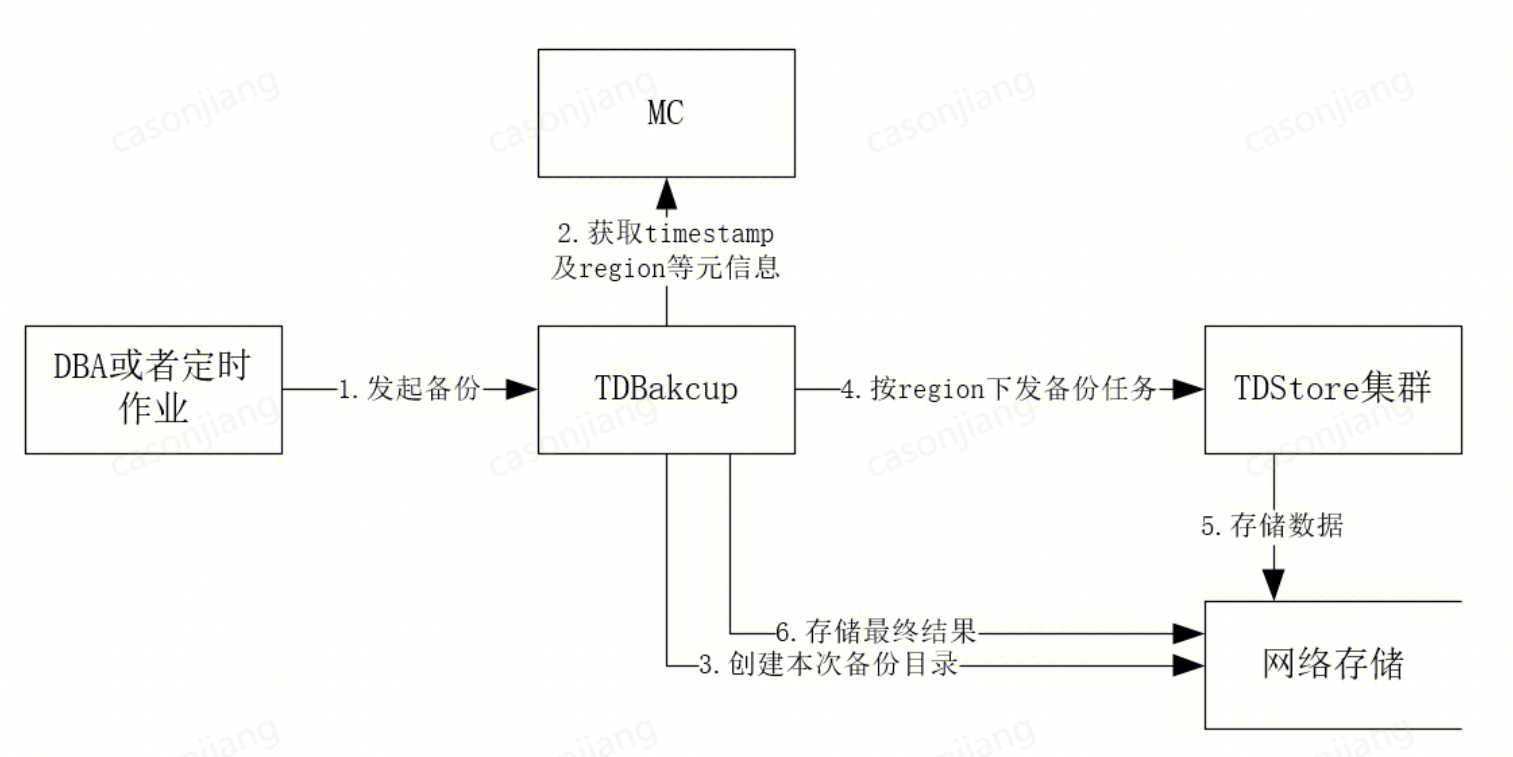
子事务：分布式事务在region上投影

## 方案

### 全量数据备份

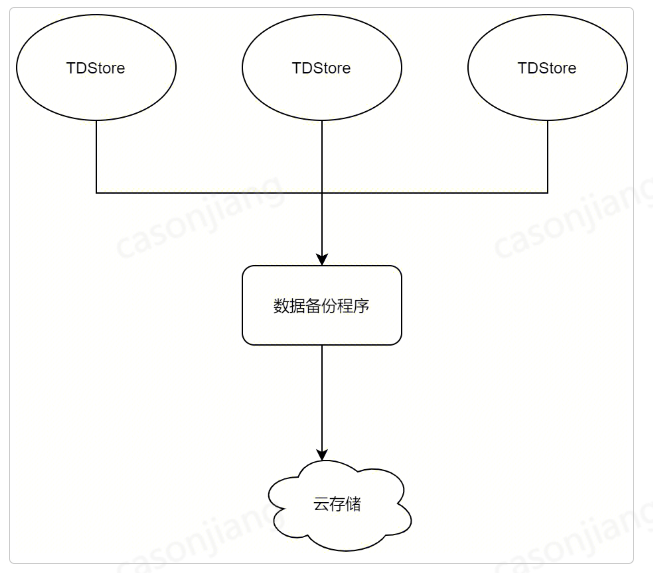
#### 概述

#### 数据流图



#### 方案选型

##### Pull模式



pull模式中，数据备份程序从各个TDStore节点中把全量（增量）数据先拉取过来，然后再转存到分布式云存储中。

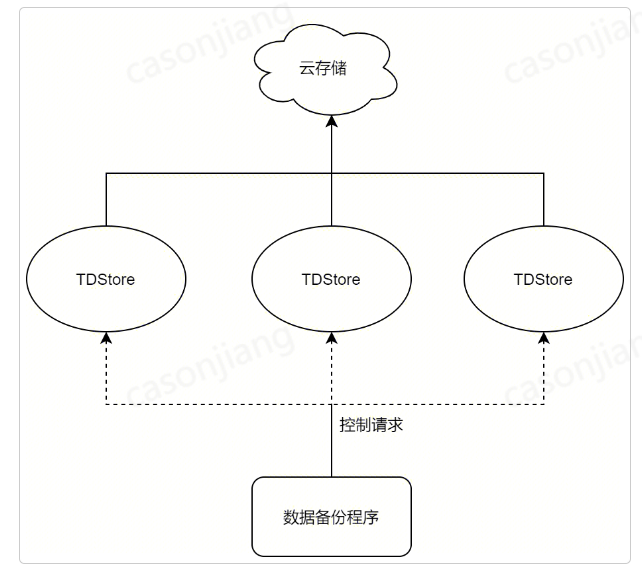
优点：将数据存储到云端的功能放在备份程序这一侧，TDStore只需提供数据拉起相关接口。这样备份的能力和TDStore解耦了，支持各种云存储不需要TDStore变更版本。

缺点：

1、数据流向出现中心节点，当集群规模比较大时，备份程序单节点将无法支撑。如果备份程序需要部署多个，那么任务划分又会比较复杂，且会给DBA运维带来复杂度。

2、全量数据备份时，可能会对TDStore节点产生流量压力，也可能会影响缓存数据。

##### push模式



push模式中，数据备份程序通过控制各个TDStore节点来让TDStore自己将备份数据存储到云端。

优点：没有数据流的中心节点，集群管理相对简单。

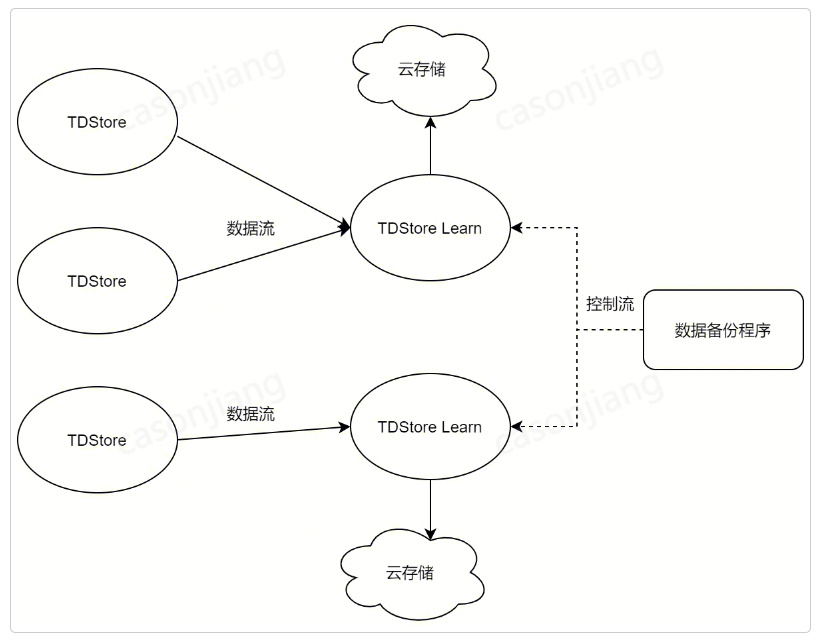
缺点：

1、TDStore需要实现存储数据到云端的功能，如果需要支持多种云端，会带来一定的复杂度，如果备份的代码出现问题，可能会影响集群稳定性；如果需要支持新的云端，需要TDStore做版本升级才能支持；

2、全量数据备份时，可能会对TDStore节点产生流量压力，也可能会影响缓存数据。

比较适合用在规模不大的集群里面，或者流量压力不大的集群。

##### learn模式



learn模式中，TDStore learn节点同步TDStore节点产生的数据，并接受数据备份程序的控制，将数据备份到云端。

优点：

1、如果需要支持新的云端，只需要升级TDStore learn节点即可。

2、功能分离到不同角色上，隔离代码影响。

缺点：

1、需要实现learn角色以及集群要有管理learn角色的功能。

2、如果leader节点要能导出一致性的全量备份，可能需要额外支持raft log index和commit ts保持偏序关系才行。

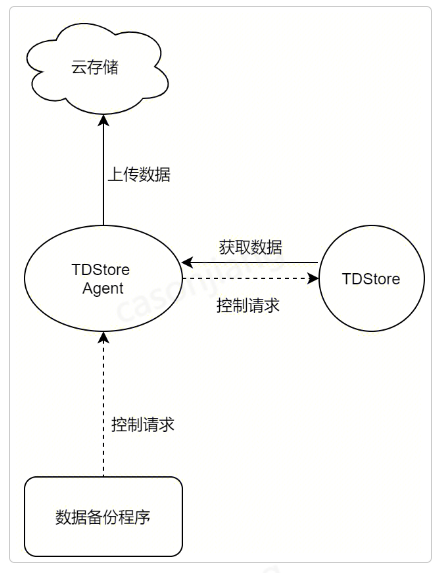
比较适合用在流量较大，或者已经有TDStore learn角色作为其它用途的集群中。

##### 混合模式

同时支持push模式和learn模式，具体行为依赖数据备份程序的控制。

##### TDStore Agent

引入TDStore Agent承担上传云存储功能，解耦数据上传部分，方便升级支持新的云存储



##### 最终形态支持

TDStore Agent+混合模式

##### 实现目标和阶段划分

实现混合模式，主要原因是部署灵活、实现曲线相对平滑。

第一阶段：

1、全量数据备份，不考虑tdstore节点故障等异常（cos）；

2、全量数据单节点恢复（cos）。

第二阶段

1、全量数据备份，支持任务中断，考虑tdstore节点故障任务重分配等异常（cos）。

2、全量数据多节点恢复（cos）。

第三阶段：

1、增量数据备份和恢复（hdfs）;

2、raft层支持learn角色，同时集群管理支持tdstore learn节点。

3、支持follow（learn）一致性读。

第四阶段：

1、更多类型的云端存储(例如hdfs等)。

2、上述过程的细节优化，例如断点续作等。

#### 基本流程

整个备份过程由一个专门的备份程序TDBackup推进，整体的基本流程：(这个备份程序也可以整合到 SQLEngine 中，由规定的SQL来触发）

1、DBA或者定时作业发起备份请求给TDBackup。

2、TDBackup获取元数据和backuptimestamp，严格按照下列顺序执行。

发送RPC请求给MC获取全量region信息列表，信息必须包括RegionLogInfo（ current\_term，log\_commit\_id ）和 RegionMeta（member list）。

从MC获取一个备份专用的timestamp和全局schema version。

向MC获取当前分配的index\_id值，以及已经分配出的index\_id对应的AutoIncrementID的值。(不需要备份这些信息，因为无法严格保证顺序，可采用恢复时从数据中生成再回填到mc中)（需要讨论下自增值是否需要放到TDStore上）

3、TDBackup在网络存储（比如hadoop, cos 等）上创建本次备份专用目录。

在网络存储中备份的根路径中新建一个目录，目录名为bakup\_${backtimestamp}\_${localtime} (路径包含mc产生的timestamp和TDBackup从计算机系统中获取的当前时间)

4、TDBackup按region下发备份rpc到TDStore，rpc request中需要带上备份timestamp。

5、TDStore收到备份rpc做如下处理：

1）tdstore检验rpc请求中的start\_key、end\_key以及region version，如果发现mis match（一般发生这种情况，说明region分裂了），回复对应的错误码，让TDBackup更新对应的region元信息信息。

2）tdstore在对应region上以备份timestamp开启备份标记，防止tdstore做compation时将backup timestamp对应快照的数据删除。（考虑是否要阻止分裂，一期可以阻止，后续迭代可以逐渐放开）

3）解析备份rpc的协议字段，转换成调用具体的网络文件系统工具，在3.a目录（即 bakup\_${backtimestamp}\_${localtime}）的下面新建名为$region\_id的目录

4）TDStore以rpc中timestamp为snapshot读取region范围内的全量数据。同时TDStore 把读取到的数据组织成 SST 文件，并把 SST 文件写到网络存储6.a创建的目录中( 这里需要流控）。

5）TDStore 将region type等元信息以及sst文件名列表以json或者其它通用配置文件保存到5.c创建的目录中（即 bakup\_${backtimestamp}\_${localtime}/$region\_id ）。

6）以最终结果+机器timestamp为文件名在 bakup\_${backtimestamp}\_${localtime}/$region\_id 创建一个文件，标记该region完结备份。

6、TDBackup统计所有region的备份情况，保证所有region都备份成功的情况下，存储备份结果信息和元信息存储到网络存储中，结束整个全量备份。

1）将本次备份相关的元信息以json或者其它通用配置文件保存到该目录下，文件名为:backup\_meta.json

2）将备份结果以文件名的形式存储backup\_failed 或者 backup\_ok

TDStore 全量备份相关RPC，协议中的Agent是保留协议扩展能力，后续如果新增分布式文件系统可以通过Agent来中转解释TDStore如何上传全量备份。

##### 全量备份RPC

rpc FullBackupTask(FullBackupTaskRequest) returns (FullBackupTaskResponse) {}

rpc QueryFullBakcupTask(QueryFullBakcupTaskRequest) returns (QueryFullBakcupTaskResponse) {}

rpc CancelFullBakcupTask(CancelFullBakcupTaskRequest) returns (CancelFullBakcupTaskResponse) {}

FullBackupTaskRequest {

tdcomm.TDCommReqHead head = 1;

uint64 region\_id = 1;

uint64 backup\_ts = 2;

bytes start\_key = 3;

bytes end\_key = 4;

uint64 region\_version = 5;

FullStorageBackend storage\_backend = 6;

}

FullBackupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

uint64 taskid =2;

}

QueryFullBakcupTaskRequest {

tdcomm.TDCommReqHead head = 1;

uint64 taskid = 2;

}

QueryFullBakcupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

bytes step = 2;

uint64 process = 3;

}

CancelFullBakcupTaskRequest{

tdcomm.TDCommReqHead head = 1;

uint64 taskid = 2;

}

CancelFullBakcupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

}

message FullStorageBackend {

oneof backend {

Local local = 1;

Hadoop hadoop = 2;

Cos cos = 3;

Aagent agent = 4;

}

}

// Local storage backend saves files into local disk

message Local {

bytes path = 1;

}

message Hadoop{

bytes endpoint = 1;

//xxxxx

}

message Cos{

bytes endpoint = 1;

//xxxxx

}

message Aagent{

bytes tool\_path = 1;

bytes tool\_params = 1;

//xxxxx

}

#### 决策点（为什么这样设计）

1、备份的一致性

3.0架构下支持数据的读一致性，以一个timestamp读取全量数据即可。

2、数据备份存放

3.0架构是多存储节点，单机存放全量数据存在风险，所以存储需要尽量放到网络存储中（网络分布式文件系统）。

3、备份的性能

由TDBackup下发任务给TDStore，直接生成sst文件后才上传到网络存储中。减少数据传递的层次，且利用上了sst文件的压缩效果。此外还可以在多个tdstore上并发进行，时间效率上也会更高。

4、恢复的性能

由于备份保存的直接是SST文件，所以恢复时可以利用TDStore IngestExternalFile的能力，避免调用kv一条一条的存储。

进一步考虑

5、备份的网络文件系统的灵活性和变更

如果由tdstore直接将生成的sst文件写到网络分布式文件系统中的话，那么如果需要适配新的备份文件系统的话，就必须要改造并升级tdstore。这里可以考虑由tdstore agent来提供将sst文件写到备份系统的功能，tdstore仅需提供一个固定的由backup timestamp生成sst之后放到本地指定目录下的功能。备份文件系统的差异的适配就有agent模块承担了。

这样TDBackup下发备份任务不再直接发给tdstore，而是先发给tdstore agent的，再由tdstore agent转发给tdstore，待tdstore将需要备份的sst文件生成好了后，tdstore agent将这些sst文件上传到网络文件系统。

同时为了方便整个集群进行备份文件系统的变更和管理，我们可以在TDBackup 和 tdstore agent的RPC协议上指定本次备份的文件系统类型，这样整个系统只需要TDBackup工具配置备份系统的类型。RPC的协议可以在路径字段定义成：local://, cos:://, hadoop:://等，代表支持不同的文件系统。

#### 实现

##### 全量数据备份方案细节

###### 基本流程(hdfs)

开发第一阶段：

整个备份过程由一个专门的备份程序TDBackup推进，整体的基本流程：

1、启动TDBackup。

2、发送请求给mc，关闭region分裂任务。

3、与DDL互斥：启动前先判断是否有正在执行的DDL，如果没有DDL任务，发送请求给mc封锁sqlengine进行DDL任务。

4、TDBackup根据配置文件和本地历史运行记录信息，决定执行全量备份，获取元数据和backuptimestamp，严格按照下列顺序执行。

1）发送RPC请求给MC获取全量region信息列表，信息包括RegionMeta（member list）。

2）发送RPC请求给MC获取glsv。

3）按region给tdstore发送获取commited raft index的rpc请求，并记录各个region应答的index。

4）从MC获取一个备份专用的backup timestamp。

5）通过一个rpc告知MC需要以backup timestamp开启备份，保证global earliest snaspshot 不大于backup timestamp，防止tdstore做compation时将backup timestamp对应快照的数据删除。

5、TDBackup在云存储上创建本次备份专用目录（cos不支持目录，但是支持key模拟目录，所以我们可以用类似的概念）。

1）在网络存储中备份的根路径中新建一个目录，目录名为${cluster\_id}/bakup\_${backtimestamp} (路径包含mc产生的timestamp)

2）创建一个region\_meta文件，并将步骤4获取的commited raft index信息已经region meta信息一起记录到region\_meta文件中，上传到云存储路径为${cluster\_id}/bakup\_${backtimestamp}/region\_meta.json。

6、TDBackup将备份任务发布到etcd上，任务信息包含backuptimestamp和region id list等元信息，并将任务状态标记为Start。

7、各个TDStoreAgent watch etcd上全量备份任务，发现有新任务发布，解析出backuptimestamp、region id list、备份的云存储类型等元信息，并根据本节点对应TDStore上region leader创建对应子任务，并标记将子任务状态标记为Start。

8、各个TDStoreAgent按region下发备份rpc到TDStore，rpc request中需要带上备份timestamp 以及 备份文件存放在本地磁盘的目录路径。

9、TDStore收到备份rpc做如下处理：

1）tdstore检验rpc请求中的start\_key、end\_key以及region version，如果发现mis match（一般发生这种情况，说明region分裂了），回复对应的错误码，让TDStoreAgent更新子任务状态，进一步让TDBackup终结整个任务（第一阶段）。

2）TDStore以rpc中timestamp为snapshot读取region范围内的全量数据。同时TDStore 把读取到的数据组织成 SST 文件，并把 SST 文件写到rpc指定的本地目录路径下，sst文件名为${region\_id}.sst（第一阶段，不考虑大小，一个region一个文件）。

3）应答rpc完成。

10、TDStoreAgent收到rpc应答后，找到对应region产生的sst文件，并上传到云存储上面，路径为${cluster\_id}/bakup\_${backtimestamp}/${region\_id}.sst。

11、TDStoreAgent将子任务状态改为Done。

12、TDBackup统计所有region的备份情况，保证所有region都备份成功的情况下，存储备份结果信息存储到云存储中，结束整个全量备份。

1）将etcd的任务更新为subtask\_done

2）按region给tdstore发送获取write fence的请求，获取各个region的write fence，并将信息记录到write\_fence.json中，然后上传到云存储中路径为${cluster\_id}/bakup\_${backtimestamp}/write\_fence.json。

3）将etcd的任务更新为writefence\_done

4）将备份结果以文件名的形式存储backup\_failed 或者 backup\_ok

5）将etcd的任务更新为backup\_done

13、发送请求给mc，解封sqlengine ddl任务。

14、发生请求给mc，允许region分裂。

15、将etcd的任务更新为all\_done

##### 全量数据恢复方案细节

###### 基本流程

开发第一阶段：

整个恢复过程由一个专门的恢复程序TDRcover推进，整体的基本流程：

1、DBA新建一个不包括sqlengine的集群，tdstore节点数等于副本数。

2、DBA运行TDRecover工具，需指定恢复目标集群（mc地址即可），还需要指定恢复的截止时间戳 或者 截止时间点 。

3、TDRecover根据指定的参数在备份云存储中寻找合适的全量备份数据，并检验全量备份的数据是否完整（一般是region的数据范围是否全域，全量备份的结果是否为success）。

4、TDRecover发rpc给MC，获取所有注册的tdstore信息(使用rpc GetAllTDStores既可)，并检验tdstore容量是否足够（例如是否大于全量备份数据的总大小的120%）（这里可以预留参数进行控制）。

5、TDRecover根据cluster\_meta信息发rpc给MC，恢复glsv信息。

6、TDRecover根据cluster\_meta信息发rpc给TDStore，恢复write fence相关信息。

7、TDRecover从云存储上获取上次备份的原信息，并发布恢复任务到etcd上，发布任务包含sst分布信息，状态为start。

8、TDStoreAagent watch到恢复任务，根据当前TDStore的region分布，写恢复子任务到etcd上，状态为start。

9、TDStoreAagent 下拉对应的sst文件到本地磁盘。

10、TDStoreAagent 发送rpc给TDStore从本地磁盘中加载指定sst，TDStore完成后应答TDStoreAagent成功。

11、TDStoreAagent 更新子任务状态为DONE。

12、TDRecover watch到所有子任务完成，根据云存储上write\_fence.json向tdstore补齐write fence信息。

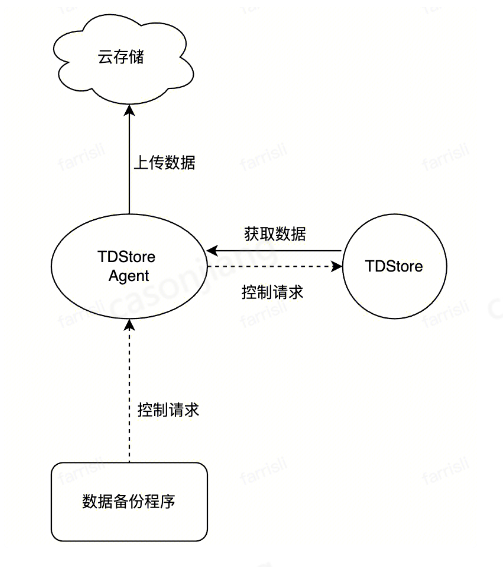
13、更新恢复任务状态为DONE，结束恢复，输出结果信息。

##### 全量备份和恢复TDStore适配点

1、需要支持一个根据timestamp读取数据并将这些读取的数据转换成sst，写到指定路径的rpc。

2、需要支持一个从指定路径加载sst的rpc.

数据备份与恢复最终选型方案



###### 设计目标

一期主要实现基础功能，打通基础流程，保证数据正确性。支持全量数据备份，不考虑tdstore节点故障等异常，支持三副本/单副本 单region恢复

二期支持cos+代码和性能优化

三期支持多副本多region恢复，考虑TDStore节点的容灾+代码和性能优化

###### 现网部署

远端存储->hdfs：三节点

ETCD：三节点

备份集群（TDStore agent必须部署）：任意部署方式

恢复集群（TDStore agent必须部署）：单副本单region或者三副本单region（尽量部署成副本数和tdstore节点数量一致）

###### 现网执行备份与恢复的方式

备份

1、搭建远端存储环境（hdfs/cos）

2、搭建etcd

3、部署TDStore agent

4、编写备份工具配置文件（远端存储信息，备份集群信息，etcd信息等）

5、执行备份工具

恢复

1、确认远端存储环境（hdfs/cos）

2、搭建etcd

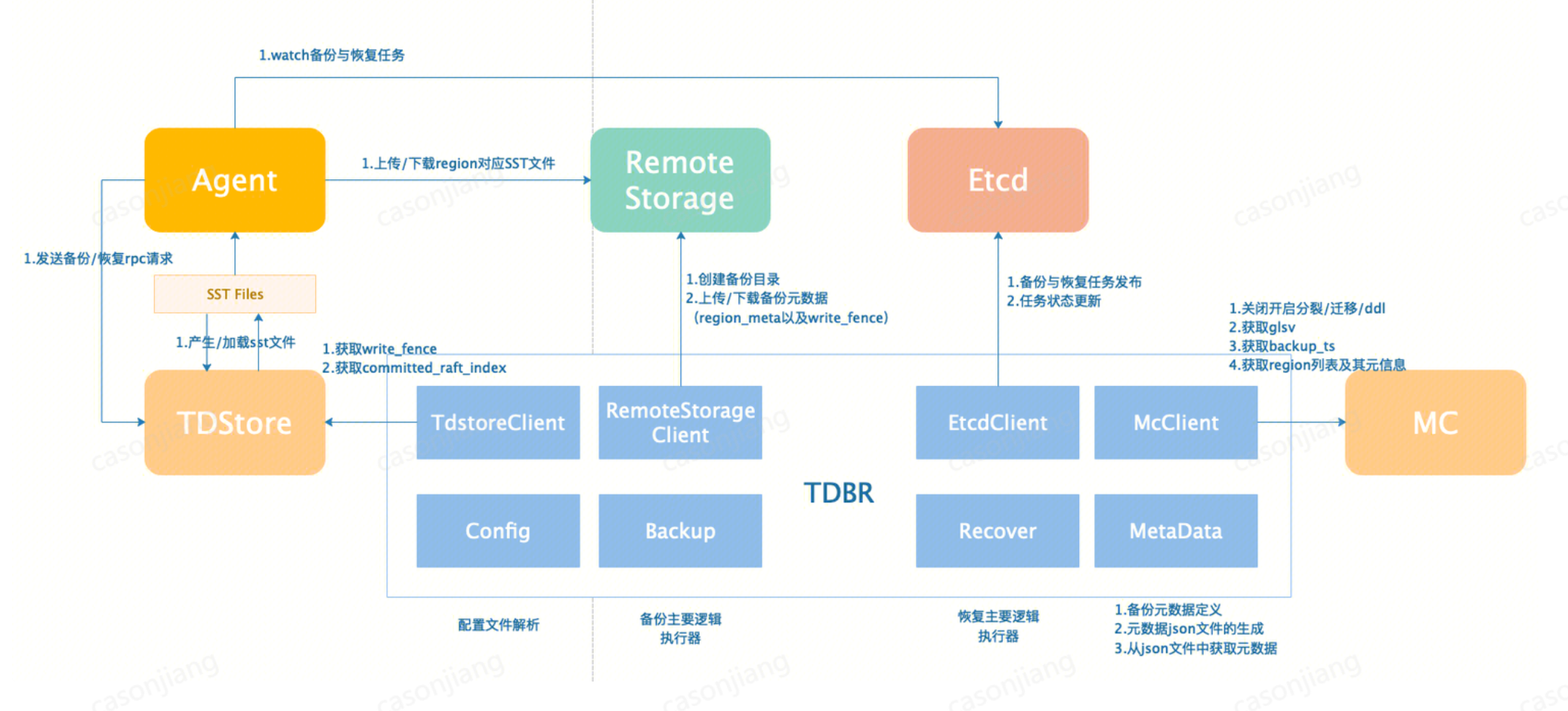
3、部署TDStore agent

4、搭建恢复集群

5、编写恢复工具配置文件（远端存储信息，恢复集群信息，etcd信息等）

6、执行恢复工具

###### TDBR框架设计



备份与恢复工具TDBR核心功能是与多个模块的通信，分别由对应的client负责，功能包括

1、元数据的获取与上传

2、元数据的下载与恢复

3、备份与恢复任务的发布

TDStore Agent需要新增以下功能

1、watch etcd上的备份与恢复任务

2、请求TDStore进行sst文件的生成和加载

3、上传和下载sst文件到远端存储

TDBR Data Validation

1、备份集群与恢复集群的数据一致性校验

###### 实现细节

备份与恢复流程已经在全量数据备份与恢复方案中描述，该小节主要对具体流程中的细节以及可能出错的地方进行阐述

分裂与迁移

一期实现主要实现基础功能以及保证数据正确性，暂时不考虑分裂、迁移并发带来的问题。因此，我们会在备份开始前禁掉分裂和迁移，并且等待正在执行的分裂和迁移全部结束。在恢复开始前禁掉分裂。

DDL

为了保证备份过程中，write\_fence元数据不会变更，我们会在备份开始前禁掉ddl任务，并且等待正在执行的ddl任务全部结束。mc需要提供关闭ddl的接口，查询是否还有正在执行的ddl任务将通过TDBR访问sqlengine采用sql语句实现

备份元数据（BackupMeta、WriteFenceMeta）

元数据格式

1、region列表，通过GetAllRegions接口从mc获取

2、glsv，mc需要新增获取和恢复glsv的接口

3、所有region对应的committed\_index，通过GetTDStoreRaftNodeInfo接口从tdstore获取

4、所有region对应的writefence信息

BackupMeta

type BackupMeta struct {

BackupRegionMetaList []BackupRegionMeta `json:"backup\_region\_meta\_list"`

GlobalSchemaVersion int64 `json:"global\_schema\_version"`

}

type BackupRegionMeta struct {

RegionId uint64 `json:"region\_id"`

RegionInfo \*metarpc.RegionInfoDisplay `json:"region\_info"`

CommittedIndex int64 `json:"committed\_index"`

}

type RegionInfo struct {

RegionId uint64 `json:"region\_id"`

StartKey []byte `json:"start\_key"`

EndKey []byte `json:"end\_key"`

RegionType metarpc.RegionType `json:"region\_type"`

RegionState string `json:"region\_state"`

RegionLevel int64 `json:"region\_level"`

RegionVersion int64 `json:"region\_version"`

MemberVersion int64 `json:"member\_version"`

DataVersion int64 `json:"data\_version"`

SchemaVersion int64 `json:"schema\_version"`

Quorum uint32 `json:"quorum"`

LeaderTdstoreNodeId string `json:"leader\_tdstore\_node\_id"`

LeaderTdstoreIpPort string `json:"leader\_tdstore\_ip\_port"`

}

type WriteFenceMeta struct {

RegionId uint64 `json:"region\_id"`

RegionType metarpc.RegionType `json:"region\_type"`

WriteFenceList []tdstore.WriteFenceInfo `json:"write\_fence\_meta\_list"`

}

用途

BackupMeta结构体填充完毕后

1、持久化成backup\_meta.json和write\_fence\_meta.json文件，上传至远端存储

2、恢复程序从远端下载backup\_meta.json和write\_fence\_meta.json文件到本地，获取region列表信息、glsv信息以及writefence信息进行恢复

备份任务元数据（BackupTaskMeta）

备份任务元数据格式

1、region列表以及对应region的commited\_index

2、backup\_ts，mc需要新增获取backup\_ts的接口，获取backup\_ts的时候，意味着备份任务开始了，备份期间，mc需要保证global\_earliest\_snapshot要一直小于backup\_ts

3、远端存储的元信息，包括远端存储类型（hdfs、cos），host，user以及备份路径

type BackupTaskMeta struct {

BackupRegionMetaList []BackupRegionMeta `json:"backup\_region\_meta\_list"`

BackupTS uint64 `json:"backup\_ts"`

RemoteStorageMeta RemoteStorageMeta `json:"remote\_storage\_meta"`

}

type BackupRegionMeta struct {

RegionId uint64 `json:"region\_id"`

RegionInfo \*metarpc.RegionInfoDisplay `json:"region\_info"`

CommittedIndex int64 `json:"committed\_index"`

}

type RemoteStorageConfig struct {

Type string `toml:"type"`

Host string `toml:"host"`

User string `toml:"user"`

}

type RemoteStorageMeta struct {

RemoteStorageConfig RemoteStorageConfig `json:"remote\_storage\_config"`

BackupDir string `json:"root\_path"`

}

用途

BackupTaskMeta结构体填充完毕后

1、作为发往etcd备份任务的元数据

恢复任务元数据（RecoverTaskMeta）

元数据格式

1、远端存储的元信息

2、要恢复的tdstore地址列表

恢复元数据

type RecoverMeta struct {

RemoteStorageMeta RemoteStorageMeta `json:"remote\_storage\_meta"`

RecoverTdstoreList []string `json:"recover\_tdstore\_list"`

}

type RemoteStorageConfig struct {

Type string `toml:"type"`

Host string `toml:"host"`

User string `toml:"user"`

}

type RemoteStorageMeta struct {

RemoteStorageConfig RemoteStorageConfig `json:"remote\_storage\_config"`

BackupDir string `json:"root\_path"`

}

用途

RecoverMete结构体填充完毕后

1、作为发往etcd恢复任务的元数据

远端存储的数据备份目录以及数据文件

备份路径

/${cluster\_group}/${cluster\_id}/backup\_${backup\_ts}，例如/test/farrisli-1/backup\_123456789

备份文件

1、各个region对应的sst文件 ${region\_id}\_${region\_type}.sst

2、备份元文件（BackupMeta结构体的持久化数据） backup\_meta.json

所有region的元信息

glsv元信息

3、备份元文件（WriteFenceMeta结构体的持久化数据）write\_fence\_meta.json

所有region对应的writefence信息

Etcd中的任务发布路径

根目录：rootPath = "/${cluster\_group}/${cluster\_id}/tdbr"

备份任务

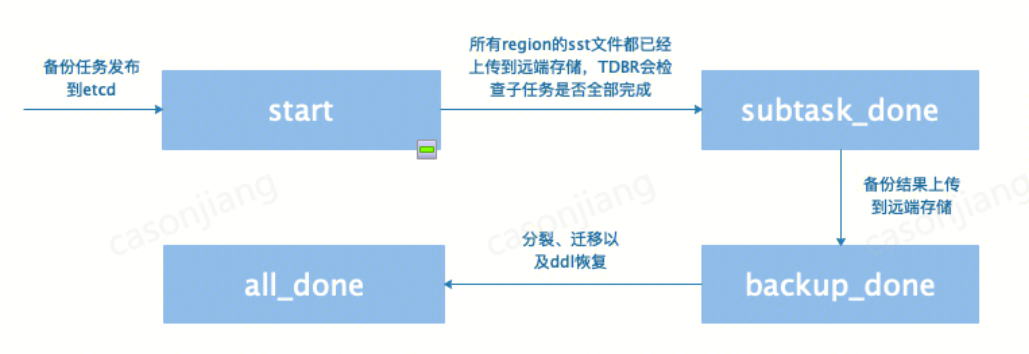
key：rootPath + "/backup\_task\_meta"

value：BackupMeta

备份任务状态

key：rootPath + "/backup\_task\_status"

value：start、subtask\_done、backup\_done、all\_done



备份子任务

key：rootPath + "/backup\_subtask\_status\_${region\_id}"

value: start、success、failed

恢复任务

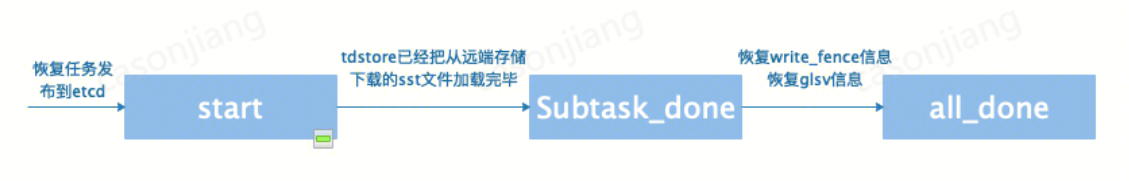
key：rootPath + "/recover\_task\_meta"

value：RecoverMeta

恢复任务状态

key：rootPath + "/recover\_task\_status"

value：start、subtask\_done、all\_done



恢复子任务

key：rootPath + “/recover\_subtask\_status\_${tdstore\_ip\_port}”

value: start、success、failed

TDStore agent从etcd收到备份任务后的行为

1、解析BackupTaskMeta，获得region列表、远端存储信息、backup\_ts等元数据

2、遍历region列表，如果当前tdstore是该region的leader，首先更新etcd的region子备份任务状态为start，依次然后发送BackupRegionData到tdstore

响应成功，继续

响应失败，发送failed状态到etcd更新子任务

将响应成功的region放入successRegionList

3、遍历生成的sst列表，将在successRegionList的region sst依次上传到远端存储

响应成功，发送success状态到etcd更新子任务

响应失败，发送failed状态到etcd更新子任务

TDBR统计所有region备份结果的行为

概述

TDBR发布备份或者恢复任务后，周期性的到etcd进行所有region子备份任务结果的检查。检查过程中，可能会仅仅重新检查，可能会重新颁发任务，也可能全部成功，由参数onlyRecheck和needReissue控制。

每次检查结果前和颁发任务后会先sleep 30s，让任务执行一会，避免一般情况下，任务没有执行完，就去检查结果的无效消耗。

核心思想是尽量等待未决的子任务更新最终结果->成功或者失败，大部分子任务出了结果后，将除了成功之外的所有子任务（正在执行的+失败的）进行子任务的重新颁发

具体策略

子任务状态全部检查完后，记录正在执行的子任务数量（doingSubtaskNum，包括start和null两种状态），失败的子任务数量（failSubtaskNum，包括fail和unknown两种状态），

成功的子任务数量（succSubtaskNum，包括succ一种状态）以及需要重新颁发子任务的列表（needReissueList，包括start，fail，unknown三种状态）

1、如果所有子任务状态都是正在执行或者成功，那么我们等待后重新检查，onlyRecheck=true

2、如果所有子任务状态都是成功，那么我们不重新检查也不重新颁发任务直接返回成功，onlyRecheck=false，len(needReissueList)=0

3、如果所有子任务状态不全是正在执行和成功，有失败

正在做的子任务数量 > 总任务数量的三分之一，继续等待后重新检查，onlyRecheck=true

正在做的子任务数量 <= 总任务数量的三分之一，说明大部分子任务都是有结果的，这时候我们对于除了成功以外的所有子任务进行重新颁发，onlyRecheck=false，len(needReissueList)>0

4、上层函数对于onlyRecheck有最大次数限制，超过了最大重试次数会进行任务的重新颁发

5、上层函数对于总的重试次数有最大限制，如果超过了总的最大重试次数，则任务失败

集群恢复

一期实现以打通基础功能为主，做好write\_fence映射的前提下，先实现单节点、单region的恢复

1、恢复集群启动无sqlengine的，单mc单tdstore1副本的集群，集群启动成功后，启动恢复进程

2、恢复进程发送恢复任务到etcd，更新恢复任务状态为start

3、agent watch到恢复任务，从恢复任务中解析出远端存储的元数据

4、agent从远端存储下载所有备份好的region对应的sst文件

5、agent下发恢复rpc到tdstore，加载sst文件，将所有region数据写到恢复集群中的单个region当中

6、所有的sst文件加载完毕，恢复任务完成，更新恢复任务状态为subtask\_done

7、恢复进程恢复write\_fence信息，glsv信息以及SetDDInitState为已初始化状态

8、更新恢复任务状态为all\_done，本次恢复结束

write\_fence恢复

write\_fence对应着一张SQLEngine的表以及表在tdstore中存储的数据范围

一个region可能会涉及到多个表的数据，因此，一个region对应着多个write\_fence

由于region分裂会切断数据范围，write\_fence的范围也会被切断

恢复时，我们并不会创建备份集群的所有region，而是把所有数据直接恢复到一个region中。

因此，备份时存储的write\_fence信息由于分裂可能被存放到多个region上，恢复时，我们需要把write\_fence被切断前的范围合并还原。

并发度

1、负责实际执行sst的生成与加载的是agent，每一个tdstore有一个自己的agent。向etcd颁发备份与恢复任务的时候，所有的agent都会同时watch到该任务。因此，对于不同tdstore，备份与恢复任务是并行的。

对于备份任务来说，每个tdstore只执行leader region的子任务，多个tdstore会分散leader，因而可以并行备份

对于恢复任务来说，每个tdstore都要加载所有region的sst文件

2、备份任务对于同一个tdstore的多个region leader，恢复任务对于同一个tdstore的多个region，一期的实现是串行的。

二期优化考虑改为并发执行，但是同时也要考虑，对于tdstore的压力，选取合适的并发数量。

大数据量region的备份

极端情况下，备份时，某个region的数据量可能会非常大，这里采取两个优化措施

1、上传和下载数据量比较大的sst时，将sst文件内容分段缓存，分多次进行上传和下载，防止一次网络请求传输的数据量太大导致网络拥堵或卡死的情况

2、tdstore生成sst文件的时候采用zstd高压缩比压缩算法，尽量减少sst的大小

region数量较大的备份

region数量比较多时

1、备份任务元数据会变大，导致超过etcd默认的消息大小

调整参数

精简备份任务元数据

分批发送任务，agent改用基于任务队列的消费者生产者模式

2、从tdstore获取raft node info返回的数据量会变大，导致执行过程会卡死（grpc的client和brpc的server）

不从tdstore一次性获取大量数据

###### 测试工作

测试场景

1、备份集群prepare数据完成，run的时候开始备份与恢复。结束后，数据一致性+恢复集群sqlengine启动+恢复集群跑测。主要测试有压力的情况下，备份与恢复流程是否正常。

2、备份集群prepare数据完成，run之前开始备份与恢复。结束后，数据一致性+恢复集群sqlengine启动+恢复集群跑测。主要测试单个region有很大数据量的情况下，备份与恢复流程是否正常。

3、备份集群prepare数据完成，run之后开始备份与恢复。结束后，结束后，数据一致性+恢复集群sqlengine启动+恢复集群跑测。主要测试region数量很多的情况下，备份与恢复流程是否正常。

备注：

恢复集群的启动方式

1、install all

2、stop sqlengine

3、rebuild mc，tdstore

测试过程

本地测试

sysbench prepare1000w行数据，sst大小：1.8G，user region最大大小：1G，region数量：3

备份与恢复成功，恢复集群sqlengine启动成功，跑测成功

sysbench prepare10亿行数据，sst大小：105G，user region最大大小：800M，region数量：184

备份与恢复成功，备份跑了两个半小时，恢复跑了5分钟。恢复集群启动成功，跑测成功

备份比较慢的原因：

TDStore生成sst文件比较慢(600M的sst大约需要1分钟)，测试的时候备份集群是单节点的TDStore，多个TDStore的话可以分担压力

单个TDStore生成sst文件是串行的

###### 未来的工作

1、TDBR与MC的心跳机制，防止TDBR异常退出，导致MC一直没有恢复备份状态

2、TDBR的各种异常处理，例如tdstore被kill

3、TDBR监控子任务执行结果由轮询改为etcd watch实现

4、同一个tdstore多个region的备份采用多协程方式

5、大数据量region备份的优化

6、支持多个备份任务并发执行，需要在协议中添加own字段，标识各个备份任务。

7、TDBR添加单元测试

8、将元数据格式改为pb，有利于传输

##### 全量备份和恢复MC适配点

1、支持恢复保存的timestamp，glsv等。

2、支持保持global earliest snaspshot 不大于backup timestamp的rpc。

##### 具体编码

###### 开发语言

TDBackup、TDRecover、TDStoreAgent采用go语言开发

TDStore是C++开发

###### TDBackup

从下面流程交互图中可以看出与mc、TDStore、云存储、etcd都有交互。需要封装出各个交互子模块。

图示

描述已自动生成

###### TDRecover

同样与mc、TDStore、云存储、etcd都有交互。同为go语言开发，跟TDBackup在同一个git项目中，共用各个交互子模块。

###### TDStoreAgent

主要任务是：

1、监听etcd的备份恢复任务；

2、从云存储中上传和下载sst文件

3、发生rpc给tdstore执行产生或者加载sst的请求。

同为go语言开发，可以跟TDBackup、TDRecover在同一个git项目中，共用各个交互子模块。

###### TDStore

主要新增任务是：

1、接收rpc请求，读取对应region，对应snapshot的数据，生成sst到指定路径。

2、接收rpc请求，加载指定路径的sst数据。

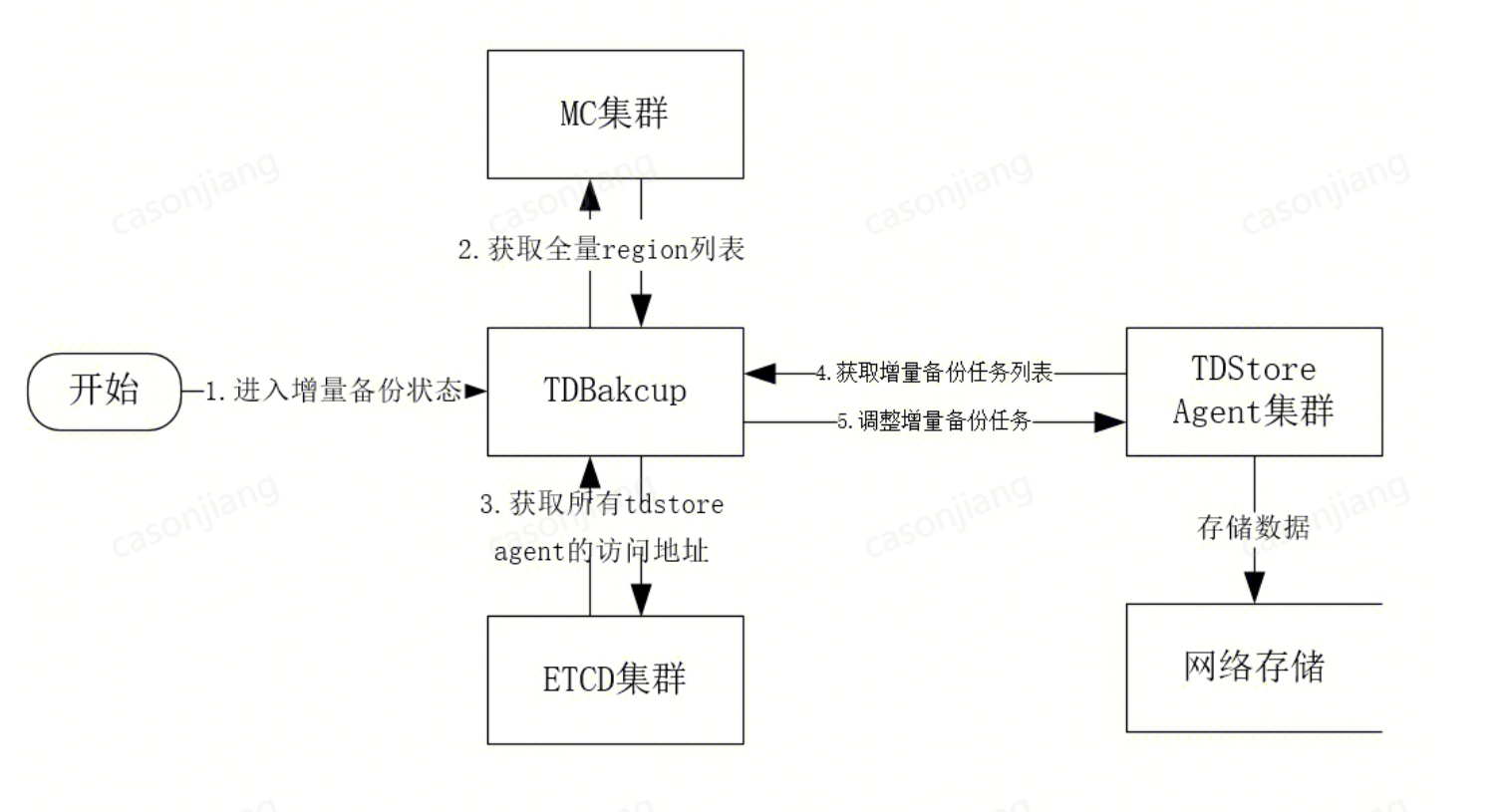
其中生成sst时，需要像scan一样，给memory lock加read scope的操作，确保能读到的数据都提交了。

### 增量数据备份

#### 概述

全量数据备份是系统某一个瞬时的快照，仅仅依赖这些离散的瞬时快照是无法还原系统任意时间点的状态的。所以我们还需要把系统产生的日志，进行实时备份，这样就可以在全量备份的基础上，回放这些日志来达到恢复系统任意时刻状态的目的。

#### 数据流图



#### 基本流程

整个增量备份过程仍然由备份程序TDBackup维护，整体的基本流程：

1、TDBackup进入执行增量备份状态

2、TDBackup在etcd上注册访问地址并进行master选举

3、启动一个线程监听访问地址，目的是动态感知TDStoreAgent发来的region变更和增量备份任务的消息。

4、从mc获取region全量信息

5、从etcd中获取所有tdstore agent的访问信息

6、依次访问tdstore agent询问增量备份任务列表，该列表的每项包括region\_id, raft index，status等。

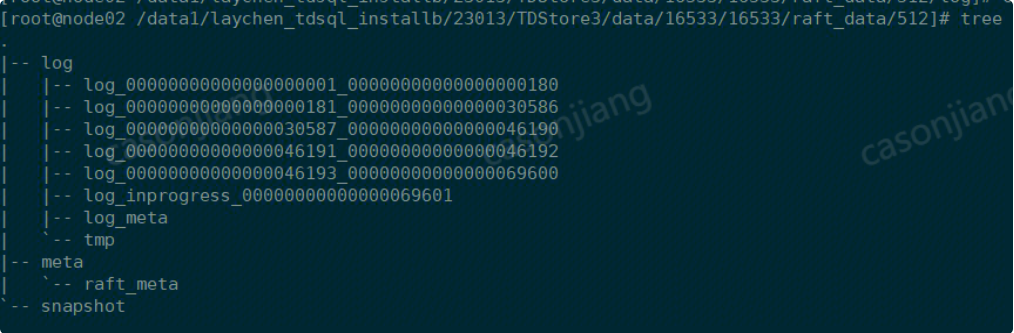
7、对比2和4的结果，发现region不存在备份任务 或者 存在region对应的备份任务status有问题等，调整region相关的增量备份任务（一般步骤是取消/新增某个tdstore agent上的增量备份任务，这里也可以做负载均衡）。

8、睡眠30min后，重新进入步骤2，睡眠过程需服务监听线程会处理TDStoreAgent发送的NotificationRequest，并进行region的局部变更

新增Region变化的watch机制：

TDStoreAgent本身因为要上报监控信息到etcd上，所以会定期获取本地TDStore上的region 信息，将每次获取的region信息与上次的做对比，发现有region新增时直接通知TDBackup。此外如果有任务遇到不可修复性问题，也可以直接通知TDBackup。

TDStoreAgent上传日志流程与互斥机制：



目前TDStore Raft日志的存储如上图所示，Raft日志会被分解成一段一段的，log\_00000000000000000001\_00000000000000000180这种有起始index和结束index，表示已经写完了的文件；log\_inprogress\_00000000000000069601仅包含起始index表示正在append的文件。不管是已经写完的文件还是正在append的文件，都有可能存在还没有commited的日志。为了避免上传了未被commited的日志，TDStoreAgent需要解析Raft log文件。

同时某些极端情况下，可能会出现两个以上的TDStoreAgent同时对同一个region进行增量备份的情况（有一个机器出现高负载，导致TDBackup联系不上这个tdstore agent的情况下，可能会在取消任务没有成功的情况下，下发新增任务给其它tdstore agent），所以tdstore agent在实现增量备份上传功能时需要考虑好互斥。互斥操作可能跟不同的备份文件系统有关，但是一般情况下，可以设定成一个包含TDStoreAgent host的文件名。

整个上传日志的具体流程如下：

1、收到备份上传任务

2、检测备份文件系统的region增量备份目录上是否存在其它节点的独占信息，存在的话删除该独占信息

3、在备份文件系统的region增量备份目录上创建属于自己的独占信息

4、初始化last\_commited\_index = 0

5、从TDStore获取最新的new\_commited\_index

6、上传last\_commited\_index到new\_commited\_index，如果出错跳转到步骤9

如果文件是done类型，且起始index和结束index被包含在last\_commited\_index到new\_commited\_index区域，直接将整个文件上传

否则对文件进行解析，并上传到new\_commited\_index位置 或者 文件末尾，如果到达文件末尾处理下一个文件

7、将本次new\_commited\_index记录在备份文件系统上，如果出错跳转到步骤9

8、令last\_commited\_index=new\_commited\_index，并检查自己的独占信息是否存在，存在则跳转到步骤5，不存在进入步骤9

9、将上传任务设置成error，并注明原因，如果是备份类文件异常错误，清理掉自己的独占信息

TDStoreAgent任务表：

由于TDBackup会定期查询TDStoreAgent上的增量备份任务，所以要求任务在失败后不能立即销毁，所以TDStoreAgent需要两个任务表，一个是active task map, 另一个是history task map。任务失败或者取消后，从active task map转移到history task map中。然后再根据一定的策略，例如保存多长时间，或者多少个这种条件，定期删除。

##### TDBackup Watch进制相关RPC

TDBackup Watc机制相关RPC

rpc Notification(NotificationRequest) returns (NotificationResponse) {}

NotificationRequest {

tdcomm.TDCommReqHead head = 1;

uint64 region\_id = 1;

NotificationEvent event = 2;

bytes description = 3;

}

NotificationResponse{

tdcomm.TDCommAnsHead head = 1;

}

enum NotificationEvent {

RegionAdd = 1;

RegionIncrementalBakcupTaskFaild = 2;

}

TDStoreAgent 增量备份相关RPC

##### 增量备份相关RPC

rpc IncrementalBackupTask(IncrementalBackupTaskRequest) returns (IncrementalBackupTaskResponse) {}

rpc QueryIncrementalBakcupTask(QueryIncrementalBakcupTaskRequest) returns (QueryIncrementalBakcupTaskResponse) {}

rpc CancelIncrementalBakcupTask(CancelIncrementalBakcupTaskRequest) returns (CancelIncrementalBakcupTaskResponse) {}

rpc QueryAllIncrementalBakcupTask( QueryAllIncrementalBakcupTaskRequest) returns ( QueryAllIncrementalBakcupTaskResponse) {}

IncrementalBackupTaskRequest {

tdcomm.TDCommReqHead head = 1;

uint64 region\_id = 1;

uint64 term = 2;

uint64 index = 3;

bytes start\_key = 4;

bytes end\_key = 5;

uint64 region\_version = 6;

IncrementalStorageBackend storage\_backend = 7;

}

IncrementalBackupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

uint64 taskid =2;

}

QueryIncrementalBakcupTaskRequest {

tdcomm.TDCommReqHead head = 1;

uint64 taskid = 2;

}

QueryIncrementalBakcupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

RegionIncrementalBakcupTaskInfo info = 2;

}

RegionIncrementalBakcupTaskInfo {

uint64 taskid = 1;

uint64 region\_id = 2;

uint64 term = 3;

uint64 index = 4;

IncrementalBakcupTaskStatus status = 5;

}

enum IncrementalBakcupTaskStatus {

Normal = 0;

Failed = 1;

}

CancelIncrementalBakcupTaskRequest {

tdcomm.TDCommReqHead head = 1;

uint64 taskid = 2;

}

CancelIncrementalBakcupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

}

message IncrementalStorageBackend {

oneof backend {

Local local = 1;

Hadoop hadoop = 2;

Cos cos = 3;

}

}

message QueryAllIncrementalBakcupTaskRequest {

tdcomm.TDCommReqHead head = 1;

}

message QueryAllIncrementalBakcupTaskResponse {

tdcomm.TDCommAnsHead head = 1;

repeated RegionIncrementalBakcupTaskInfo = 2;

}

#### 决策点（为什么这样设计）

1、容灾

TDStore Region一般是多副本的，允许某些节点故障。所以增量备份这里也需要有容灾机制。当某些节点故障了，备份任务仍然要被重新激活。

2、适应region分裂，迁移，合并

当region发生变化的时候，能感知到，并及时调整。

3、方便做负载均衡

由TDBackup做大脑，可以比较容易做到负载均衡。

### 数据恢复

#### 基本流程

0、DBA以recover模式构建集群，该模式下初始化集群时mc不会在tdstore上创建region.

1、DBA运行TDRecover工具，需指定恢复目标集群（mc地址即可），还需要指定恢复的截止时间戳 或者 截止时间点 ，以及恢复点的停止的方式（恢复点停止方式有四种，在3.3.2约束部分进行讨论）。

2、TDRecover根据指定的参数在备份网络文件系统中寻找合适的全量备份数据，并检验全量备份的数据是否完整（一般是region的数据范围是否全域，全量备份的结果是否为success）。

3、TDRecover发rpc给MC，获取所有注册的tdstore信息(使用rpc GetAllTDStores既可)，并检验tdstore容量是否足够（例如是否大于全量备份数据的总大小的120%）

4、TDRecover发rpc给MC，预创建region。（需要根据容量和备份region的数据范围，分配好合适的region，可以实现为原集群和新集群region一一对应的方式，这样复杂度比较好控制）

5、TDRecover发rpc给TDStore，从网络存储中加载指定region的全量备份数据（就是3.1中按region备份好的sst文件）。

6、启动多线程以全量备份的region列表为基准并行解析增量备份的region raft log，并将数据以tomas rule write的方式写入恢复集群对应的region中，运行过程中记录好最大的commit\_ts。

解析日志将redo log产生的数据缓存在内存中。

遇到commit log才可以发起rpc，向对应的tdstore写数据。

遇到abort log放弃对应事务的数据。

遇到split log ，将新的region，纳入解析目标（merge等操作等后续tdstore支持后在考虑如何处理）。

7、达到指定的时间点后以指定的停止方式停止相应region的回放。

8、TDRecover理解sqlengine的schema元信息，创建所有数据库对象的shcema version.

9、TDRecover理解sqlengine的表元数据，获得global schema version 、max index id以及每个index id对应的max auto increase id. (如果sqlengine将 index id放入tdstore中存储，max index id以及每个index id对应的max auto increase id可不进行恢复)

10、恢复保存的最大的commit\_ts 以及 步骤9中从数据中理解出来的global schema version 、max index id 、 max auto increase id到新MC。

11、结束恢复

#### 约束点

1、 TDRecover需要检验分布式事务的完整性，以防事务中某个region的数据丢失导致事务的部分数据未恢复。

2、增量数据回放的起点

全量数据保存是以timestamp为基准生成的sst文件，而增量数据的保存是以raft log为对象。在TDStore中timestamp与raft index不能保证偏序关系，即事务T1的commit ts 大于 事务T2的commit ts，并不能推导出事务T1的raft index 大于 事务T2的raft index。这是因为TDStore的实现中获取commit ts到写commit log这里是可以并发导致的。

这种特性导致了在raft log的日志的从后往前找backup timestamp合适的回放点，是没有基准的，因为即使在日志中遇到了比backup timestamp小的时间戳，也不意味着在日志的更前面没有比backup timestamp大的时间戳。

所以通过我们在备份全量数据时，获取备份backup timestamp前获取的全量region index来保证回放点的安全。整个推导过程如下所示，我们以绝对的物理时间为参考系来进行论证：

1、tdstore获取当前所有region的raft log index，记该时刻物理时间为T（local get index）。然后tdstore上报这些raft log index到mc，记该物理时间为T（report index）。根据发生的先后顺序可知，T（report index） > T（local get index）.

2、在mc收到raft log index，才能被相应rpc获取到。即T(mc get index) > T（report index）。

3、TDBackup先获取raft log index，然后再获取backup timestamp。即T（backup ts）> T(mc get index) 。

4、MC会保证所有获取的逻辑 timestamp与物理时间满足偏序，从这里我们可以推导出对于所有commit ts大于backup ts的事件，物理时间满足T（get commit ts） > T(backup ts).。

5、tdstore commit 流程是先获取commit ts然后才能写commit log。所以这里满足T（write commit log） > T(get commit ts)

综上可以得出T（write commit log） > T（local get index）。而raft log的写入是有序的，既raft index 大的，一定在物理时间上更晚。所以可以推导出备份get raft index小于或者等于backup ts后面提交的事务的raft index，即备份get raft index是安全的回放起始点。

当然这种情况下可能会存在回放部分事务的提交的commit ts < backup ts的情况，这里可以由回放时tomas rule write来保证幂等。

#### 增量数据回放的终点

**一：满足分布式事务原子约束 且 乐观恢复（各个region中第一个大于指定时间点的commit\_ts的事务会提交）**

增量数据回放的终点就不像起点那样有明确的raft index了，因为通过现实时间点指定回放的终点的方式本身就是不够精确的，所以这里我们只保证回放的数据满足分布式事务的要求，即不能出现某个分布式事务只回放部分数据的情况，满足约束1即可。具体方式如下：

1、TDRecover在回放过程中维护一种分布式事务表，项为(transid，各region上的子事务是否完结列表） ，并进行相应操作。

1）每回放一个新的分布式事务，就插入一项；

2）每在一个region中提交一个子事务（分布式事务在该region上投影）就找到分布式事务表中该事务对应的项，将子事务完结标记上；

3）一旦某项子事务全部标记成功，则从分布式事务表中移除该项。

2、当回放到region中第一个大于指定逻辑时间戳或者指定现实时间点的commit log，该回放事务提交，同时冻结当前region回放过程，保留所有内存中其它正在回放的事务。

3、等待所有region回放都进入步骤2的冻结状态。

4、检查分布式事务回放列表是否还存在项未完结，如存在则继续回放项中未完结的region，注意不是该项对应的transid，不准提交。（这里是担心出现交错上升的情况，导致出现必须回放完所有增量数量）

5、直到分布式事务回放列表为空，结束增量数据的回放流程。

**二：满足分布式事务原子约束 且 中性恢复（除非分布式事务中已有部分region提交，否则各个region中大于指定时间点的commit\_ts的事务一定不会提交）**

具体步骤如下：

1、TDRecover在回放过程中维护一种分布式事务表，项为(transid，各region上的子事务是否完结列表） ，并进行相应操作。

1）每回放一个新的分布式事务，就插入一项；

2）每在一个region中提交一个子事务（分布式事务在该region上投影）就找到分布式事务表中该事务对应的项，将子事务完结标记上；

3）一旦某项子事务全部标记成功，则从分布式事务表中移除该项。

2、当回放到region中第一个大于指定逻辑时间戳或者指定现实时间点的commit log，该回放事务不提交，且冻结当前region回放过程，保留所有内存中其它正在回放的事务。

3、等待所有region回放都进入步骤2的冻结状态。

4、检查分布式事务回放列表是否还存在项未完结，如存在则继续回放项中未完结的region，注意不是该项对应的transid，不准提交。（这里是担心出现交错上升的情况，导致出现必须回放完所有增量数量）

5、直到分布式事务回放列表为空，结束增量数据的回放流程。

**三：满足分布式事务原子约束 且 悲观恢复（各个region中大于指定时间点的commit\_ts的事务一定不会提交）**

具体步骤如下：

1、TDRecover在回放过程中维护一种分布式事务表，项为(transid，各region上的子事务是否完结列表） ，并进行相应操作。

1）每回放一个新的分布式事务，就插入一项；

2）每在一个region中提交一个子事务（分布式事务在该region上投影）就找到分布式事务表中该事务对应的项，将子事务标记成可提交；（注意实现时，该region需要继续回放后面的事务，但是如果存在与标记成可提交但是未提交的事务存在key冲突的，不予提交，因为存在先后关系）

3）一旦某项子事务全部标记可提交，则把该分布式事务统一提交，并从分布式事务表中移除该项。

2、当回放到region中第一个大于指定逻辑时间戳或者指定现实时间点的commit log，该回放事务不提交，且冻结当前region回放过程，保留所有内存中其它正在回放的事务。

3、等待所有region回放都进入步骤2的冻结状态。

4、检查分布式事务回放列表是否还存在项未完结，存在的话，打印出来，并删除掉。

5、直到分布式事务回放列表为空，结束增量数据的回放流程。

**四：冻结态恢复并手工pick**

具体步骤如下：

1、TDRecover在回放过程中维护一种分布式事务表，项为(transid，各region上的子事务是否完结列表） ，并进行相应操作。

1）每回放一个新的分布式事务，就插入一项；

2）每在一个region中提交一个子事务（分布式事务在该region上投影）就找到分布式事务表中该事务对应的项，将子事务标记成可提交；

3）一旦某项子事务全部标记可提交，则把该分布式事务统一提交，并从分布式事务表中移除该项。

2、当回放到region中第一个大于指定逻辑时间戳或者指定现实时间点的commit log，该回放事务不提交，且冻结当前region回放过程，保留所有内存中其它正在回放的事务。

3、等待所有region回放都进入步骤2的冻结状态。

4、分别打印region所有冻结事务以及分布式事务表，并等待DBA输入

5、提交DBA指定的分布式事务，并放弃未指定的事务。

#### mc 数据恢复相关新增RPC

mc 数据恢复相关rpc

rpc PreCreateRegion(PreCreateRegionRequest) returns (PreCreateRegionResponse) {}

rpc RecoverMetaInfo(RecoverMetaInfoRequest) returns (RecoverMetaInfoResponse) {}

PreCreateRegionRequest {

tdcomm.TDCommReqHead head = 1;

uint64 region\_id = 1;

uint64 region\_version = 2;

bytes start\_key = 3;

bytes end\_key = 4;

uint64 quorum = 5;

uint64 need\_capacity = 6;

}

PreCreateRegionResponse {

tdcomm.TDCommAnsHead head = 1;

}

RecoverMetaInfoRequest {

tdcomm.TDCommReqHead head = 1;

uint64 max\_commit\_ts = 2;

uint64 global\_schema\_version = 3;

uint64 max\_index\_id = 4;

repeated SingleAutoIncIDPair = 5;

}

RecoverMetaInfoResponse {

tdcomm.TDCommAnsHead head = 1;

}

message SingleAutoIncIDPair {

uint32 index\_id = 1;

uint32 max\_auto\_increase\_id = 2;

};

### TDStore适配

1、需要支持一个根据timestamp读取数据并将这些读取的数据转换成sst，写到指定路径的rpc。

2、需要支持一个从指定路径加载sst的rpc.

3、tdstore agent需要支持全量备份和增量备份的一些系列工作。

4、日志格式要微调在log header中加入实际时间点。

### MC适配

1、支持recover模式构建集群

2、支持外部接口指定tdstore列表的方式创建region

3、支持恢复保存的timestamp，以及所有index id和对应的auto\_inc\_id等元信息。