# 概述

## 简介

## 特性

### 多级SCALE-OUT扩展

实现各个系统的扁平化（水平扩展）：

1. 客户端接入层、中间件网关层、DB层三个层次上可横向线性扩展，满足性能及容量的何种处理需求；
2. 中间件网管层支持自动路由分配，根据业务需要可以灵活配置服务网关（proxy）的数量，实现处理能力线性扩展；
3. 网关层内置连接池实现链路共享，数据库连接数线性可扩展；
4. 数据自动切分分片，分片数据在DB根据策略自动重分布，存储容量随着DB节点数线性扩展。

### 强一致分布式事务

### 分布式优化器

内置大量的优化规则，对上百个场景进行优化，复杂SQL语句兼容性和处理性能好，同时支持prepare预处理、执行计划缓存、数据集透传等功能，保证数据一致性条件下实现高性能SQL处理。

支持的典型优化包括：

1. 分片剪枝
2. 合并下压
3. 并行执行
4. 条件下推及条件繁殖
5. 排序下推、limit下推等
6. 聚合函数优化

### 分布式批处理

提供分布式架构下批处理功能，满足金融、政企、运营商等行业日终大数据批处理需求，通过分布式FetchSize和存储过程对数据进行批处理，减少客户端与DB访问次数，批量返回数据集并进行批量处理。

### 读写分离

GoldenDB支持将DQL查询负载均衡到从库，提升系统资源利用和处理效率，读性能可线性扩展。

1. 对应用透明：应用无需改造，在数据库运维界面简单配置即可实现功能；
2. 可视化运维管理：支持在线调整权重，支持在同一个集群中不同业务使用不同的负载均衡模式；
3. SQL Hint提示：支持个别高实时性的SQL通过Hint指定至master执行。

注：

Hint是oracle提供的一种机制，用来告诉优化器按照告诉它的方式生成执行计划。

### 灵活数据切分技术

支持哈希、范围、列表、复制多种数据分片规则，可以根据业务数据特征，选择最合适的分片技术把数据分别存储在多个数据安全组中；通过合理的数据分片规则，发挥分布式数据库的最佳性能。

说明：所谓的切分就是把对应的表建立在多个节点后，下发的SQL语句需要分布在多个节点执行才可以，就需要将一个复杂的SQL拆分到多个节点上。

### 分布式数据备份恢复

支持快照恢复到任意时刻和恢复到快照生成时刻，且可以保证集群数据的强一致性。

### 在线数据重分布

支持数据节点的扩容、缩容，能够高效地将数据均匀分布到数据库集群上，同时保证对在线业务影响小、且可操作性强。

### 分布式数据导入导出

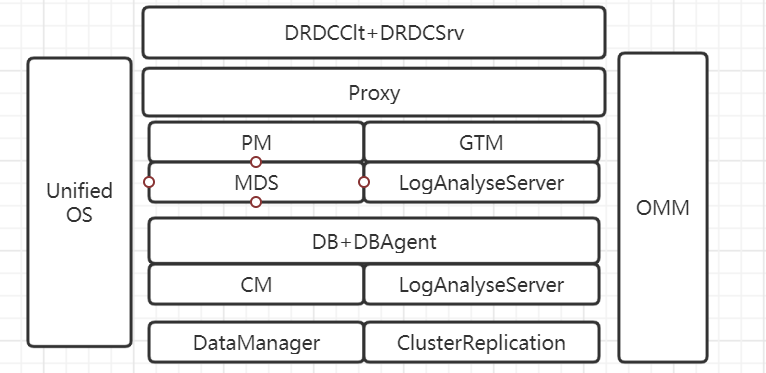
数据导入导出与交易分离，降低对联机交易的影响。

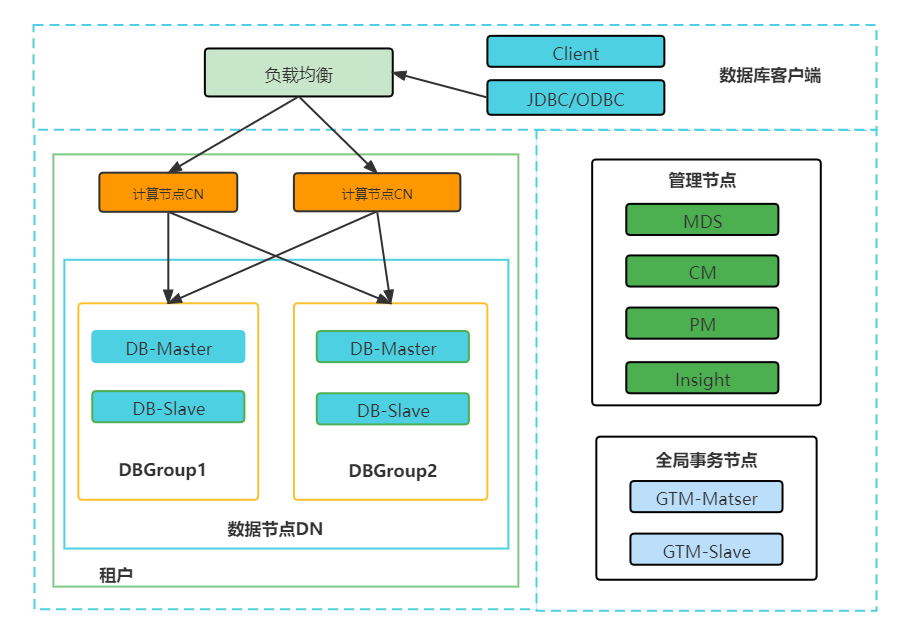
## 产品性能

GoldenDB通过分布式执行缓存、并发事务控制、数据复制优化等手段，对整体性能进行提升。在金融行业客户实测数据中，银行转账核心业务系统TPS能够达到4万（2017年），银行信用卡核心交易TPS10万（2020年），满足银行业务高并发低延时的特性。

# 架构

## 逻辑架构





### 数据库驱动

数据库驱动以集成的方式嵌入到具体应用中，和应用一起部署。GoldenDB支持通用数据库标准协议，并能够提供JDBC、ODBC、Java、Python、C、C++等常见开发语言使用的接口驱动。

从功能角度来看，数据库驱动实现了计算节点的透明接入、负载均衡和故障透明转移（LVS）。

分布式数据库的计算节点负责处理应用的SQL请求，计算节点集群由多个计算节点组成，用户可以规划计算节点和应用之间的对应关系（可以理解为一种负载均衡）。数据库驱动根据配置的规则将应用请求发送给合适的计算节点，并确保负载均衡地分配到这些计算节点上。当这些计算节点发生故障后，驱动层能够实施透明的故障转移，将应用的新请求发送给正常的计算节点，并在故障节点恢复后，能够将应用的请求重新路由到该节点上。

说明：

DRDCCtl：jar包，以数据服务的方式向应用提供数据库功能，并对应用请求或响应进行编解码。

DRDCSrv：将数据服务转换为一系列SQL请求，并发送到合适的目标端口上。

### 计算节点集群(Proxy)

计算节点集群是分布式数据库的核心层，由无状态的计算节点（Proxy）组成。计算节点从驱动层（业务操作）或者管理节点（内部管理命令）接收用户的操作（一般以结构化查询语言描述，即SQL），进行逻辑优化和物理优化，生成满足分布式事务一致性的分布式查询计划。计算节点在执行分布式查询计划时，通过持续地访问数据节点，完成用户的最终操作请求。用户可以根据应用对可靠性、可用性、性能等因素的不同要求，对计算节点进行合理的规划。

### SQL引擎

### 数据节点集群

数据节点集群是应用数据的最终存储组件。所有的数据节点组成一个或多个数据库集群，用户操作的事务尽量不要跨越多个数据库集群（目前已经支持跨集群，但是还是不够完善）。

数据库集群由一个或多个安全组（DBGroup）组成，集群中每个表中的数据按照某种策略进行横向分片后存放到对应的安全组中，分片策略包括复制策略、哈希策略、范围策略和列表策略。

数据按照上述策略（复制策略除外）分片后，每个安全组上的实际数据在理论上只有总数据量的1/N（数据分布的均匀程度依赖于切分策略和真实数据分布的匹配程度）。随着安全组数量的增加，每个安全组承载的数据量和读写负载会相应的减少，从而在数据节点集群内部具备了读写能力的水平拓展。

安全组是由一个或者多个数据节点组成的数据库节点组，组内的数据库节点拥有相同的数据。当安全组存在多个数据节点时，其中一个数据节点为主用节点其他数据节点都是备用节点，数据在主备节点之间实时复制。主用节点具备读写能力，备用节点可以提供度能力。

### 管理节点

管理节点在分布式数据库中负责集群管理流程，不涉及业务的访问流程，无负载压力，一般采用两节点主备方式部署。管理节点按照功能分工，可以有如下几个模块：

#### 统一运维管理(OMM)

OMM是GoldenDB分布式数据库的统一操作维护入口，可以进行用户和权限管理、元数据管理、计算节点管理、数据节点管理、DDL执行、节点扩容、备份恢复、统计及告警管理等。

#### 元数据管理器(MetaDtaServer)

元数据指的是数据的元信息，如库、表、视图、存储过程、触发器、自定义函数等数据模型的定义，元数据管理器存放系统的全量元数据，是整个分布式数据库集群的元数据中心。

此外，GoldenDB元数据管理器还保存了整个集群的拓扑信息，因此是更广义的元数据管理。

为了提高启动和运行的效率，除了元数据管理器存有元数据定义外，计算节点和数据节点也会存放元数据定义，但是计算节点和数据节点只存放本节点所涉及应用的元数据定义，即当计算节点中的元数据和管理节点不一致时，会同步管理节点的元数据到本地。

#### 计算节点管理(ProxyManager)

负责管理计算节点集群。管理工作一般分为两类：

一类为集群的组建管理，包括计算节点的创建、启用、禁用和删除；

另一类为集群的应用管理，包括定义计算节点和应用的对应关系、计算节点异常后的数据恢复调度。

#### 数据节点管理(ClusterManager)

数据节点管理也分为两类：

一类为集群的组建管理，包括数据节点、安全组、数据节点集群的创建、变更和删除；

另一类为集群的任务管理，包括数据节点异常、恢复后的调度管理、数据节点备份恢复的调度、数据重分布等功能的任务调度管理。

#### 全局事务管理节点(GlobalTransactionManager)

全局事务管理器在分布式数据库中维护全局事务的全生命周期，提供申请、释放、查询全局事务的能力，并采用双活方式部署。

## 物理架构

数据库驱动：可以直接使用标准数据库驱动接入GoldenDB集群，如JDBC，可以支持按照最快响应时间和均匀随机方式将SQL请求下发至多个计算节点，当前也可以使用常用的负载均衡设备LVS、F5等接入GoldenDB集群。

**计算节点Proxy：**计算节点理论上可以单机部署，但是通常要求部署2个以上，以便当某些计算节点发生故障后，驱动层能够实施透明的故障转移，将应用的新请求发送给正确的计算节点。

**数据库集群：**数据节点集群可以部署一个或多个，当业务之间的数据要求物理隔离时，可以将不同的业务数据存放在不同的数据节点集群；也可以多个业务的数据共享一个数据节点集群。每个集群内部的安全组数目可以根据业务量、访问性能要求、硬件条件确定；理论上每个安全组内部的副本数目越多，可靠性就越高，但是成本也越高。

**管理节点：**管理节点的各个模块可以单机部署也可以使用HA（现在已经使用ZK）软件进行双击冷备部署。

**全局事务控制节点：**分布式事务涉及节点间数据提交的一致性，该数据的一致性主要依赖于全局事务的状态，所以全局事务节点必须是高可靠的，全局事务控制节点为双活模式，在发生异常时可以进行服务快速接管。

计算节点集群具备横向扩展能力，数据节点集群具备横向和纵向扩展能力。在高性能要求下，分布式数据库集群会涉及大量的计算节点和数据节点。物理部署计算节点和数据节点时可以非常灵活，既可以部署在物理机器上，也可以部署在虚拟化平台上；同时每个节点既可以部署在一个设备上，也可以将多种类型的节点部署在同一设备上。不管采用哪种方式，一般都需要遵守如下原则：

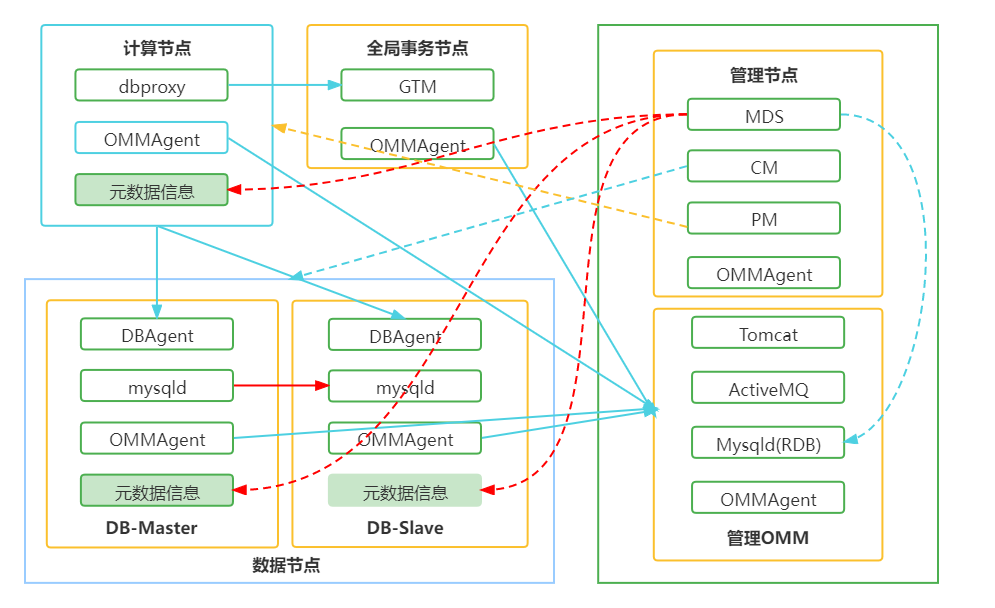
1. 计算节点需要分配更高的计算资源，数据节点需要分配更高的内存和存储资源；
2. 归属于同一个应用的多个计算节点避免部署在同一台物理机器上，更进一步避免部署在同一机框或者数据中心；
3. 归属于同一个安全组的多个数据节点避免部署在同一台物理机器上，更进一步避免部署在同一机框或数据中心。

## 组件和进程关系

|  |  |  |  |
| --- | --- | --- | --- |
| 节点类型 | 模块名 | 进程名 | 说明 |
| insight | InsightServer | Insight | 运维管理平台服务器 |
| 大数据组件 | Zookeeper | 管理节点高可用管理 |
| Kafka | 运维数据统一传输管道 |
| Elasticsearch | 性能数据存储单元 |
| Logstash | 性能数据采集节点 |
| Redis | 消息缓存 |
| 主机代理 | InsightAgent | Insight代理，每台服务器部署 |
| Filebeat | 日志采集，每个服务器部署 |
| 计算节点 | CN | dbproxy | 计算节点进程 |
| OMMAgent | OMM组件代理 |
| 数据节点 | DN | mysqld | 数据节点mysql进程 |
| dbagent | 数据节点代理 |
| OMMAgent | OMM组件代理 |
| 全局事务节点 | GTM | GTM | 全局事务 |
| OMMAgent | OMM组件代理 |
| 管理节点 | OMM | Tomcat | OMM管理界面Web服务 |
| ActiveMQ | 内部消息队列 |
| mysqld | RDB数据库 |
| OMMAgent | OMM组件代理 |
| Manager | MetadataServer | 元数据管理 |
| ProxyManager | 计算节点管理 |
| ClusterManager | 数据节点管理 |
| OMMAgent | OMM组件代理 |
| 数据导入导出 | LoadServer | loadserver | 数据导入导出工具 |
| OMMAgent | OMM组件代理 |

## 组件交互

各个功能组件之间的访问关系如下图所示：



1、计算节点和数据节点之间通过dbagent建立**长连接**，所有的副本都需要建立

2、数据节点主节点和从节点之间通过mysql的binlog同步复制原理，实现数据的同步

3、事务访问计算节点，如果需要申请全局事务ID，会通过GTM管理节点申请GTID

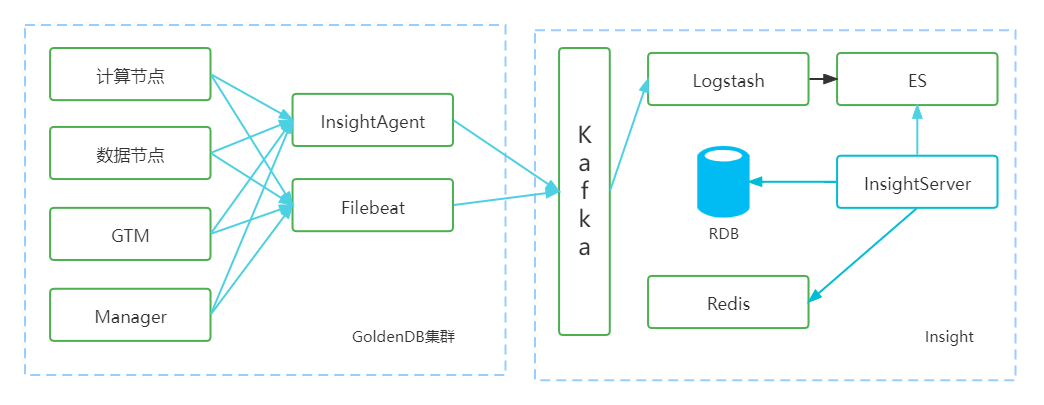
4、管理节点中ClusterManager会统一管理数据节点，比如数据节点的状态、扩容缩容，并协同计算节点控制数据的访问

5、管理节点中的ProxyManager实现对计算节点的统一管理，会和每个计算节点进行连接

6、管理节点中的Metadataserver管理元数据信息，有DDL相关的变更会在这里同步更新，元数据会保存在RDB中。同时为了优化执行效率，元数据信息也会同时同步到每个计算节点和数据节点的内存中，业务访问的时候优先从本地读取元数据信息。

7、每台服务器会部署ommagent，用于执行OMM管理节点下发的命令，并将告警信息同步到OMM管理节点

Insight运维管理平台访问关系：



1、InsightAgent是主机代理，每台主机上部署，执行insightserver下发的命令，并将数据收集推送到kafka

2、Filebeat是日志采集代理，用于收集每台服务器的日志数据

3、运维性能数据经过kafka消息队列后通过logstash采集到elasticsearch中存储

4、Insightserver会查询ES中的性能数据、RDB中的集群信息以及Redis中的缓存信息进行展示和汇总分析

## 待改进

### 架构

1. 基于分库分表中间件的架构设计思路，相对于NewSQL效率相对低（分片不够细化）
2. OLTP比较完善，但是OLAP还是不够完善
3. Oracle兼容性不够，兼容性需要proxy和DB分别实现，存在重复
4. 网元过多，交互比较麻烦，且高可用复杂
5. 无法避免热点数据，只能限流
6. 高可用采用每个网元上的agent判断，如果agent异常，则认为组件异常，不合理

### 组件

#### Proxy

#### SQL

1. 在线DDL不够完善
2. 分发规则不够细化，很多可以合并的没有计算出来
3. 子查询优化不足
4. 如果是悲观锁，锁失效的时候行锁升级表锁
5. 对于proxy需要SQL拆分的情况，如果涉及now()等时间函数，则会有时差
6. 对于聚合函数，无法保证严格的一致性
7. 分布式存储过程、视图支持不够完善
8. 分布式系统表查询

#### GTM

1、多线程实现并发，但是对于回滚代价比较高

#### DB

1. Oracle兼容性不够
2. DB主从复制有效率提升，但是相对于目前NewSQL还是不够高效

#### LDS

1. 导入导出走的是后端DB的链路，如果表中存在自增列，则在导入数据后，需要重新连接proxy刷新一下自增列的最大值

说明：后端导入数据的时候，RDB里面并不会感知全局的自增列大小，所以需要后端导入成功后，在自增列最大值基础上增加一个偏移量。

1. 导入导出需要在DB节点操作，依赖于FTP服务，并且无法从前端发起服务，这个在银行金融领域权限管控比较严格的场景，不一定能够开放这么高权限

注：后续采用goldendumper，但是还是不够完善，这个需要专门设置一个数转的proxy，避免对正常业务干扰。

3、MySQL关键字需要自己手动处理，无法智能识别处理（KEY->`KEY`）

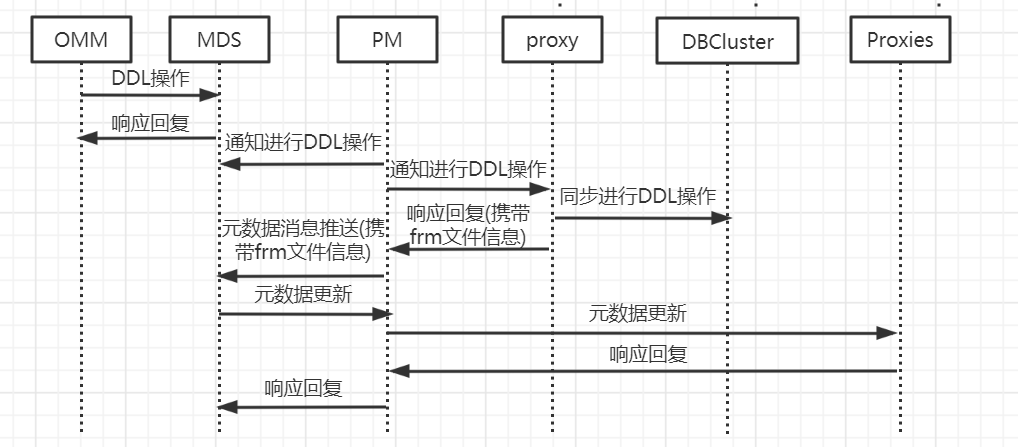
# 原理

## 协作流程

### 元数据管理流程

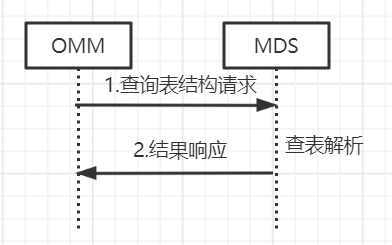
#### DDL流程

具体步骤：



1. 用户在web界面查看cluster信息，如果集群状态正常则可实施DDL操作，选择连接实例执行DDL操作；
2. DDL操作被推送至MDS，对于文本类的DDL操作集合，OMM全部读取后，可以分包推送；
3. MDS接收全部DDL操作集合，将DDL操作入操作状态表，标记状态位置位；做简单的逻辑判断后回复用户响应（譬如判断集群是否正常）；同时将消息推送至PM；
4. PM收消息包后，向指定proxy推送DDL；
5. Proxy做语义分析和语法检查，将消息推送至DB上实施落地。对于DDL执行时间较长的情况，需要proxy定时返回中间响应给PM，告知DDL正在执行中，PM将中间响应返回给MDS，MDS更新操作记录表中最近一次响应消息的时间。PM和MDS上都设有定时器，收到中间响应会更新定时器；
6. DDL执行成功后，proxy发送响应给PM，携带DDL执行结果的元数据信息；
7. PM透传proxy的响应消息包给MDS；
8. MDS收到消息后，解析消息，更新数据库中记录；
9. DDL实施完全成功后MDS更新数据库中记录的标志位，同时通知PM去向相关Proxies推送元数据信息，如果推送某个proxy元数据失败，pPM告警；
10. 最终的操作结果由用户在web界面上查询获取。

#### 查询表结构

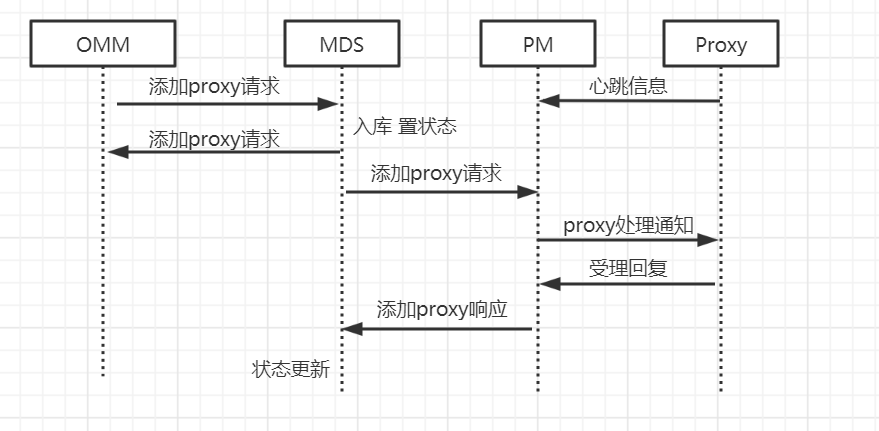


1. 界面展示集群表列表，用户选择指定表查询结构信息，OMM将消息下发至MDS；
2. MDS收到消息后，查表解析.frm文件；
3. MDS更新操作记录，返回结果给OMM。

### Proxy管理流程

#### 新增proxy

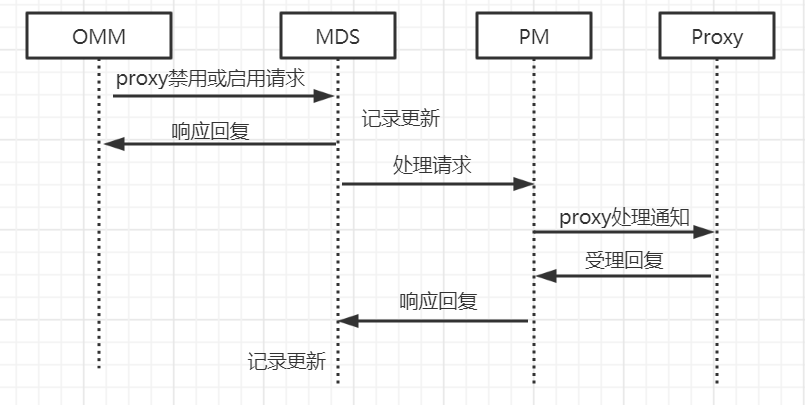
用户在OMM界面进行过添加设备操作后，用户便可以在proxy展示界面看到已经准备好的未分配使用的proxy节点，并可以选择将其加入proxy集群中，步骤如下：



1. proxy启动后向PM发心跳信息，PM不对其做任何处理；
2. 用户在OMM选择将未分配的proxy加入proxy集群；
3. OMM发消息通知MDS添加该proxy到集群中；
4. MDS将操作入库，标记状态位；同时将响应消息回送至OMM；
5. MDS将消息传送至PM；
6. PM查看用户指定IP的proxy是否心跳正常，同时通知proxy做一系列初始化工作；
7. Proxy初步判断准备就绪后，PM回复MDS添加proxy响应消息；
8. MDS同时入库更新本条记录；
9. 用户通过后续操作查看操作的实施结果。

#### Proxy启用和禁用

启用和禁用即proxy的上线和下线，如proxy升级的场景，在进行禁用当前客户端应该不再向本proxy发送新事务，当前正在执行的新事务执行完毕后，禁用操作完成。



具体步骤：

1. 用户在OMM发起将proxy禁用或启用的请求；
2. MDS读取该条proxy及所有连接实例的记录，检查当前是否有操作在进行，如果有则拒绝本地操作；佛足额允许发起操作并更新所有标志位；
3. MDS回复OMM响应消息；
4. PM将请求透传至proxy；
5. Proxy实施操作处理：
6. 禁用操作：禁用所有绑定的连接实例，即proxy不再接受新消息，当前在线事务需要处理结束；
7. 启用操作：核实连接池的状态等。
8. Proxy回送响应；
9. MDS根据操作实施记录：更新落地。

#### Proxy删除

删除proxy是指将该proxy在逻辑关系上从系统中移除。只有处于禁用状态的proxy才可以被删除。

具体步骤：

1. 用户在界面将某proxy删除；
2. OMM将消息发送至MDS；
3. MDS读取该条proxy及所有连接实例的记录，检查当前是否有操作在进行，如果有则拒绝本次操作；检查该proxy是否尚未启用或已经处于禁用状态；满足所有条件后允许发起操作并更新所有标志位；
4. 通知proxy操作实施；
5. MDS根据操作实施记录落地：更新或删除。

#### Proxy状态查询

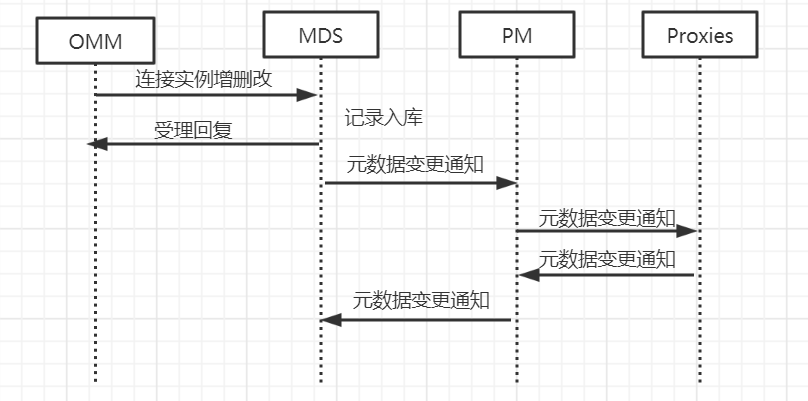
具体步骤：

1. 每个proxy都实时上报统计和状态信息给PM；
2. PM再上报至MDS。根据不同的信息类别，分别由MDS入库，或者进一步上报OMM入库；
3. 用户在发起当前状态和统计信息查询时，OMMServer读取数据并展示。

### 连接实例管理流程

#### 新增/修改/删除连接实例

用户可以在界面为一个集群新增一个连接实例，即访问某数据库集群的对外端口号和访问用户名、密码信息。供后续的proxy和cluster的关联使用。同时用户也可以在界面修改或删除连接实例，前提是该连接实例未被任何proxy绑定。

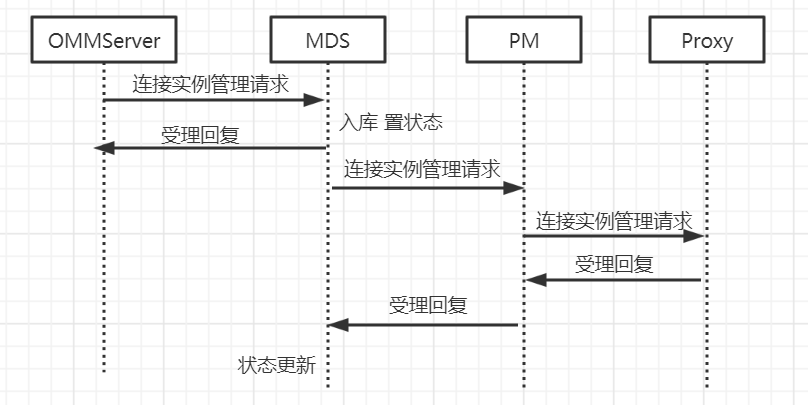


具体步骤：

1. 用户在界面为集群增加、修改或删除一个连接实例；
2. MDS对其校验后入库持久化。

#### 绑定连接实例

连接实例的绑定、启用、变更、禁用、解绑的流程图如下所示：



具体步骤：

1. 用户在界面选择绑定proxy的连接实例。用户可以按照集群、连接实例层级展示出端口、用户名、密码、连接数等信息，用户可以指定连接实例的连接池数目以及前端连接数；
2. MDS在数据库里添加一条记录，对其标志位赋值；
3. MDS回复用户OMM响应消息；
4. MDS依次将消息推送至proxy实施；
5. Proxy受理后发送回复消息，依次回送至MDS；
6. MDS根据处理结果处理该条记录：更新落地；
7. 用户通过后续查询操作查看操作的实施结果。

注：连接实例的绑定时，proxy只是建立绑定关系，但不进行实际连接池的创建。

#### 启用连接实例

具体步骤：

1. 用户在界面选择启用响应proxy绑定的某连接实例；
2. MDS在数据库里添加一条记录，对其标志位赋值；
3. MDS回复用户OMM响应消息；
4. MDS依次将消息推送至proxy实施；
5. Proxy受理后发送回复消息，依次回送至MDS；
6. MDS根据处理结果处理该条记录：更新落地；
7. 用户通过后续查询操作查看操作的实施结果。

注：用户选择启用连接实例时，proxy进行连接池的创建。

#### 变更连接实例的连接数

具体步骤：

1. 用户在界面修改proxy的连接实例的连接数目，支持增加和减少。在减少的场景下，用户可以指定proxy处理方式：优雅减少，或强制减少。
2. MDS在数据库里查询出原纪记录，检查其当前状态标志位，如果正在执行更新动作则不处理当前用户请求；如果数据未发生变化则不处理当前用户请求；更新本记录的状态；
3. MDS回复用户OMM响应消息；
4. MDS依次将消息推送至proxy实施；
5. Proxy受理后发送回复消息，依次回送至MDS。对于实例数目减少的情况，proxy的路由模块应该控制不再有新发起的事务推送至该链路，在该链路上已有的事务处理结束后销毁该连接。如果用户指定为优雅方式，proxy等待链路上的消息处理结束后回复响应，同时可配置完成时间，如果在该时间内仍未处理完，则强制销毁连接；指定强制关闭时，不需要等待事务执行完毕就可以关闭；
6. MDS在数据库里做记录更新落地；
7. 用户通过后续查询操作查看操作的实施结果。

#### 禁用连接实例

在发起禁用操作时，客户端不再向该proxy的端口发起新事务。

具体步骤：

1. 用户在界面操作要将某proxy上的某监听实例禁用；
2. OMM将消息下发至MDS，最后落到proxy实施；
3. MDS检查该proxy及连接实例当前是否有操作在进行，如有则拒绝本次操作；否则更新标志位；
4. MDS回复用户OMM响应消息；
5. Proxy拒绝处理来自该端口的新消息，给发送回复消息，依次回送至MDS；
6. MDS根据返回结果做字段更新落地；
7. 用户通过后续查询操作查看操作的实施结果。

#### 解绑连接实例

在删除一个连接实例时应该将其禁用，通过状态查看保证其上面没有激活的全局事务后再将其彻底删除。

具体步骤：

1. 该实例已经被禁用；
2. 用户在界面将其proxy的连接实例删除；
3. OMM将消息下发至MDS；
4. MDS检查proxy及连接实例当前是否有操作在进行，如有则拒绝本次操作；检查此实例当前统计数据的激活事务数目为0则允许发起删除操作，更新记录标志位；
5. 通知proxy操作实施；
6. MDS根据操作实施记过进行该记录的处理：更新或删除；
7. 用户通过后续查询操作查看操作的实施结果；
8. 通过后续查询操作查看操作的实施结果。

### Cluster管理流程

#### 集群新增

用户可以通过界面新增一个集群，系统内部做集群名字和集群ID的映射关系。

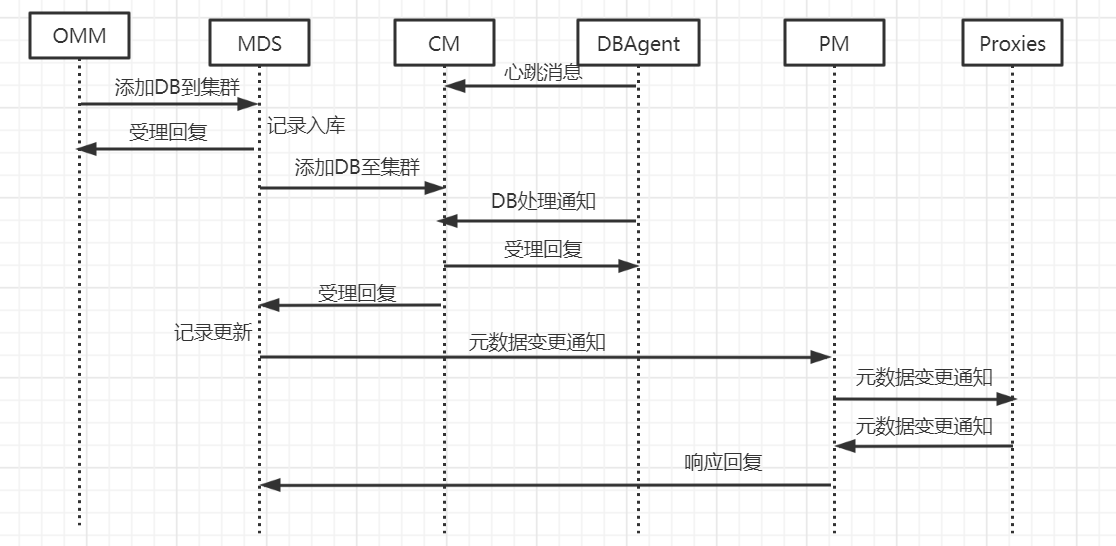


具体步骤：

1. 用户新增一个集群，填写集群名称；
2. OMM下发消息至MDS；
3. MDS分配一个内部唯一ID；
4. MDS回复用户响应；
5. MDS将信息推送至CM。

#### 集群组建

集群组建的过程即为将DB添加到集群的过程：

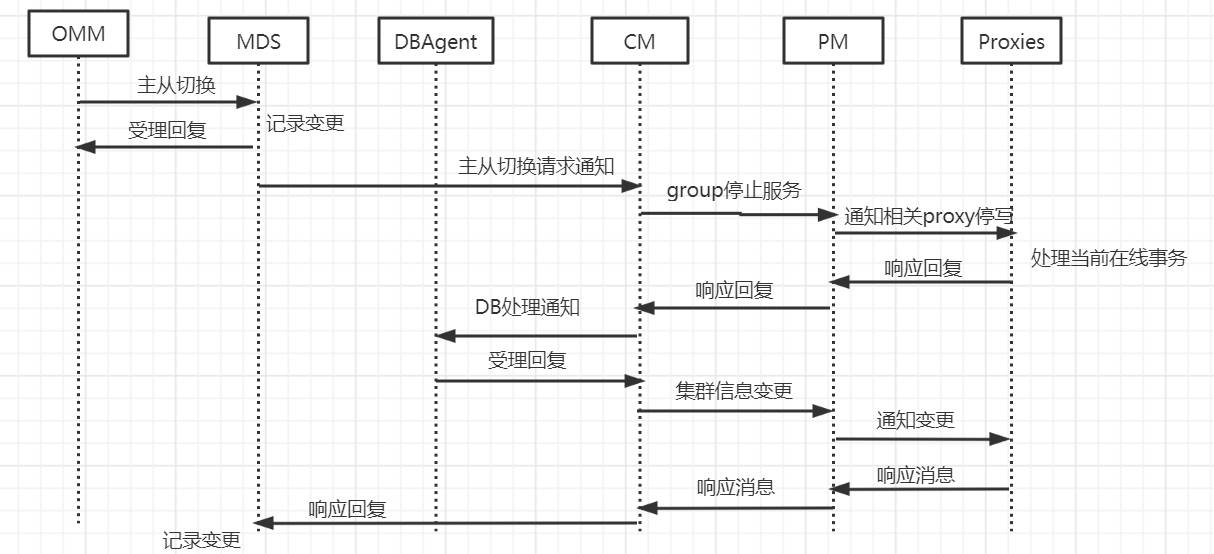


具体步骤为：

1. 安装配置完各个DB节点后，DBAgent自动启动，并向CM发心跳消息；
2. 用户通过OMM进行集群组建，选择处于未分配状态的DB进行cluster组建，这些DB信息是在进行规划组网时录入OMMServer库的。支持group至DB的逐级引导添加，完成后发送消息至MDS；
3. MDS需控制在一次添加操作未结束时不处理新发起的请求；将操作所涉及的所有记录都存入相关表并做状态标记；
4. MDS回复用户中间响应；
5. MDS将消息下发至CM；
6. CM接收到请求后，从待分配列表中找到该DB信息，然后进行必要检查并协调各个DB进行对应的变更；
7. 各DB上的DBAgent接收配置变更及组网规划数据后，进行相应的检查或变更，如协调slave和master之间建立复制通道；处理结束后将结果返回给CM；
8. CM将集群组建结果恢复给MDS；
9. 集群组建成功后MDS将信息入库落地更新信息；
10. MDS通过PM将集群信息推送至各个proxy（这样后续就能获取最新的集群信息，但是这个不是从MDS获取的吗？我们这里应该不需要推送？）
11. 用户可以在界面查询集群组建情况及DB的状态。

#### Group内部主从切换

用户主动发起的主从切换流程如图所示：

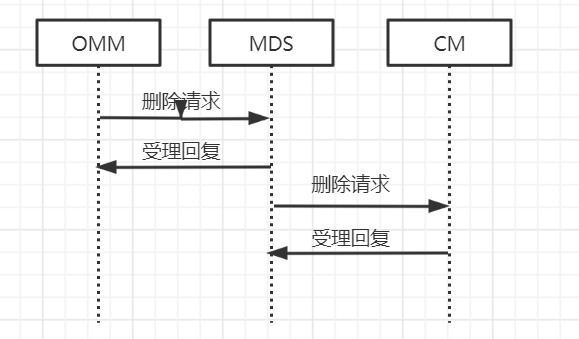


具体步骤：

1. 用户在界面查看集群内DB状态正常，则可对某group发起主从切换；
2. MDS收到请求消息，从库查出该group记录，包含但不局限于以下动作：
3. 判断当前状态标志位，如果有其他的操作在进行，则拒绝本次操作；
4. 查看新主的状态，如果为异常，则拒绝此次用户操作；
5. 更新数据库的记录，置状态。
6. MDS回复用户中间响应；
7. MDS向CM发主从切换请求通知；
8. CM做进一步校验，并通知PM至相关proxy停止处理该cluster的新事务；
9. Proxy停止发起涉及该cluster的新事务，处理完所有涉及该cluster的当前在线事务后回复CM响应消息；
10. CM通知相关的DBAgent实施主从切换操作，结束后回送MDS响应消息；
11. MDS根据返回结果将消息持久化入库。

#### Cluster删除

删除只是解除该资源与系统的逻辑关系，不改变DB的运行状态。删除上层资源时，归属其的下层资源同样被处理。



具体步骤：

1. 业务不再发起新事务；
2. 对各proxy至该cluster的连接实例去激活；
3. 删除各proxy至该cluster的连接实例；
4. 用户在界面操作将该cluster删除，接触逻辑关系；
5. 元数据中心做保护，如果当前有proxy和该cluster有绑定关系，则拒绝该操作。

#### Cluster启用和禁用

专门的web界面实现集群启停，可以通过如下操作步骤同样达到目的：

1. 用户确保不再有新请求发送过来，则当前cluster相当于下线；
2. 进一步提高高可靠性，用户可以在界面将该cluster涉及的连接端口实例在proxy层面上全部下线；
3. 可以在proxy层面查看cluster所涉及连接端口实例的在线会话统计信息，确保整个集群无在线会话。



具体步骤如下：

1. 用户在界面对cluster发起启用、禁用请求；
2. MDS收到请求后，检查相关连接实例是否有操作进行，有则拒绝本次操作，没有则更新数据库记录，标记状态；
3. MDS向PM推送相关proxy连接实例进行启用、禁用请求；
4. PM通知相关proxy启用、禁用，proxy进行相关处理后，回复响应消息给PM；
5. MDS收到PM的响应回复后，更新数据库状态。

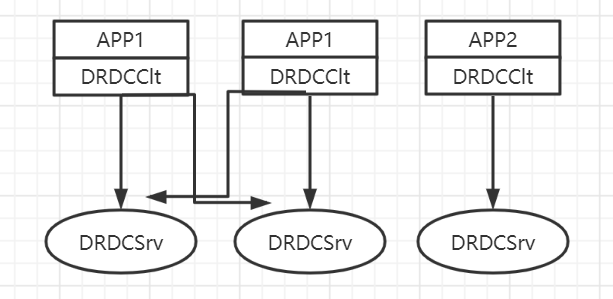
#### 集群信息查询

用户通过OMM可以实时查询集群的信息，包括集群状态及集群内部各DB信息等。

1. 每个DB都实时上报状态信息给CM；
2. CM再上报至MDS。根据不同的信息类别，分别由MDS入库，或进一步上报OMM入库；
3. 用户在发起当前状态等信息查询时，OMMServer读取数据并展示。

## 数据库客户端DRDC

DRDC和应用部署在一起，分为DRDCClt和DRDCSrv。二者的视图关系如下：



在某个DRDCSrv发生异常时，应不影响APP访问。

### DRDCClt

DRDCClt为jar包，以数据服务的方式向应用提供数据库功能，并对应用请求或响应进行编解码。

1. API均采用同步调用；
2. 使用统一的API类型，DRDCClt做成通用版本；
3. 查询的结果集暂时按照结+结果集的形式；
4. DDL语句均为web界面下发，只有临时表可以在运行态动态从客户端下发；
5. DRDCClt和DRDCSrv之间采用长链接的方式通讯，各连接轮询发送，如果本地DRDCSrv异常再通过其他DRDCSrv发出；
6. 客户端同时支持java和c版本。

### DRDCSrv

DRDCSrv将数据服务转换成一系列SQL请求，并发送到合适的目标端口上。

设计考虑：

1. DRDCSrv在转换SQL语句时考虑配置文件的方式。配置文件里有API名字的段，下面配置该事务执行的SQL语句；DRDCSrc支持启动时加载该配置和动态加载。API调用时将参数传入，DRDCSrv结合配置sql得到最终的执行sql语句。
2. DRDCSrv和proxy之间采用长链接方式进行直连，proxy通过端口号区分要访问的集群及访问使用的用户名和密码。
3. 在一个API内只有一个分布式事务。

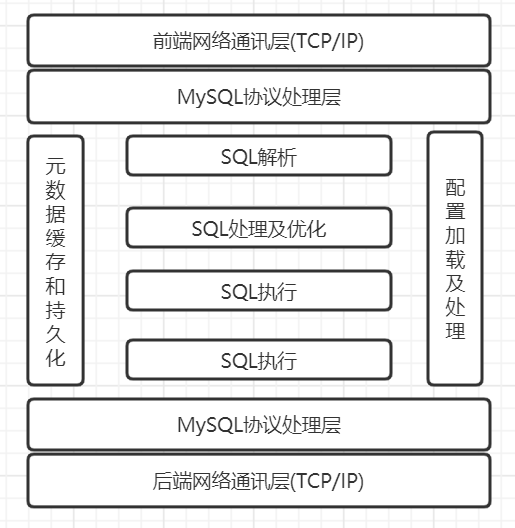
## Proxy

Proxy是无状态的代理节点，负责SQL优化、SQL路由、数据节点的负载均衡、分布式事务的调度等；Proxy和上层的数据库客户端以及下层的数据局集群都是采用兼容MySQL的通讯协议。

缺点：

1. 不支持跨集群操作
2. SQL支持Oracle需要开发两套代码（SQL和DB），存在重复
3. 很多算法需要优化（尤其是JOIN）
4. 构造语法树不是非常合理，尤其复杂语句

### 逻辑架构



### 链路管理

Proxy和客户端的连接与proxy和DB的连接不是一一对应的。

#### 前端链路

1. 当proxy感知proxy与非客户端模块链路异常，不会主动断开与客户端的连接
2. 当proxy接收到连接实例禁用请求时，会主动断开同客户端以及DB（是有意的主备DB）的连接
3. 当proxy连续7小时没有接收到客户端的新请求时，会主动断开与客户端的当前链路（时间可配置）

#### 后端链路

1. proxy启用时，proxy初始化会与所有DB主备机建立一条管理连接
2. 当客户端连接池与proxy初始化建立（例如10条长链接），且没有下发业务SQL时，proxy会与g1的主DB节点新增10条连接，proxy与其他DB还是一个管理连接

在没有业务SQL下去的情况下，5分钟后proxy与g1的主DB节点之间保留OMM页面上配置的“DB最大空闲连接数”个连接，因此，“DB最大空闲连接数”需要配置的小于客户端的连接池数量，而业务的并发度通常小于等于客户端的连接池数量

1. 当有业务SQL下发时，proxy与被涉及的DBGroup主机DB节点之间会新增连接，当新增的连接不够用时，会继续新增连接，正常情况下新增连接的综述不会超过客户端的并发数

在没有业务SQL下发的情况下，5分钟后proxy与所有连接数超过OMM页面配置的“DB最大空闲连接数”的DB节点之间，保留“DB最大空闲连接数”个连接，多余的空闲连接释放掉

1. 在没有读写分离模式的情况下，proxy与备机DB节点之间始终都是一条管理连接
2. 加入集群中有2个DBGroup，且为一主一备
3. 当g1节点的一个备机DB异常，客户端与proxy新增连接成功，执行业务成功
4. 当g1节点的另外一个备机DB也异常，客户端与proxy新增连接成功，执行写SQL的commit时会卡主10秒（只有当半同步转异步的时候才会卡，后面不会再卡，10秒超时后，半同步转为异步，10秒值可以配置），然后SQL提交成功
5. 当g1节点上的最后一个主机DB也异常，客户端与proxy新增连接成功，执行涉及g1节点的SSQL成功

说明：

1. proxy与DB之间的管理连接是连接实例级别的，每个连接实例一个管理连接（不是每个proxy一个）
2. 管理链路不会被业务复用

### 前后端通信协议及MySQL协议层

1. proxy对前端采用兼容mysql的协议，对后端采用mysql协议，需支持mysql的协议处理；
2. Proxy对前端采用TCP长链接，客户端一次性将语句下达至proxy；
3. Proxy对后端采用连接池的方式处理，连接池的数量及用户密码可配置，和端口是一一对应的关系；
4. Proxy根据应用访问的端口号来选择对应的集群以及具体的连接；
5. Proxy连接池耗尽后是否可以动态申请可以配置；
6. Proxy对上做局部流控，如果当前连接数超过配置的最大连接数，要等某连接上的在线事务全部处理结束后再关闭该连接。

### 元数据缓存及持久化

为减少proxy频繁访问元数据给MDS所造成的压力，proxy设置有元数据缓存模块，全量缓存本proxy所涉及的Cluster对应的元数据，具体包括但不局限于：

1. 库表定义以及数据分布信息，存储过程及编译后的语法树；
2. 集群信息；
3. 业务对应的库表统计信息。

元数据缓存模块是能够高效的对元数据更新和查询：

1. 接受推送的元数据变更；
2. 接受proxy内部模块元数据查询。

为了增强系统的健壮性，proxy具备元数据持久化功能，元数据更新时能够实时将元数据信息落地，以便在MDS发生异常时，proxy重启后能使用旧的元数据继续正常对外服务。

### SQL解析

SQL解析模块主要是将文本格式的SQL语句解析成诶不语法树，该步骤只做语法检查。

### SQL处理及优化

SQL优化模块主要是对普通的语法树结合元数据及统计信息进一步分析，生成满足分布式事务场景的分布式语法树。该语法树包括全量信息，SQL执行时不再访问元数据。

考虑的优化包括但不局限于如下实现：

1. 尽可能将条件信息推送到叶子节点；
2. 做条件优化；
3. 子查询优化，将能够满足优化条件的子查询转换为JOIN；
4. Order by、group by优化；
5. Limit优化；
6. 函数优化；
7. 分布式动作；
8. 支持explain查看执行计划。

### SQL执行

根据语法树各个节点的指令动作执行对应的动作，并完成执行结果的分析和处理。

1. 外部交互类：如何GTM的交互，和SQL路由模块的交互；
2. 内部处理类，如排序、分组、join等。

### SQL路由

SQL路由模块负责选择一个连接，将语句发送至合适的DB节点上执行。

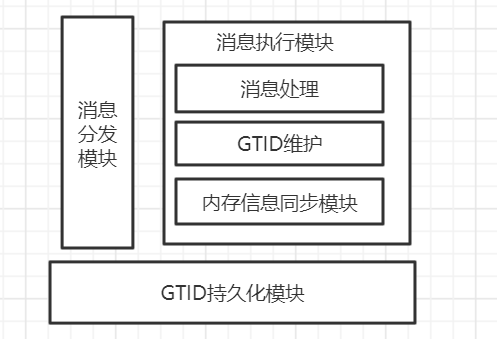
### 配置加载及处理模块

配置加载及处理模块蛀牙负责接收统一管理传递过来的配置变更请求，并实施这些变更。

## GTM

全局事务协调中心，用于协助Proxy进行分布式事务管理，主要包括全局事务ID的生成和活跃事务的维护以及当前活跃GTIDs的快照。

### 逻辑架构



消息分发模块：负责将收到的外部请求消息根据clusterid转发至相应的消息处理模块。

消息执行模块：负责处理收到的外部请求消息，并进行GTID的维护，同时将内存消息同步至备机模块。

数据持久化模块：后台定时将激活的GTIDs持久化，同时需要考虑其他手段降低GTIDs丢失的概率，但仍可能丢失一部分最新的GTIDs，如果GTM异常，则需将大于已持久化的GTIDs的所有分布式事务回退，这样可能会误差一些已经结束的分布式事务，但是却是保证数据强一致性的很安全的方案。

### 设计要求

#### 性能要求

为提高GTM全局争抢GTID的性能，GTM的线程结构设计为一个分发线程加若干可配置的执行线程。

1. 消息分发模块根据请求消息中携带的clusterid将其分发至固定的执行线程
2. 每个执行线程对应的clusterid维护一套独立的GTIDs
3. GTM应能根据clusterid和proxyid索引到GTIDs

#### 可靠性设计

GTM采用双活的模式，双机之间有心跳机制，双机内存同步更新。

脑裂问题的解决：引入第三方监控，在主备机之间发生网络故障时，由第三方决定。认为不存在双机之间及双机和第三方之间均出现网络故障的情况。

### 典型流程

#### 线程梳理

持久化管理线程：GTMSAVE

公共处理线程：DTMDIS

Sequence处理线程：SEQ

集群对应的线程：

工作线程：GTM1

GTID内存管理线程：GTIDMEM1

Sloth线程：

ZK线程：SLOTH\_ZOOKEEPER

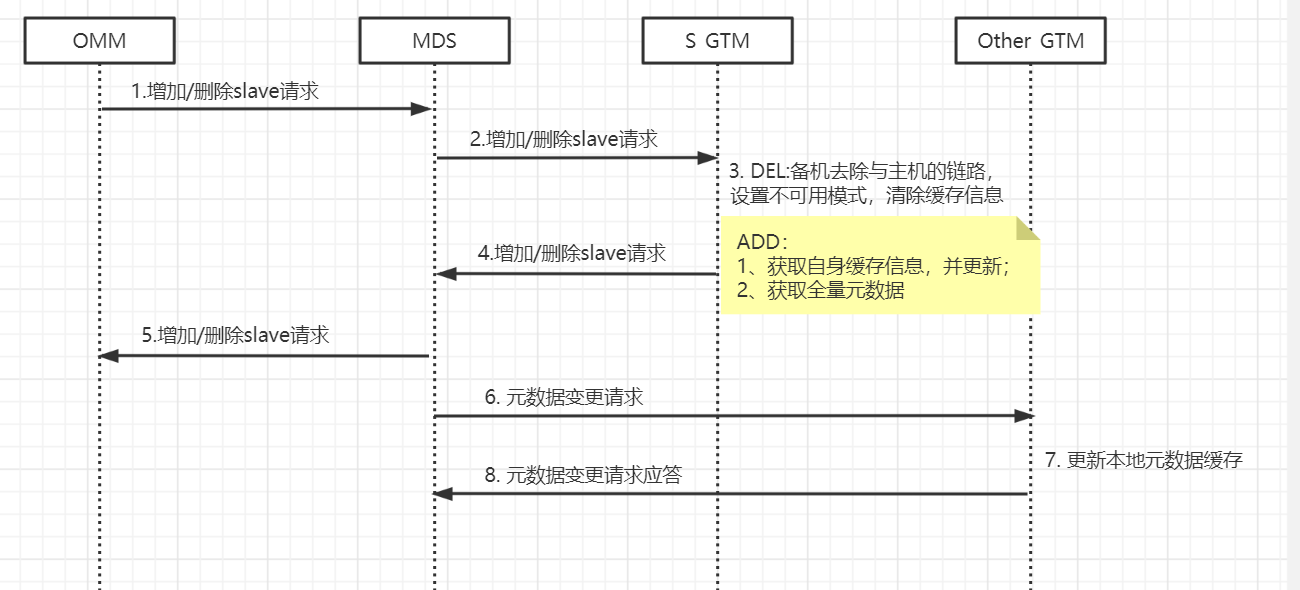
持久化线程：SLOTH\_PERSISTPNO

发送线程：SLOTH\_SENDPNO

#### 公共处理线程定时器

dis线程共有8个定时器，在dis的线程入口函数GTMDISProcess中，收到消息后，首先检查是否为定时器消息。

#### 增加/删除GTM



#### GTM全量同步

主GTM向所有备机发送全量同步请求消息，备机收到消息EVENT\_COMMON\_M\_S\_REQ后，会向主应答消息，这个过程是检查主备间的链路关系，之后主收到备机应答消息EVENT\_COMMON\_S\_M\_ACK事件后，会给GTIDMem线程发消息，将主GTID的消息发送给备机，备机收到全量GTID同步消息EVEVT\_TOTAL\_GTID\_SYNCHRO\_REQ后，通知自己的GTIDMem线程更新本地的GTID信息，完成后，给主回应答消息。

#### GTM主备切换

#### MDS

##### 获取元数据

1. 程序启动后，在dis线程初始化函数中，会设置定时器2，定时器2默认时间是1s
2. 定时器2触发后，通过状态控制进行，状态有如下几种：

//启动请求元数据状态

typedef enum{

STATUS\_INIT = 0,

STATUS\_UNSEND,

STATUS\_SEND,

STATUS\_INVALIDGTM,

STATUS\_PECEIVED, //获取元数据完成

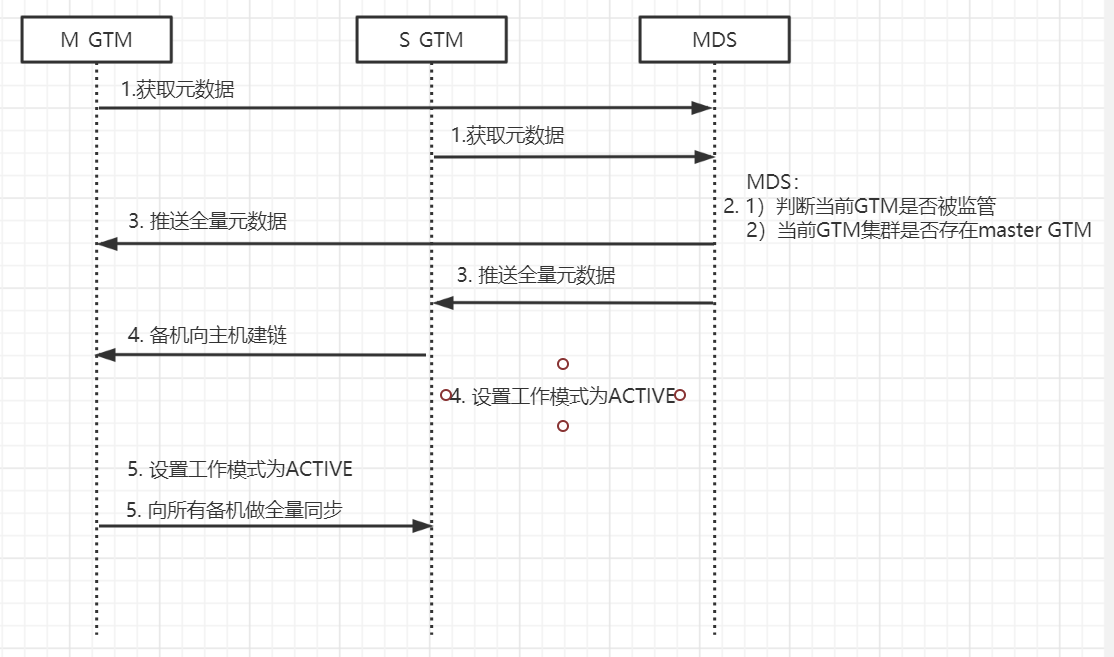
STATUS\_READY,

STATUS\_FINISH,

STATUS\_FINISH\_PINGERROR

}E\_MetaStatus;

1. 定时时间内会向MDS发送获取元数据请求消息，等待MDS回复后处理，每完成一个状态会设置响应的状态，此过程中定时器一直在等待状态，等解析完元数据应答消息，会根据消息修改缓存及设置当前工作模式。



##### GTM给元数据上报状态

GTM给MDS上报状态，有两个消息：

**EVENT\_STATUS\_REPORT**

1、此消息为所有GTM（主和备）定时5s上报给MDS自身状态，此消息中上报的状态是GTM自身主备的角色状态，也就是把自己是主GTM还是备GTM高速MDS，其中：

SLAVE 备机 0

MASTER 主机 1

初始化状态 NONVALID 2

2、在GTM启服中（CLUSTER\_ENABLING）或者停服（CLUSTER\_DISABLING），不会上报给MDS自身状态。

**EVENT\_SLAVE\_STATUS\_REPORT\_REQ**

1. 此消息为主GTM主动上报备机状态（是否可用），上报的备机状态为如下的情况：

SALVE\_ABNORMAL 备机不可用 0

SLAVE\_STOPSERVICE 备机停服 1

SLAVE\_RECOVER 备机已恢复 2

1. 上报备机状态的触发场景：
2. 当备GTM与主GTM断链后，主GTM上报此备机状态为不可用
3. 当链路异常恢复后，全量同步成功后，主GTM上报此备GTM状态未可用
4. 当备GTM被切换为主GTM，设置当前工作模式为MODE\_ACTIVE时，会将所有与主GTM断链的被GTM上报给MDS，状态为不可用
5. 全量同步完成后，如果同步成功，主上报此备可用，如果同步超时，主上报此备不可用

#### 与Proxy交互

对外支持的功能包括：

1. 接收处理proxy的create GTID请求，顺序递增生成全局唯一的事务ID；
2. 接收处理proxy的release GTID的请求，释放对应的全局事务ID；
3. 接收处理proxy的查询归属于特定集群的当前激活的GTIDs请求；
4. 接收处理proxy的查询归属于特定proxy的GTIDs的请求。

GTM仅维护当前激活的GTIDs，其属性值包括但不局限于GTID归属的proxy、clusterid信息。

##### GTID申请控制

写事务控制：

除了DELETE，所有单节点操作都不申请GTID，多节点都申请GTID。

###### INSERT

|  |  |  |
| --- | --- | --- |
| INSERT语句 | 单节点 | 多节点 |
| SW | 不申请GTID | 申请GTID |
| CW | 不申请GTID | 申请GTID |

###### UPDATE

update单节点不申请GTID是因为不存在已提交事务的回滚情况，由mysql自己回滚：

|  |  |  |
| --- | --- | --- |
| UPDATE | 单节点 | 多节点 |
| SW | 不申请GTID  不需要select for update | 申请GTID  不需要select for update |
| CW | 不申请GTID  需要select for update | 申请GTID  需要select for update |

###### DELETE

|  |  |  |
| --- | --- | --- |
| DELETE | 单节点 | 多节点 |
| SW | 不申请GTID  不需要select for update | 不申请GTID  不需要select for update |
| CW | 申请GTID  需要select for update | 申请GTID  需要select for update |

###### SELECT

读事务的控制：

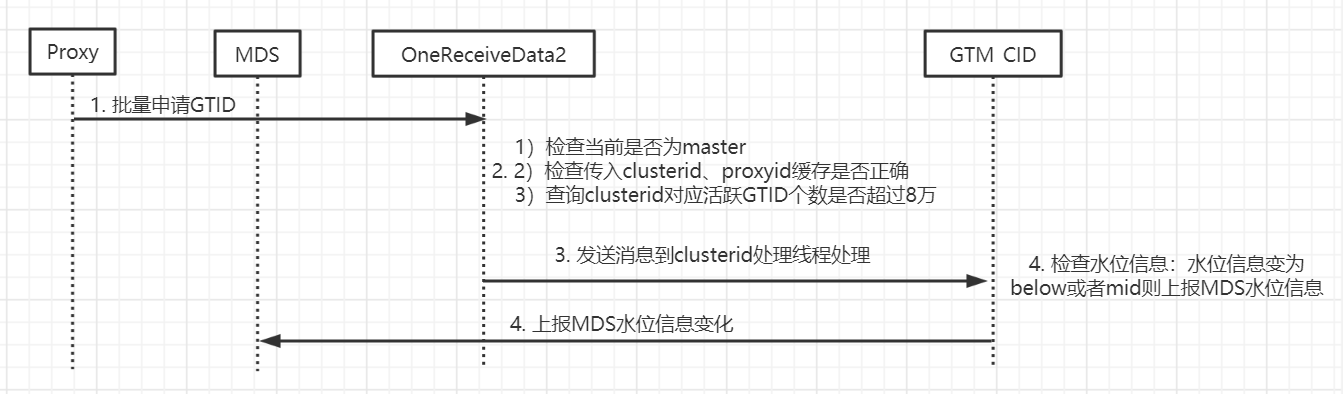
|  |  |  |
| --- | --- | --- |
| SELECT | 单节点 | 多节点 |
| UR | 不查询GTID  不查询活跃事务 | 不查询GTID  不查询活跃事务 |
| SEMI-CR | 查询GTID  查询活跃事务 | 查询GTID  查询活跃事务 |
| CR | 查询GTID  查询活跃事务 | 查询GTID  查询活跃事务 |

|  |  |  |
| --- | --- | --- |
| 汇聚函数的控制 | 单节点 | 多节点 |
| UR | 不需要lock in share mode | 不需要lock in share mode |
| SEMI-CR | 不需要lock in share mode | 需要lock in share mode |
| CR | 不需要lock in share mode | 需要lock in share mode |

注：汇聚函数是无法保证强一致性的，这主要是为了防止select for update加锁扩大导致全表锁，所以汇聚函数采用共享锁lock in share mode，这就导致会出现多个同时访问一组数据，并进行修改，这个是最终一致性而不是强一致性。

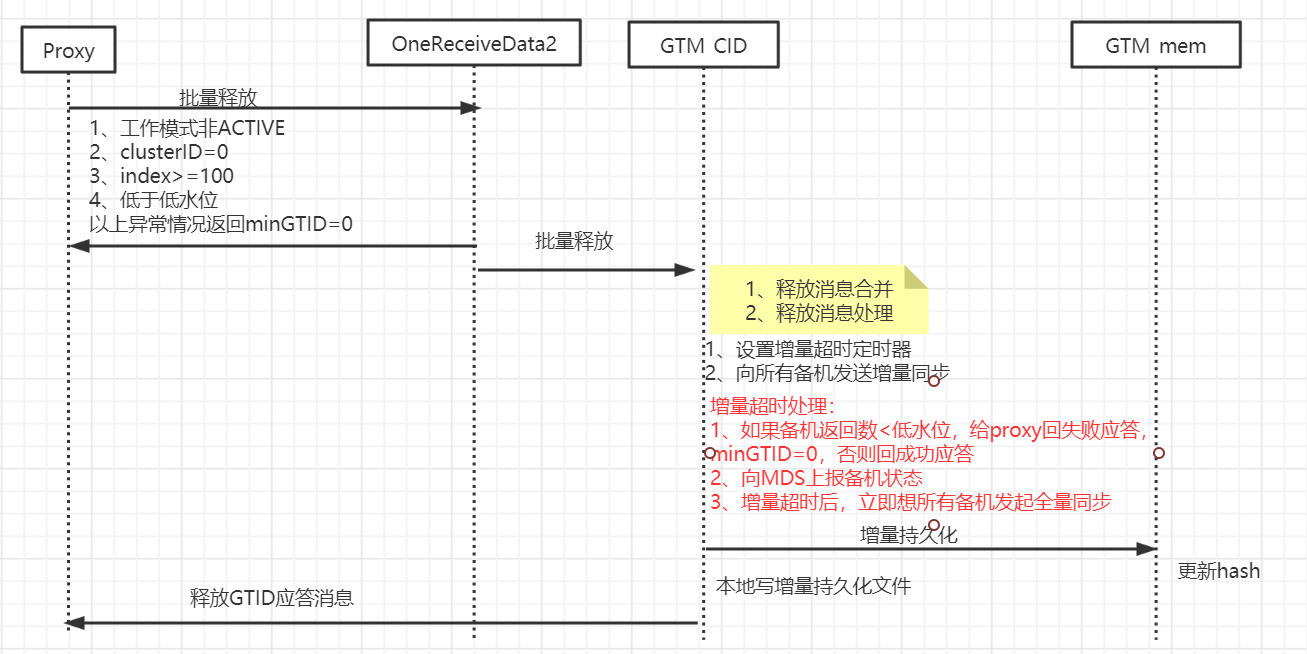
##### 批量申请GTID

1. GTM收到Proxy申请GTID消息



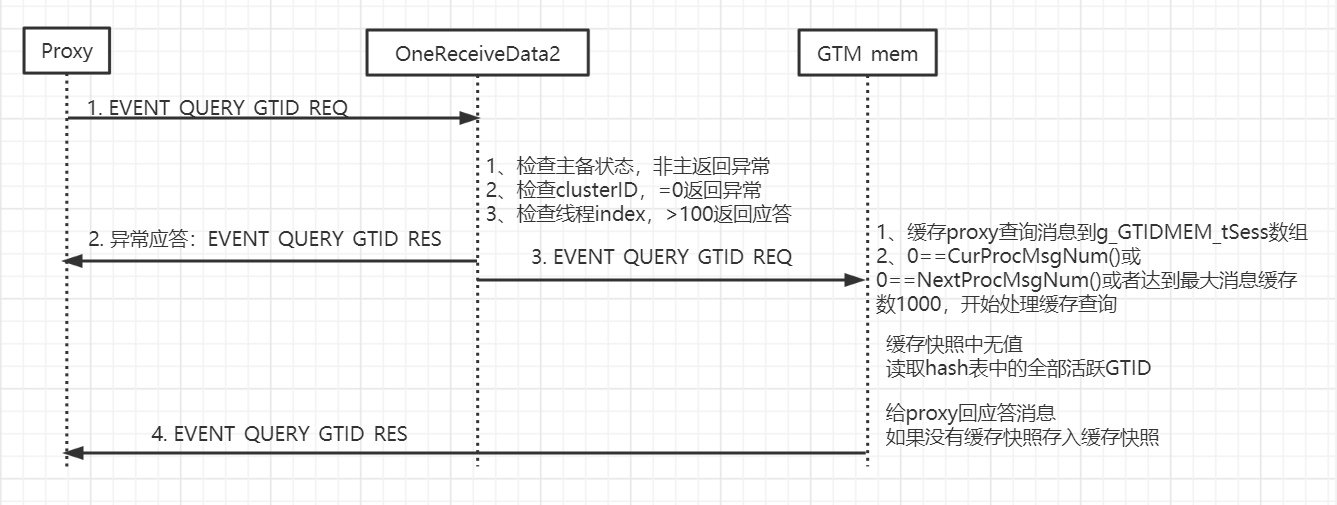
1. Master消息合并及处理

##### 批量释放GTID



##### Proxy查询活跃GTID

定时查询活跃GTID消息：EVENT\_QUERY\_GTID\_REQ



##### GTM向proxy反查GTID

基本处理逻辑：

1. 全量持久化的时候，会触发活跃GTID反查
2. 反查的时候，先将所有活跃GTID进行第一轮过滤，过滤出超过阈值（默认5分钟）的个数，同时记录符合条件的ProxyID

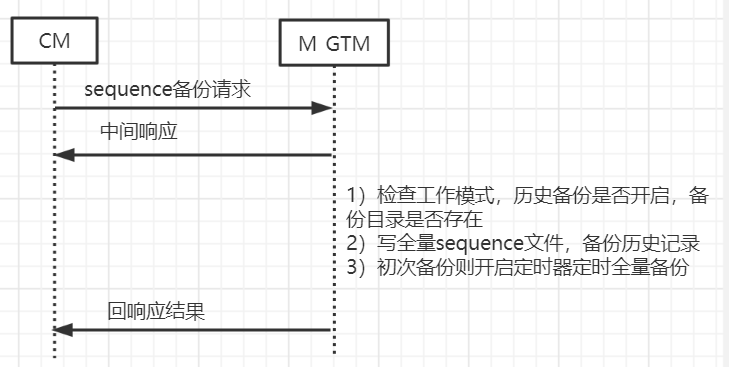
这里处理的巧妙之处在于，所有活跃的GTID其实是按照申请时间的先后顺序存储的，当发现一个不符合要求的（即时间没有超过要求的5分钟），那么就不再进行下面的GTID比较，直接进行第二轮的筛查

1. 第二轮进行ProxyID的筛查，根据第一轮筛查出来的ProxyID，检查链路是不是通的，向所有链路通的ProxyID进行反查
2. 在进行第二轮筛查的时候，需要判断的是ProxyID符合要求，并且不能带有XA信息，XA活跃的GTID不需要进行反查

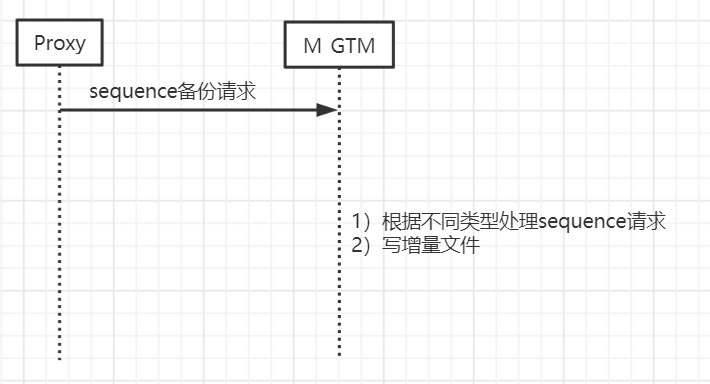
#### Sequence操作

##### 备份sequence

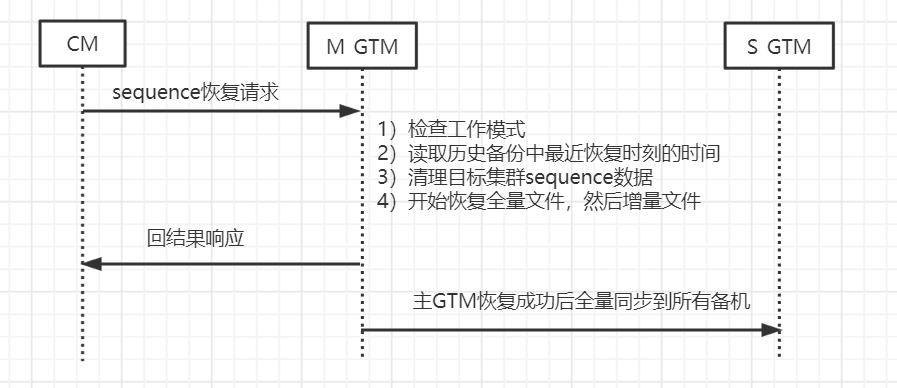
CM->M\_GTM：全量备份



Proxy->M\_GTM：增量备份



##### 恢复sequence



#### Sloth高可用

### 性能测试

### 使用限制

## MetaDataServer

下文所述元数据是一个广义的概念，它在普通意义上的数据库元数据概念基础上增加了集群的结构管理信息。

元数据管理模块负责对元数据创建、修改、查询、删除、持久化等的管理公国；采用主备方式提供高可靠性和高可用性。

### 逻辑架构

MDS主要有如下的几种类型的数据：

1. 数据字典。

如表定义、索引定义、存储过程定义、函数定义等数据库对象定义信息。

1. 分片信息。

当表需要分布式存储时，就需要定义表的分布策略信息，这些形成分片信息，用于描述表数据分布的算法、分布到哪些节点等。

1. 数据库集群信息。

集群基本信息，如集群ID；集群内部各个DB节点信息。

1. Proxy集群信息。

如proxy和Cluster连接实例的对应关系等。

**元数据持久化功能：MDS保存有最全的信息，需要实时持久化，底层挂载mysql数据库（即RDB）。**

查询及推送功能：支持PM、CM等模块的元数据查询功能。

变更功能：支持CM、OMMServer等模块发起的元数据变更，并控制变更过程，协调各个模块有序完成元数据的变更。

### 设计要求

#### 可靠性设计

MDS采用主备的部署，每台机器上均部署mysql数据库，主备之间使用半同步方式保持数据一致性。在主机上的进程或数据库发生异常时，切换到备机。

### 典型流程

MDS是通过RDB数据库存储各种元数据信息的。

#### 实施proxy集群管理

主要指实施OMMServer发送过来的用户的proxy集群管理命令，包括proxy的增加、删除、启用及禁用。

#### 实施连接实例管理

主要是指实施OMMServer发送过来的用户的连接实例管理命令，包括连接实例的增删改、连接实例的绑定、启用、禁用、变更、解绑。

#### 实施cluster集群管理

主要指实施OMMServer发送过来的用户的cluster集群管理命令。

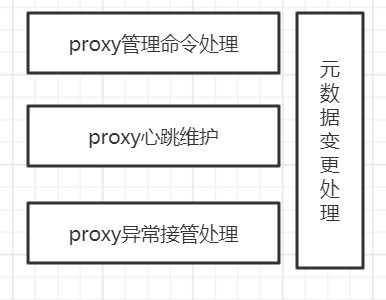
#### 接收数据信息查询

MDS可以接受回复各其他模块发起的元数据查询请求。

## ProxyManager

Proxy集群管理模块，包括proxy的发现、异常、查询，并接收处理MDS传递过来的用户操作。

### 逻辑架构



Proxy命令管理处理模块：负责将接收到的MDS的操作命令进行具体实施。

Proxy心跳维护模块：负责对proxy状态进行监控。

Proxy异常接管处理模块：负责在proxy发生异常时自动触发异常proxy的在线活跃事务回滚。

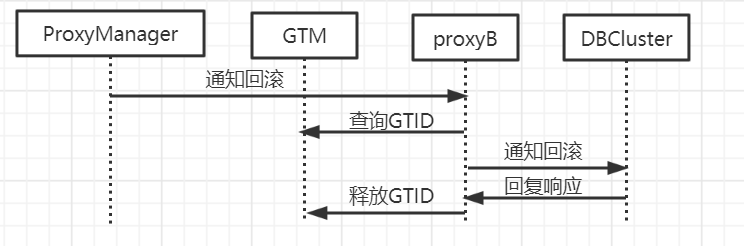
元数据处理：负责接收外界的元数据增量变更请求；同时可以根据任务定期去MDS获取全量数据。

### 典型流程

#### 异常处理

Proxy发生异常，其和PM的心跳异常，PM发起异常处理流程：

1. PM通知和当前异常proxyA集群上对应proxy发起回滚流程，图中以某proxyB为例；
2. ProxyB去GTM查询该异常proxy对应集群上的活跃的GTIDs忙活去其需要处理的部分，向对应cluster发起回滚流程。



#### 系统全部异常时的恢复机制

PM开放出接口供运维工具使用。当系统全部异常掉电重启后，通过该工具触发，PM能够通知各个proxy发起全局事务回滚。

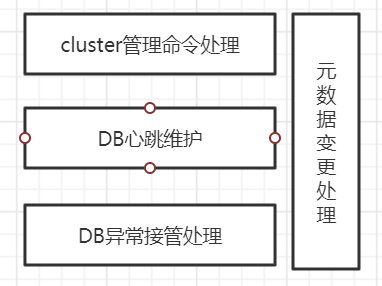
## ClusterManager

数据库集群管理模块，负责配合实施集群的管理，包括集群发现、集群组建、集群变更、集群查询、集群监控等功能。使用双活机制确保高可用性。

Cluster的管理较proxy管理更加复杂：

1. 设计多层管理（cluster、group、db）
2. 在发生自动主从切换时CM和OMMServer存在同时操作资源的情况：
3. 表设计上需要对DB的状态位和操作标志位细化考量；
4. CM在发起自动主从切换时，发消息至MDS进行操作标志位申请。

### 逻辑架构



Cluster命令管理处理模块：负责将接受到的MDS的操作命令进行具体实施。

DB心跳维护模块：负责对各个DB状态进行监控。

DB异常接管处理模块：负责在DB发生异常时自动触发主从切换流程。

元数据管理：负责接收外界的元数据增量变更请求；同时可以根据任务定期去MDS获取全量数据。

### 典型流程

#### 集群监控

当集群内部某DB发生异常时，CM能够感知DBAgent的心跳异常，触发主备切换，集群里的master访问地址信息发生变化。

基本流程：

1. master出现异常，DBAgent对其尝试重启恢复失败；
2. DBAgent上报CM DB异常；
3. CM进行告警，同时判断若其具备主备切换条件，通知相关的proxy停写，通知MDS执行主备切换；
4. CM通知slave切换为master；
5. Slave完成配置调整和binlog replay后回复CM切换结果；
6. CM将集群信息变化通知给MDS；
7. CM通知PM去向向硅谷那proxy推送集群变更信息。

## DBAgent

部署在DB节点上，负责DB的配置更改、复制变更、状态监控、统计上报等，同时配合实施事务回退、集群管理、数据备份、数据迁移等功能，DBAgent使得这些功能落地。

### 逻辑架构

### 典型流程

#### 告警和统计信息上报

1. DBAgent定时检测DB的状态，如果发生异常则尝试对其进行恢复，几次尝试失败后则上报CM DB异常；
2. DBAgent定时将DB上搜集的统计信息上报CM。

#### 主备切换

#### 已提交事务回滚

已提交事务的回滚是分布式事务方案能保证数据一致性的关键步骤，是DBAgent的最重要功能之一。流程操作如下：

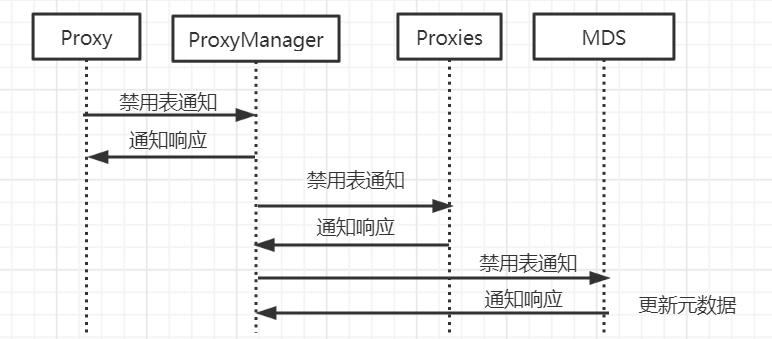
1. proxy通知DBAgent进行某GTID事务的回滚；
2. DBAgent分析读取binlog通过GTID定位到该事务的日志；
3. 根据日志中记录的新旧值构造反向SQL，对已提交事务进行回滚。

说明：

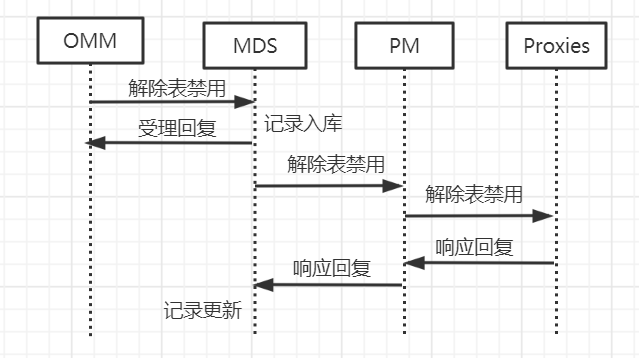
1. 搜索GTID高效；
2. 支持对指定GTID的事务及将大于指定GTID的事务回滚两种模式；
3. 不重复回滚；
4. 回滚失败时能够输出易于对账的日志，当找到该binlog，但无法实施回滚时，包括但不局限于以下场景：
5. 该记录发生了写写冲突，该记录在commit和rollback之间被其他事务修改
6. 该binlog涉及事务是一条delete语句，无法实施回滚

在发生对已提交事务回滚失败的场景时，需要将该事务涉及的表禁用，暂停对这些表的访问，具体分为如下场景：

1. 系统正常运行时，proxy中保留有全局事务的完整信息，可以发起对涉及的表的禁用操作；
2. 在系统发生异常时，如proxy异常，接管发起分布式事务回滚的proxy应能够从DBAgent上报的回滚失败语句中获取该事务所涉及的表，并对其发起禁用流程；



在人工干预恢复后，在界面发起接触表禁用操作，恢复对该表的访问：



## DB

数据节点，执行数据的读写操作和存储，基于MySQL8.0。

### 性能优化

#### 备机IO线程和SQL线程读写锁拆分

备机IO线程持有锁流程图：

备机SQL线程主要复制从relay\_log中读取event事件，分发到worker线程执行。读取relay\_log事件的主要函数为next\_event()，该函数中判断如果读取的relay\_log是hot\_log（即SQL线程读取的relay\_log和IO线程写入的relay\_log是同一个），则必须持有relay\_log::LOCK\_log锁。

其他几处需要使用relay\_log::LOCK\_log锁的场景为：

1. 调用flush\_master\_info函数时，flush\_relay\_log的io\_cache是会持有relay\_log::LOCK\_log锁，其主要再IO线程、change\_master调用；
2. 调用terminate\_slave\_threads函数时，停止IO线程和SQL线程时需要持有relay\_log::LOCK\_log锁，其主要在复制异常终止和stop slave时调用；
3. 调用init\_relay\_log\_pos函数时，初始化relay\_log位置持有relay\_log::LOCK\_log锁；
4. 调用rli\_init\_info函数时，判断是否为hot\_log需要持有relay\_log::LOCK\_log锁；
5. 调用MYSQL\_BIN\_LOG::append\_event函数时，需要持有relay\_log::LOCK\_log锁，该函数比较特殊，需要注意。

优化原理：

#### 主机收发event合并优化

### 功能优化

## DataManager

数据控制模块，主要负责数据重分布的管理、数据备份与恢复的管理。

### 备份与恢复

优先级提升。

### 数据重分布

在系统扩容，增加新的数据节点或表的原有数据分布策略不合理，导致性能问题、热点数据问题等场景下需要对原有数据进行重分布。

在决策需要对数据重分布时需要数据分布情况及访问情况的统计分析。

## ClusterReplication

集群复制模块，用于集群间数据复制，保证数据一致性。

## 模块间统一通讯（OS）

统一消息通信是ZXOS统一支撑的核心组件，完成消息线程的建立、调度、管理、回收等，负责底层的通信链路管理包括端口的侦听、建链、数据的发送和接收、多条并发链路的管理等。

## OMMServer

OMMServer通过web节点提供人机交互功能，具体包括集群规划、集群变更、元数据操作、权限管理、告警与监控、日志分析等功能。

OMMServer与MetaDataServer共用一个数据库，但实行分库管理，各自维护自己的数据；OMMServer会根据需要读取MetaDataServer的库做展示，但是不会对其进行写操作，要做好权限控制。

# 数据分布

## 概述

数据量达到1000万的数据表尽量分片。

集群中每个表中的数据按照某种策略进行横向分片后存放到对应的安全组中，分片策略包括复制策略、哈希策略、范围策略和列表策略。

注：也可以通过配置文件，采用默认hash或复制的分发策略。

### 优点

可以灵活实现数据的分片，尤其是多级分片表。

### 缺点

有一些比较复杂的条件查询无法精准命中分片，会群发，造成不必要的网络开销，且对于大数据量的重分布效率低，且会对在线业务产生影响。

## 数据分片规则

数据的切分（sharding）根据切分规则的类型，可以分为垂直切分和水平切分两种模式：

垂直切分是按照不同的表切分到不同的数据库中，适用于业务系统之间耦合度低、业务逻辑清晰的系统。

水平切分是根据表中数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库上，对应用来说更为复杂。

数据分片是将数据库横向扩展到多个物理节点的分布式技术，具有以下特性：

无限扩展：数据分片水平切分扩展到多个物理节点，理论上支持无限扩展

性能提升：数据分片以后单个数据节点上上的数据集变小，数据库查询的压力变小、查询更快，性能更好；同时水平切分以后查询可以并发执行，提升了系统的吞吐量

高可用：部分分片节点宕机只会影响该部分分片的服务，不会影响整个系统的可用性

在GoldenDB中支持的分片规则和使用场景如下：



以下部分将介绍GoldenDB的几种分片方法：



文本

描述已自动生成

### 复制策略

#### 算法

复制表是在数据节点保留全量的数据副本，有多节点复制表和单节点复制表，通常用于表的数据读多写少变化不大的场景，比如参数表。复制表的好处是将公共访问表存放在本地数据节点，可以减少跨节点的表数据访问和分布式事务开销，提升SQL的性能。

#### 语法

GoldenDB中复制表示例：

Create table t1(a varchar(4),b char(6)) distributed by duplicate(g1,g2,g3);

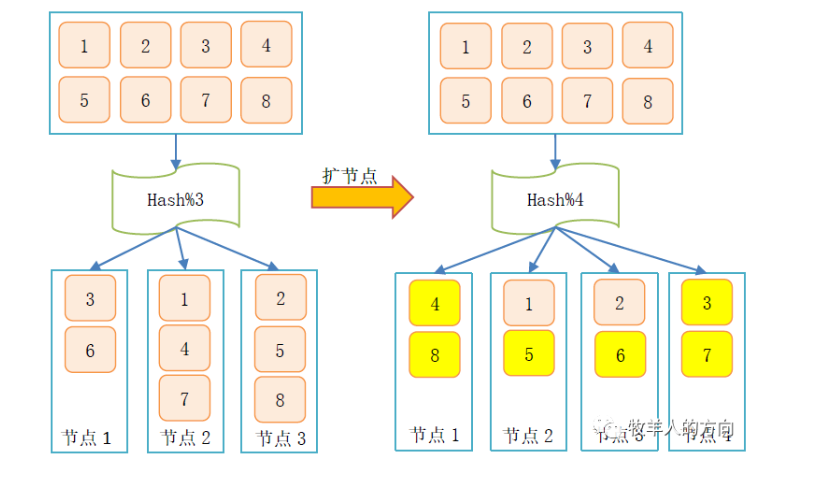
#### 适用场景

复制策略（duplcate）：适用于不经常修改（因为修改需要同步多个节点，比较耗时），且频繁出现在关联或子查询中的小表。复制策略下，复制表的数据保存在每一个节点都需要这个表中数据的情况下，可以减少节点之间网络数据（RPC）的传输，提高查询性能。

### 哈希策略

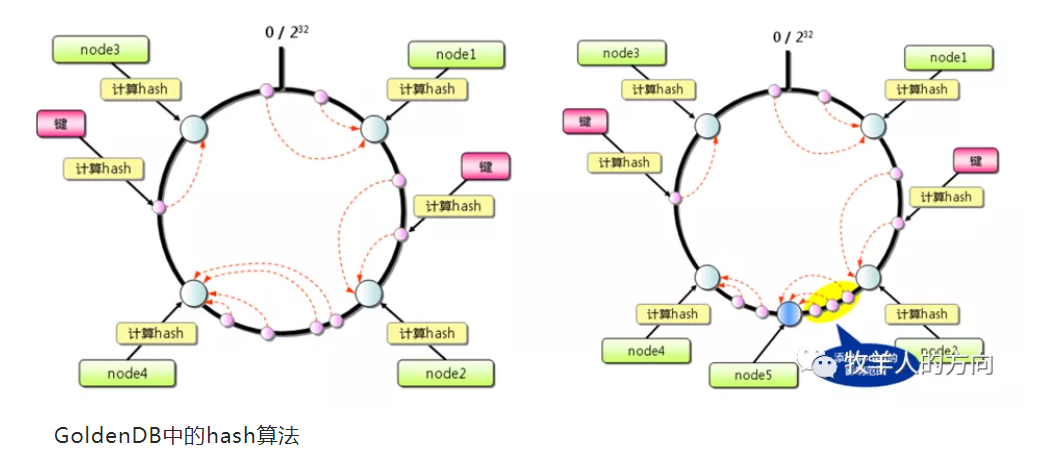
#### 算法

Hash分片的算法就是对分片键的值进行hash（取模）计算，将计算后的key值映射到不同的服务器上，这样的好处是能将海量的数据均匀的打散在不同的数据节点上。但是普通的hash算法在节点扩容时候，需要对所有的数据进行重新打散，存在大量的数据搬迁。



一致性Hash算法最早是解决分布式cache提出的，它将整个hash值空间组织成一个虚拟的圆环，根据节点名称的Hash值将服务器节点放置在这个Hash环上，然后根据数据的Key值计算得到其Hash值，接着在Hash环上顺时针查找距离这个Key值的Hash值最近的服务器节点，完成Key到服务器的映射查找。

如下图所示，当增加新的节点的时候，只有在圆上增加服务器的逆时针的第一台服务器上的主键会受到影响。这种算法解决了普通余数Hash算法伸缩性差的问题，可以保证在上线、下线服务器的情况下尽量有多的请求命中原来路由到的服务器。



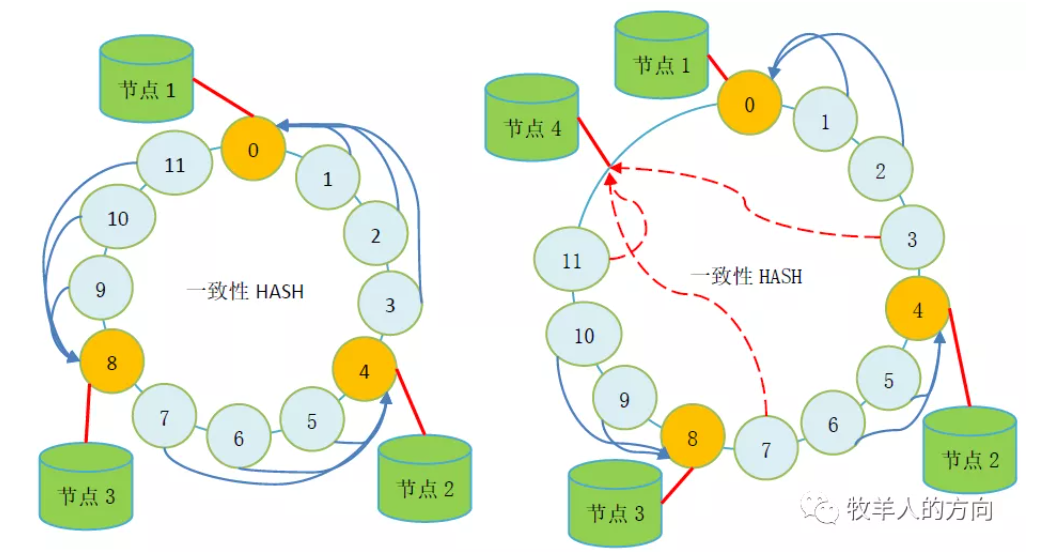
GoldenDB中使用的是一致性hash算法，它可以尽可能减少节点数变化引起的数据迁移。

对分片键计算hash值，该hash值会落到0~N的封闭圆环中

按照分片数量M将hash值均分为M段（M<<N），确定每个分段的值范围

将落入同一个分段的hash值划分在同一个分片上

当增加数据节点时候，会使用到hash流式重分布策略，将其它分片的hash值搬动到新增的节点，使得数据均匀分布



#### 语法

Hash分片的语法如下：

Create table t1(a int,b int) distributed by hash(a)(g1,g2,g3);

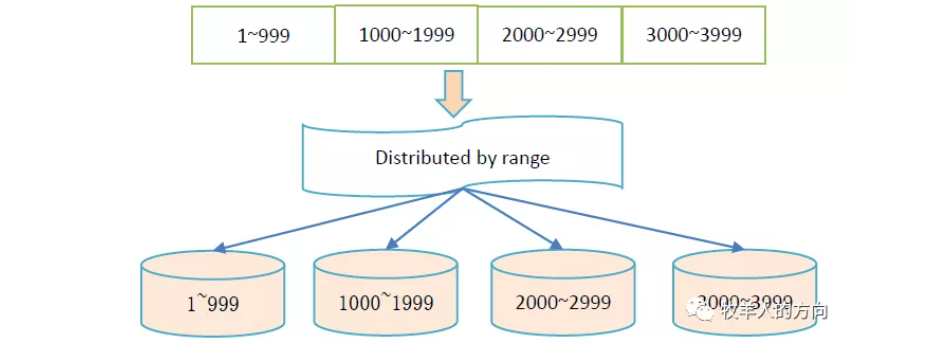
#### 适用场景

哈希策略（Hash）：适用于将数据均匀的分布到预定义的各个安全组中，保证各个安全组的数据量大致一致。一般用户不需要关心分发字段的取值范围和具体含义，且对该表的SQL操作基本都是等值操作的场景。

### 范围策略

#### 算法

Range分片是按照数据范围分片，常用于按照时间分区或者ID分区来切分，比如按照不同日月的数据分散到不同的库中。Range分片优点在于集群扩容时候只需要添加节点即可，不需要对分片数据进行迁移；同时对于范围查找时，可以通过分片数据快速查询，有效避免跨分片的查询问题。Range分片也有很明显的问题就是出现冷热数据，以及热点数据集中在单个数据节点上出现性能瓶颈。



#### 语法

GoldenDB中range分区示例：

Create table t1(a bigint,b char(6)) distributed by range(a)  
(g1 values less than (1000),g2 values less than (2000),g3 values less than maxvalues);

#### 适用场景

范围策略（range）：适用于指定一个给定的列值或列值集合应该保存在哪个安全组上，常用于时间、日期、数值等类型的字段上，如数据按照自然月或者自然天分布存储。

### 列表策略

#### 算法

List分片是根据不同的枚举值将数据分布在不同的分片上，比如分片键为银行法人，将不同的法人部署在单独的分片上。List分片适合分片键数据较少且固定的场景，但是不同枚举值的数据分布可能出现不均衡的问题。

#### 语法

GoldenDB中List分片示例：

create table t1(a varchar(4),b char(6)) distributed by list(a)  
(g1 values in(‘CN’),g2 values in (‘HK’));

#### 适用场景

列表策略（list）：适用于含有一定列限定性或枚举性的字段上，如数据按照机构代码、国家代码、地区代码分布存储。

数据按照上述策略（复制策略除外）分片后，每个安全组上的实际数据在理论上只有总数据量的1/N（数据分布的均匀程度依赖于切分策略和真实数据分布的匹配程度）。随着安全组数量的增加，每个安全组承载的数据量和读写负载会相应的减少，从而在数据节点集群内部具备了读写能力的水平拓展。

安全组是由一个或者多个数据节点组成的数据库节点组，组内的数据库节点拥有相同的数据。当安全组存在多个数据节点时，其中一个数据节点为主用节点其他数据节点都是备用节点，数据在主备节点之间实时复制。主用节点具备读写能力，备用节点可以提供度能力。

### 多级分片表

#### 算法

多级分片是通过多维属性对数据分片进行精确控制，通常是通过多个字段进行多层次分片，比如通过法人字段进行一级分片，再按照客户类型字段进行二级分片。

#### 语法

GoldenDB中多级分片示例：

Create table t1(a int,b int,c int) distributed by   
 case c when 1 then case when b<100 then subdistributed by hash(a) (g1);  
 else subdistributed by hash(a) (g2);  
 end case;  
 when 2 then subdistributed by hash(a) (g3);  
 else subdistributed by hash(a) (g4);  
 end case;

GoldenDB对同一张表并不是只能采用一种分片策略，还可以使用多机分片功能同时使用多种分布策略：

case where pk=1 distributed by hash(name)(g1,g2)

where pk=2 distributed by hash(age)(g1,g2)

else distributed by duplicate(g1,g2);

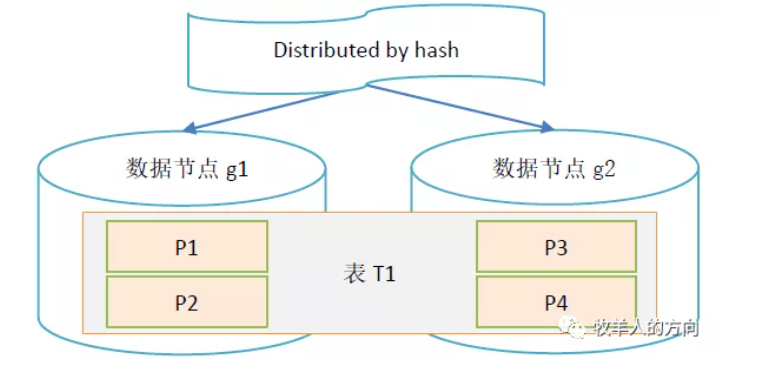
#### 适用场景

GoldenDB支持最多5层的分片规则设置，适用于需要精细化控制数据在集群中的分布形态的场景，如多法人场景，将不同法人的数据划分到不同数据安全组。

### 分片+分区

#### 算法

当表在横向切分后，单节点的表数据依然很大，可以考虑进行纵向分区，将节点上的分片数据进一步进行切分。这样做的好处是比如一些流水表按照时间进行分区，历史档需要定期清理的时候直接drop历史分区即可；同时业务访问表数据的时候可以减少读取的表数据量，快速定位数据提升性能。



#### 语法

GoldenDB中分片+分区示例：

Create table t1(a int,b int,c int) partition by range (c)  
 (partition p1 values less than (20210101), partition p2 values less than (20210201),  
partition p3 values less than (20210301), partition p4 values less than (20210401))  
 distributed by hash (a) (g1,g2);

## 分片路由



### 表分片DDL过程

GoldenDB中表分片的DDL会在每个DB数据节点中执行并保存一份完整的DDL信息，整个过程如下：

DBProxy接收到DDL信息后，会通知MDS更新元数据信息，并持久化保存到RDB中

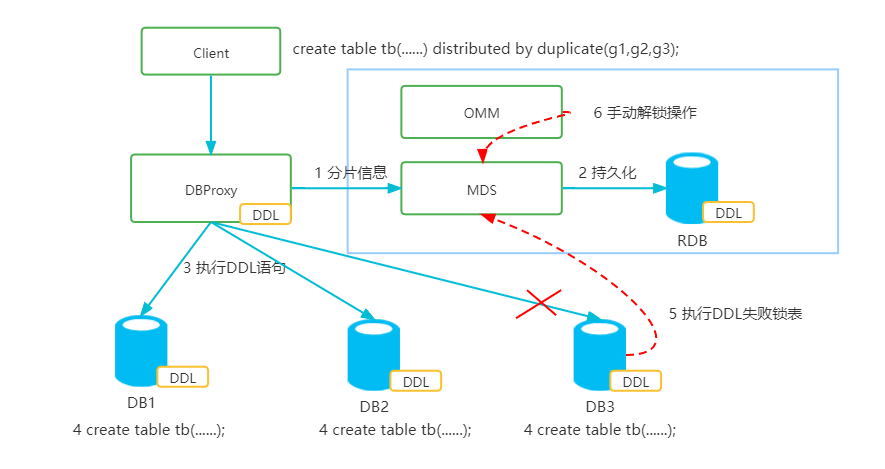
DBProxy将DDL语句下推到每个数据节点分别执行

DBProxy本地内存和数据节点中会保存一份全量的表结构信息

DDL执行过程中如果出错，会通知MDS将表状态禁用，需要手动解锁；表禁用后业务访问会出错

RDB中的DDL信息会定期同步到DBProxy计算节点和DB数据节点

应用访问时会优先从本地读取DDL信息



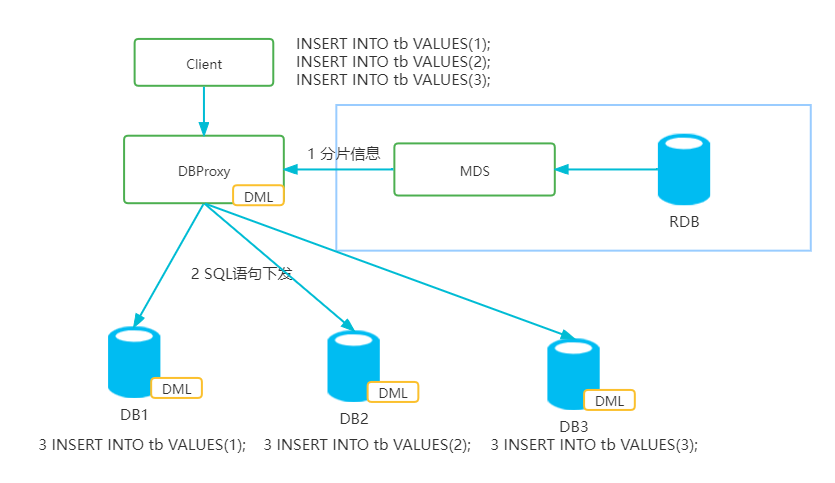
### 表分片的插入过程

GoldenDB中分片表的插入过程如下：

DBProxy接收到SQL信息后，从MDS获取表的分片信息并解析SQL

DBProxy根据分片规则将SQL语句下发到不同的DB分片节点

DB分片节点执行insert语句，并返回执行结果给计算节点DBProxy



根据路由规则可以看到，相同的表名在不同的DB数据节点执行的是不同的SQL语句。

### 表分片的查询过程

GoldenDB中分片表的查询过程如下：

DBProxy接收到SQL信息后，从MDS获取表的分片信息并解析SQL

DBProxy根据分片规则将查询语句下发到不同的DB分片节点

DB分片节点执行SQL查询语句，并返回结果集给计算节点DBProxy进行汇总，再返回给客户端



注：对于不带分片键的查询，会将DB数据节点的查询结果汇总到计算节点，当数据量很大时会出现计算节点OOM情况。

## 重分布

### 原理

实现数据重分布功能的基本策略是：针对要重分布的表，建立一个和原表字段结构相同但是分发策略为目标分发策略的新表，对于需要重分布数据的group节点，将其上的数据导出，然后将该部分数据导入到新表，导入过程会使用新表的分发策略。因此可实现数据在group节点间的重分布，最后将新表改名为原表。

重分布分为两大阶段：

1. 第一阶段是线下操作，进行全量数据的导出和导入、数据校验，所需要的时间非常的长，但是不影响在线业务；
2. 第二阶段是线上操作，对线下操作期间的增量数据进行迁移，数据校验，所需要的时间短，但是影响在线业务的写。

#### 分发方式

##### 哈希分片

适合分片键粒度细而均匀的表，如账号表按照账号进行hash分片，哈希分片是海量数据的主要分片规则，能将大量的数据均匀地打散。

##### 范围分片

适合分片键可以顺序比较的表，如流水表按照日志进行分片。

##### 列表分片

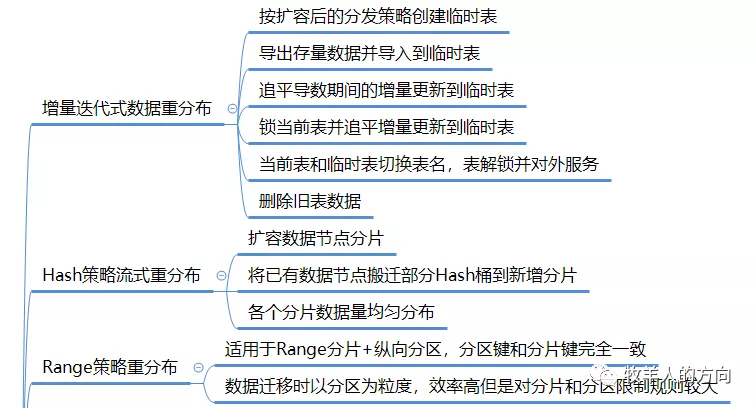
适合枚举类型字段的分片键，比如：银行法人。可以根据不同的法人将表数据分布在不同的分片上。

##### 复制分片

适用于配置表，在每个分片上都有一个全量的数据。

#### 数据重分布策略

GoldenDB支持动态的数据重分布，将数据高效均匀的分布在数据节点上，可实现在线重分布，对业务影响小。重分布过程通过在OMM管理界面以管理任务的形式运行，执行、暂停、继续取消以及异常情况下的重试等，全程可视化控制。同时，数据重分布根据不同的场景有不同的策略，常见的有：增量迭代式数据重分布、HASH策略流式重分布以及RANGE策略重分布优化。

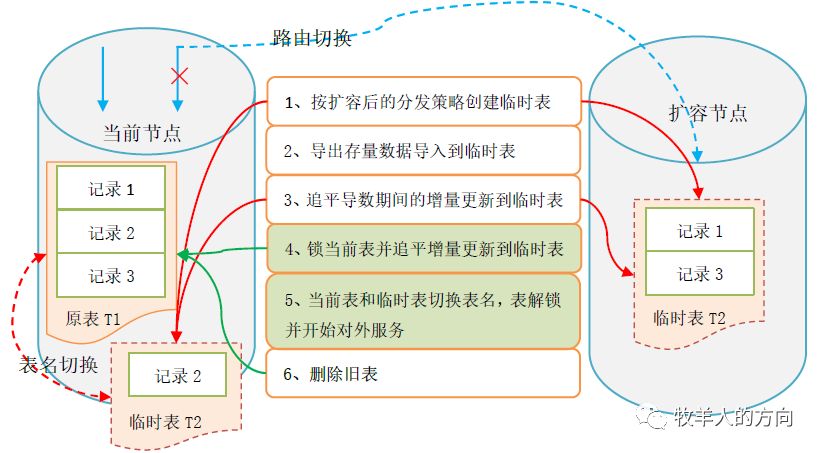


##### 增量迭代式数据重分布

增量迭代式数据重分布是一种通用策略，适合所有的数据重分布场景，比如：

1. 分片策略的变更，由hash变为range，复制表变为分片表等
2. 表分片键的变更，由A列调整为B列
3. 分片策略不变，数据节点横向扩容的场景

增量迭代式数据重分布过程如下图所示：



实现数据重分布策略的基本策略是针对重分布的表，建立一个与原表字段结构相同但是分发策略为新的分发策略的新表，然后将重分布的数据节点上数据导出并导入到新表，最后将新表和旧表的表名切换，总的过程如下：

1、按照扩容后的分发策略创建临时表

2、导出需要重分布的数据节点的数据，并导入到临时表

3、通过binlog数据追平导数期间增量更新的数据到临时表

4、将当前表锁住（此时应用不能更新表），并通过binlog追平增量更新数据

5、数据校验完毕后，将临时表和新表切换表名，解锁并对外提供服务

6、删除旧表的数据

**数据重分布过程中需要注意：**

锁表和切换表名的过程中，影响应用的写操作

重分布过程中临时表需要额外的存储空间，重分布操作前需要保证存储空间充足

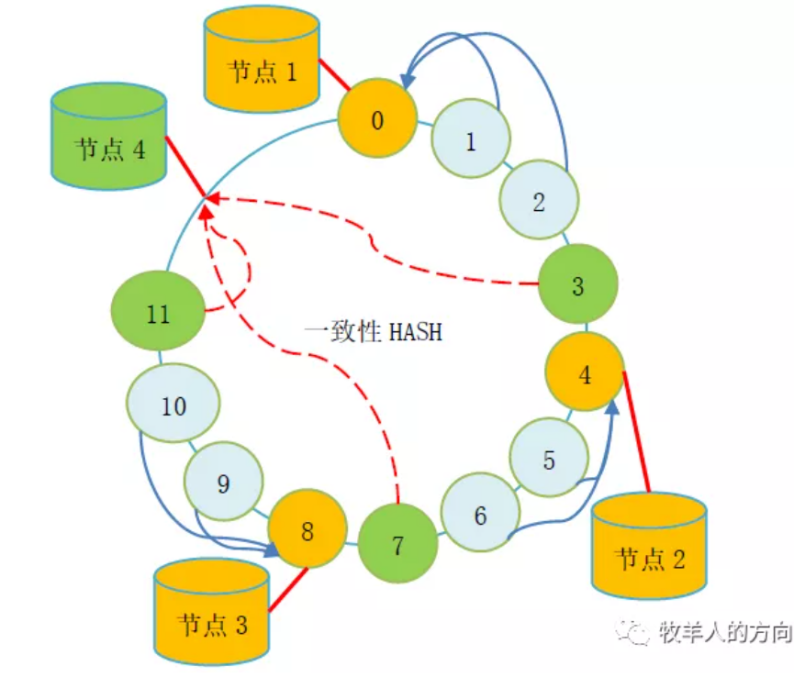
##### Hash策略流式重分布

Hash策略流式重分布是对通用重分布策略的优化，适用于分片键信息不变，仅进行Hash分片策略横向扩容的场景。

系统中默认包含2048\*128个HASH桶，新增数据节点的过程中会并行的从现有的每个分片节点迁移部分数据到新增分片上，确保所有分片数据平均

整个数据迁移的过程是在现有分片数据上动态操作，对磁盘空间没有额外的要求

同步更新MDS和DB节点中表的分片信息



Hash策略流式重分布还有几个细节问题尚待明确：

分片数据从现有分片移动到新增分片的方法，直接导出再load进去？

数据移动的过程中，应用访问到的数据的节点信息和移动后的数据信息不一致该如何处理？

现有分片数据移动到新增分片的算法，即哪些数据会搬到新的分片节点

Hash桶数量是有限的，是否会出现Hash值重复的情况？

#### 重分布任务



##### 创建重分布任务

重分布任务通过OMM管理平台创建，主要分为三步：

解析重分布任务：MDS对重分布任务进行检查，解析出新的重分布策略，并根据表的老旧分发规则解析出本次重分布涉及哪些分片

分解重分布步骤并保存重分布任务：MDS会把重分布任务分解为执行步骤，并保存到元数据RDB中

创建临时表：MDS向各DB分片发起创建临时表的请求，临时表的命名规则为原表名\_时间戳

重分布上述三个步骤其中一个失败，整个重分布任务即为失败。同时需要注意的是一张表同时只能有两个重分布任务并行执行，并且重分布的表必须有主键，因为在增量同步过程中读取binlog解析增量更新的数据，再根据主键更新到临时表中。

##### 线下阶段（全量部分）

数据重分布线下阶段分为全量部分和增量部分，只有完成全量部分后才能进行增量部分操作。全量部分是对整个表进行全量的备份和全量恢复到临时表中，主要包括四个步骤：

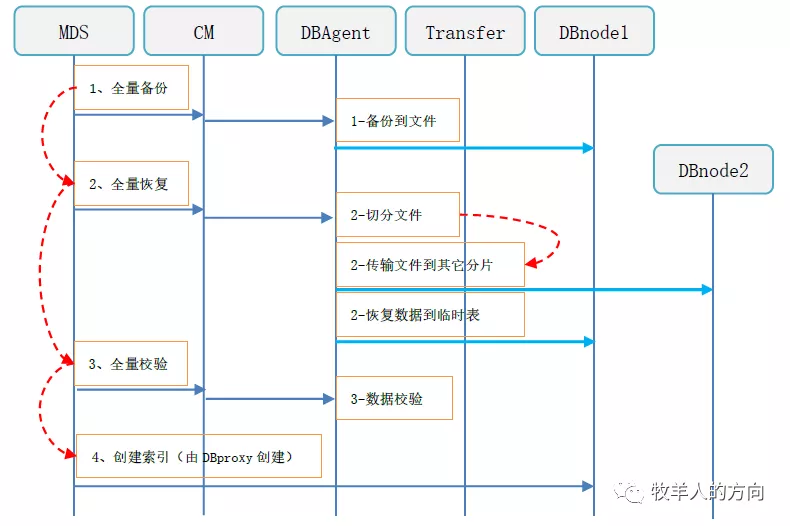
全量备份：使用mysqldump进行全量数据备份

全量恢复：全量恢复是将备份文件根据新的分发策略切分为多个文件传输到其它DB数据节点，并恢复到临时表中

全量校验：进行全量数据校验，如果数据出现不一致，需重头发起重分布任务

创建索引：数据校验完成后会通过DBProxy为临时表创建索引

在数据重分布过程中，MDS会有一个定时器定时检测上述步骤的执行情况，并同步更新保存到RDB中。如果某一步执行失败，会重试该步骤，如果出现数据不一致MDS会重头开始执行任务。重做之前会清理垃圾数据，包括导出的数据文件、拆分的数据文件以及表中已经导入的数据等。



##### 线下阶段（增量部分）

数据重分布线下阶段的增量部分是对全量恢复数据期间的binlog解析回放的过程，通过多次追加binlog直到已经基本追平增量数据为止。



##### 线上阶段

数据重分布线上阶段是对表写禁止，同时完成全部增量的binlog数据追加，并切换临时表和原表的表名。主要分为以下步骤：

表写禁止，此时业务只能查询，不能增删改操作

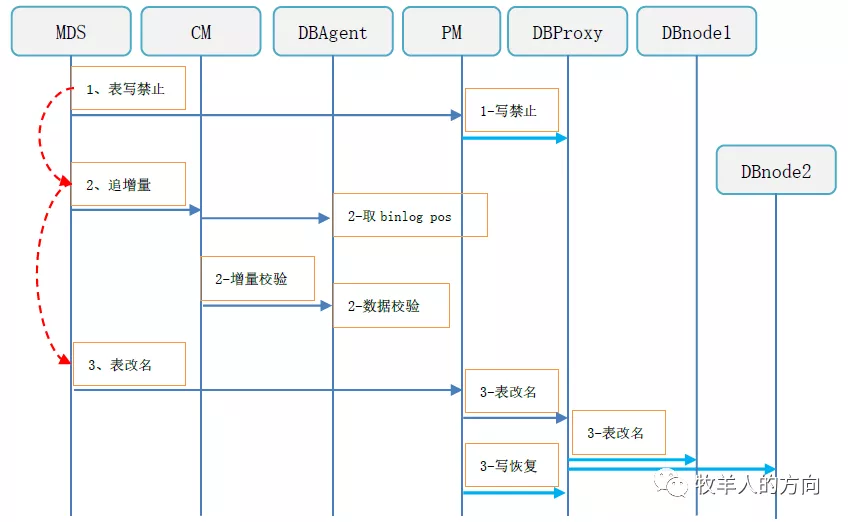
追加全部的binlog数据

增量校验数据的一致性，如果不一致则需要重新开始重分布任务

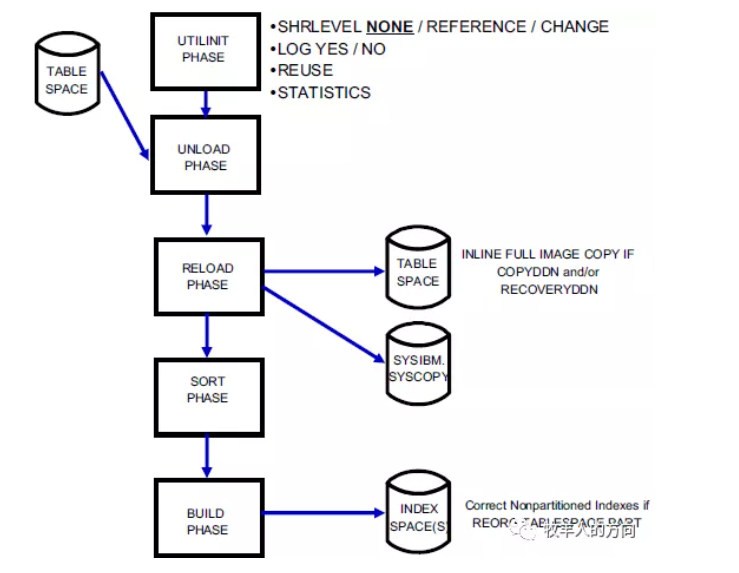
数据校验完成后切换表名

恢复表的写访问，对外提供服务

清理旧表的数据



至此，GoldenDB分片和数据重分布过程到此结束，其实在线数据重分布的过程和DB2的在线重组过程在实现机制上非常相似。



#### 比对规则

range/list/旧hash重分布比对规则：

1. 分发类型改变，则新旧规则中所有分片都需要进行重分布
2. list->list

对新旧规则中当前分片的范围值进行比较，比较规则如下：

1. 首先比对范围值的个数是否相同，范围值就是values i()里的值，个数不相同需要重分布；
2. 范围值的个数一样的，将所有范围值按照解析出来的次序拼接成字符串进行字符比对，相同的话则认为当前分片不需要进行重分布。
3. range->range

比对规则如下：

1. 对新旧规则中当前分片是否使用最大值（maxvalue），使用最大值规则不一致需要重分布；
2. 使用最大值规则一致的话，则比较范围是否相等，不同需要重分布。
3. hash->hash

同range->range，新hash重分布使用bucket。

#### range重分布

#### Hash重分布

桶是新hash表的最小存储单位；

新hash重分布基本原理：

根据新的分发规则，将桶再次平均分布到每个分片上，以一个桶为例：

1. 禁用当前桶，禁用成功所有操作这个桶的语句会失败
2. MDS连接proxy执行搬运桶的语句，即从一个分片搬到另一个分片
3. 广播元数据，把桶的位置从旧分片更新到新分片
4. 解禁当前桶

### 环境准备

1. 检查互信是否配置

在各db用户下执行如下命令，看是否能直接登录到redis用户下：

ssh [redis@xxx.xx.xxx.xxx](mailto:redis@xxx.xx.xxx.xxx)

配置互信的步骤：

1. 以root用户登录DB节点，分别创建redis用户，并设置用户密码：

useradd redis

passwd redis

1. 选择某一DB组件用户，如db1，进行互信配置：

su - db1，将产品包中的huxin.sh脚本上传到db组件用户下。

1. 在用户db1下，新建文件hx\_config，添加如下互信IP地址、用户名和密码信息：

192.168.1.103 redis 密码

192.168.1.103 db1 密码

192.168.1.104 redis 密码

192.168.1.104 db1 密码

192.168.1.105 redis 密码

192.168.1.105 db1 密码

192.168.1.106 redis 密码

192.168.1.106 db1 密码

1. 执行huxin.sh脚本
2. 在db组件用户下执行如下命令，验证互信是否配置正确：

ssh [redis@192.168.1.103](mailto:redis@192.168.1.103)

ssh [redis@192.168.1.104](mailto:redis@193.168.1.104)

ssh redis@192.168.1.105

ssh redis@192.168.1.106

如果可以直接登录到redis用户下，说明互信配置正确。

1. 重分布路径是否配置

在各数据节点服务器上看是否有/home/redisBasePath目录，并赋权777

### 操作步骤

1. 扩充节点

设置新的节点->保存创建重分布任务->提交任务

1. 修改分发键
2. 修改分发规则

### 使用场景

1. 数据扩容

如果需要扩展业务的存储容量，则可以考虑扩展数据节点集群

1. 修改分发规则

比如分发方式由复制方式修改为hash方式

1. 修改分发键

比如一个表的分发键由字段A改为字段B

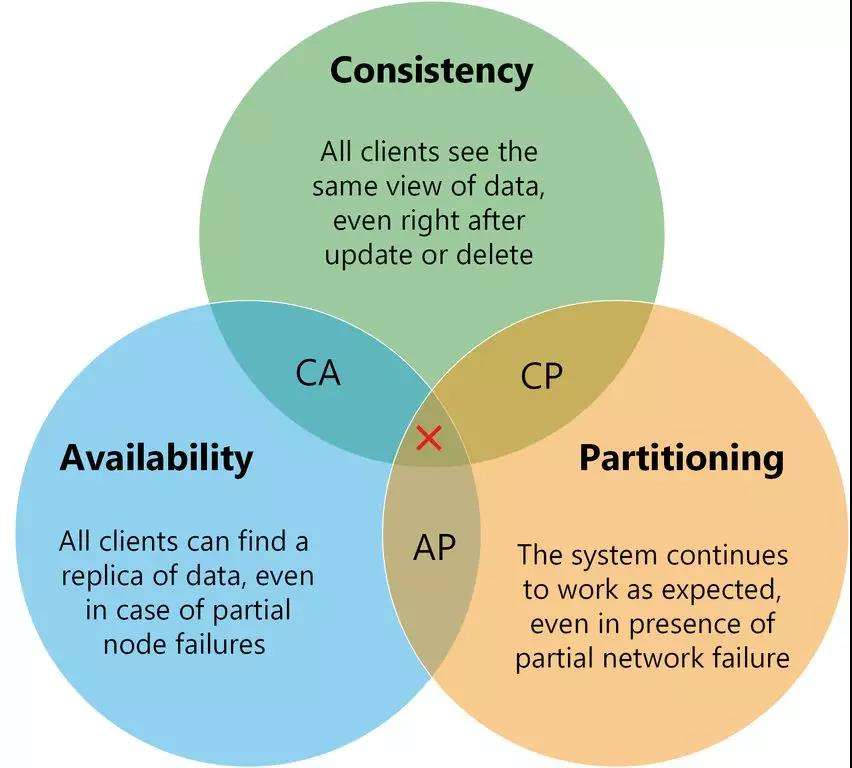
# 分布式事务

分布式事务和单机事务一样，都需要保证事务的ACID四个属性。在分布式环境下，由于网络通信延迟、故障和服务器故障等因素，保证事务一致性尤其具有挑战性。目前业界分布式数据库的数据一致性解决方案为强一致性、弱一致性和最终一致性。GoldenDB实现了强一致性的解决方案，即当用户完成数据更新之后，后续所有读操作都能且只能读到更新以后的值，这是事务一致性的最高级别。

注：强一致性虽然安全性更高，但是也存在性能的问题，即在出现故障时回滚代价非常高。

## 事务理论模型

### CAP理论



分布式数据库的设计遵循CAP理论，即一个分布式系统不能同时满足Consistency（一致性）、Availability（可用性）和Partition Tolerance（分区容忍性）这三个基本需求。

**一致性Consistency**

一致性指的是更新操作成功并返回客户端后，分布式集群中每个节点都返回相同的、最新的数据。对于一致性，可以从客户端和服务端两个不同的视角来看：从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题；从服务端来看，则是更新如何复制到整个分布式系统，以保证数据的最终一致性。在多进程并发访问时，根据获取更新过的数据的不同策略，一致性又分为强一致性、弱一致性和最终一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

**可用性Availability**

可用性指的是系统服务一直可用，在合理的时间内返回读写请求的响应。对于一个可用的分布式系统，衡量系统可用性的时候，一般通过停机时间来计算的。

|  |  |  |
| --- | --- | --- |
| 可用性分类 | 可用水平(%) | 年可容忍停机时间 |
| 容错可用性 | 99.9999 | ＜1min |
| 极高可用性 | 99.999 | ＜5min |
| 具有故障自动恢复能力的可用性 | 99.99 | ＜53min |
| 高可用性 | 99.9 | ＜8.8h |
| 商品可用性 | 99 | ＜83.8min |

通常说金融核心业务系统的可用性水平达到5个9，即即全年停机时间不超过 (1-0.99999)36524\*60 = 5.256 min，这是一个极高的要求。

**分区容忍性Partition Tolerance**

分区容忍性指的是分布式系统在网络分区出现故障时，仍然能够对外提供满足一致性和可用性的服务。在分布式数据库系统中，分区容忍性和扩展性紧密关联，一个好的分区容忍性能够保证当其中几台机器故障或者网络异常后，能够快速的进行故障隔离不影响其它机器的正常运行；也能够保证系统扩展性，节点的新增和删除做到对业务无感知。

**CAP的权衡**

根据CAP理论，无法同时满足一致性、可用性和分区容忍性。但是在分布式数据库系统中，分区容忍性是必须的，分区是始终会存在的，因此需要在一致性和可用性之间进行权衡。

CP without A：分布式系统容许系统停机或者长时间无响应，一旦发生网络故障或者消息丢失等情况，就要牺牲用户的体验，等待所有数据全部一致了之后再让用户访问系统。传统的分布式数据库事务都属于这种模式，对于金融行业的分布式数据库产品而言，优先保证数据的一致性。

AP without C：分布式系统中允许数据不一致，一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

在实际的分布式数据库系统中，基于分片解决扩展性问题并可以实现负载均衡，当某个分片服务不可用时，只会影响部分业务，即服务降级。同时基于多副本构成集群架构，提升系统的高可用。

### ACID理论

ACID指的是数据库系统中事务所具有的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。

原子性：一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性：在事务开始之前和事务结束以后，数据库的完整性限制没有被破坏。

隔离性：当两个或者多个事务并发访问数据库的同一数据时所表现出的相互关系。

持久性：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。

分布式数据库中，将数据库事务的ACID理论延伸到分布式架构下：

|  |  |  |
| --- | --- | --- |
| ACID | 单机数据库 | 分布式数据库 |
| 原子性-A | 多条记录的多次操作要么一起成功，要么一起失败 | 多个数据分片上的操作要么一起成功，要么一起失败 |
| 隔离性-I | 不同连接不会相互访问未提交事务的数据 | 多个计算节点上的不同连接不会相互访问到在多个数据分片内未提交事务的数据 |
| 持久性-D | 事务提交前先将日志落盘，机器重启后不丢失数据 | 事务提交前必须将日志在主节点和从节点都得到复制，主节点故障时从节点仍能找回数据 |

## 分布式一致性算法

在分布式系统中，每一个机器节点虽然都能够明确地知道自己在进行事务操作过程中的结果是成功或失败，但却无法直接获取到其他分布式节点的操作结果。因此，当一个事务操作需要跨越多个分布式节点的时候，为了保持事务处理的ACID特性，就需要引入一个称为"协调者"的组件来统一调度所有分布式节点的执行逻辑，这些被调度的分布式节点则被称为"参与者"。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务真正进行提交。基于以上原则，分布式事务一致性算法主要包括2PC、3PC、Paxos和Raft。



### 2PC

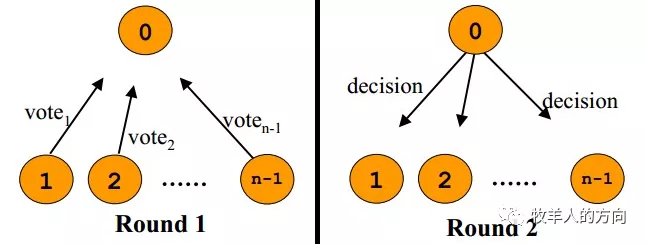
二阶段提交2PC是一致性协议算法，可以保证数据的强一致性，该算法能够解决很多临时性系统故障（包括进程、网络节点、通信等故障），被广泛地使用于关系型数据库系统中。2PC协议中系统分为协调者和参与者，整个过程分为两个阶段：

**阶段1：请求阶段**

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

**阶段2：提交阶段**

在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行响应的操作。



2PC协议的优点是原理简单、实现方便，但也有以下缺点：

同步阻塞：在二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态，也就是说，各个参与者在等待其他参与者响应的过程中，将无法进行其他任何操作。

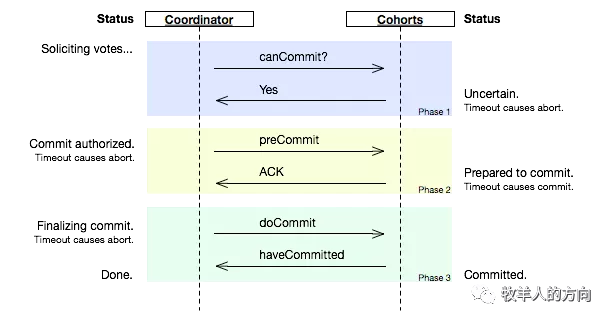
单点问题：协调者的角色在整个二阶段提交协议中起到了非常重要的作用。一旦协调者出现问题，那么整个二阶段提交流程将无法运转，更为严重的是，如果协调者是在阶段二中出现问题的话，那么其他参与者将会一直处于锁定事务资源的状态中，而无法继续完成事务操作。

数据不一致：在阶段二时，当协调者向所有的参与者发送Commit请求之后，发生了局部网络异常或者是协调者尚未发送完Commit请求之前自身发生了崩溃，导致最终只有部分参与者收到了Commit请求。于是，这部分收到了Commit请求的参与者就会进行事务的提交，而其他没有收到Commit请求的参与者则无法进行事务提交，于是整个分布式系统便出现了数据不一致现象。

太过保守：二阶段提交协议没有设计较为完善的容错机制，任何一个节点的失败都会导致整个事务的失败。

### 3PC

3PC协议在协调者和参与者中都引入超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交，包括CanCommit、PreCommit和doCommit三个阶段。



1）CanCommit阶段

3PC的CanCommit阶段和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

2）PreCommit阶段

协调者根据参与者的反应情况来决定是否可以继续事务的PreCommit操作。根据响应情况，有以下两种可能。

2.1 假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会进行事务的预执行：

发送预提交请求。协调者向参与者发送PreCommit请求，并进入Prepared阶段。

事务预提交。参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

响应反馈。如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

2.2 假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就中断事务：

发送中断请求。协调者向所有参与者发送abort请求。

中断事务。参与者收到来自协调者的abort请求之后（或超时之后，仍未收到Cohort的请求），执行事务的中断。

3）DoCommit阶段

该阶段进行真正的事务提交，也可以分为以下两种情况:

**3.1 执行提交**

发送提交请求。协调者接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。

事务提交。参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

响应反馈。事务提交完之后，向协调者发送ACK响应。

完成事务。协调者接收到所有参与者的ACK响应之后，完成事务。

**3.2 中断事务**

协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

3PC协议降低了参与者阻塞范围，并能够在出现单点故障后继续达成一致。缺点是引入 preCommit阶段，在这个阶段如果出现网络分区，协调者无法与参与者正常通信，参与者依然会进行事务提交，造成数据不一致。

### Paxos和Raft

2PC/3PC用于保证多个数据分片上操作的原子性，这些数据分片可能存在于不同的服务器上，2PC/3PC保证这些操作要么全部成功，要么全部失败。Paxos和Raft则是用于保证同一个数据分片的多个数据副本之间的数据一致性。

Paxos算法属于多数派算法，主要解决数据分片的单点问题，目的是让整个集群对某个值的变更达成一致。集群中的任何一个节点都可以提出要修改某个数据的提案，是否通过这个提案取决于这个集群中是否有超过半数的节点同意，所以Paxos算法建议集群中的节点为奇数。

Raft算法是简化版的Paxos，Raft划分成三个子问题：一是Leader Election；二是Log Replication；三是Safety。Raft 定义了三种角色Leader、Follower、Candidate，最开始大家都是Follower，当Follower监听不到Leader，就可以自己成为Candidate，发起投票，选出新的leader。

### 1PC+补偿事务

1PC协议阶段就是从应用程序向数据库发出提交请求到数据库完成提交或回滚之后，将结果返回给应用程序的过程。**一阶段提交不需要“协调者”角色，各结点之间不存在协调操作，因此其事务执行时间比两阶段提交要短，但是提交的“危险期”是每一个事务的实际提交时间，相比于两阶段提交，一阶段提交出现在 “不一致”的概率就变大了**。

在分布式系统中如果存在跨分片的分布式事务，1PC提交并不能保证数据的一致性。像GoldenDB分布式数据库就是通过1PC协议+全局活跃事务ID（GTID）来实现分布式事务。

## 技术难点

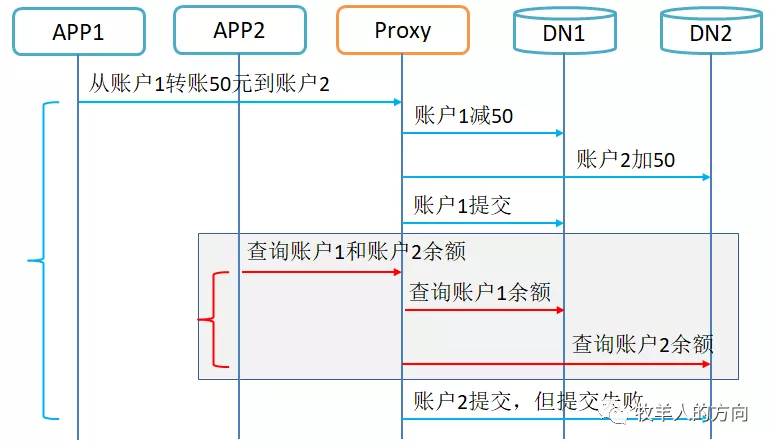
**分布式事务的难点：**

实现分布式强一致性事务的难点在于分布式环境中事务的原子性和隔离性：

部分DB提交失败，如何保证全局事务的原子性

并发访问时，每个事务都不知道其它事务的状态，如何保证事务之间的隔离性

当有部分DB提交成功，部分DB提交失败时，如何保证回滚期间的隔离性



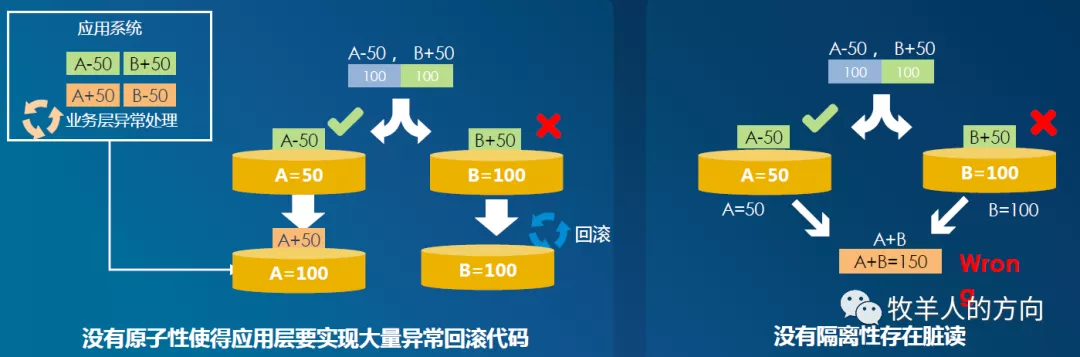
如上，以转账交易为例：

交易前账户1和账户2余额为100元，事务T1从账户1转账50到账户2

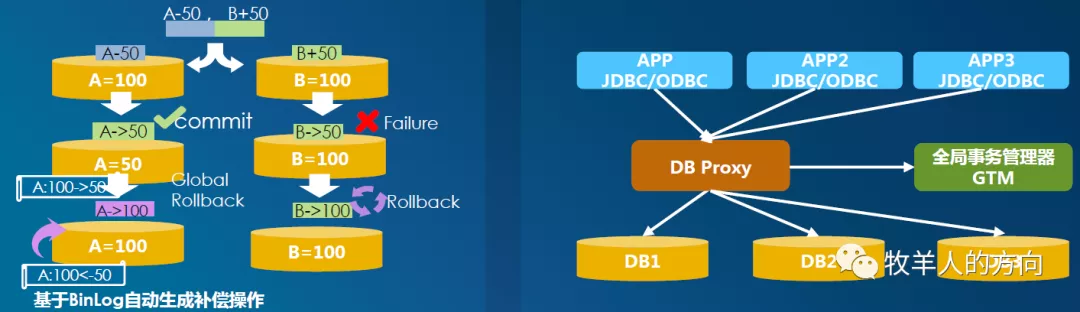
在事务T1提交期间，事务T2读取2个账户的余额，发现两个余额之和是50+100=150元。因为事务之间的隔离性问题产生数据读不一致

可能存在事务T1对账户T1扣钱成功，但是给账户2加钱失败的情况。因为事务内部的原子性问题产生数据写不一致

如果没有分布式数据库系统级别的分布式事务，应用层面解决以上问题需要花费大量工作进行异常回滚代码改造以保证原子性，同时依然会存在因为数据不一致出现的脏读问题。



为解决上述的原子性和隔离性问题，goldendb中基于自动补偿机制实现分布式事务的原子性，引入GTM彻底解决分布式数据库脏读的问题，保证全局事务的一致性。同时分布式事务采用一阶段提交方案，提升系统处理的性能。



## 方案

在GoldenDB分布式事务解决方案中，数据节点作为分布式事务参与者要保证对自身数据库操作的本地事务满足ACID属性，计算节点作为分布式事务协调者协调多个本地事务完成整个分布式事务控制，同时将全局事务状态实时记录在全局事务管理器中。

**核心思想：**

引入全局事务管理器（GTM），记录当前所有未提交的全局事务标识（GTID）及其状态，把全局事务标识（GTID）加到每条数据记录中，作为分布式环境下表的**全局锁**，通过全局锁控制对数据的**并发访问**。

**关键设计：**

1. 事务开始时申请GTID，事务结束时释放GTID，记录在GTM中的所有事务均为未提交事务，成为活跃事务
2. 更新数据时同时更新GTID

### 原子性

GoldenDB分布式事务是借鉴普通事务实现，与普通事务的区别在于GoldenDB会将所有表中增加gtid隐含列，该列在分布式事务中存放该全局事务ID（每个分布式写事务都会得到一个全局唯一且递增的整型值），在执行完所有SQL后，所有DB节点统一完成commit。**如果有DB节点commit提交失败，所有节点的DBAgent通过解析binlog中是否含有该异常事务的全局事务ID来构造反向回滚语句，并向DB下发该反向语句，进行事务回滚，从而保证分布式事务的数据原子性**。

**通过回滚进行事务补偿：**

1. 定位：根据GTID相关信息定位要进行分析日志文件列表
2. 遍历：遍历日志文件，找到GTID对应的事务日志块
3. 生成：分析日志块，为事务中每条SQL语句生成反向SQL语句
4. 执行：将所有反向SQL语句逆序执行，并保证在一个事务中

说明：这种SQL的是逻辑日志操作，这个是可逆的，但是如果是物理日志的变更，则不可逆。这个与undo log是可逆的（SQL），redo log不可逆是一样的道理（物理块变化）。

**提升回滚性能：**

1. 日志预分析：提前对日志文件进行分析，具备GTID索引信息，快速定位待回滚事务在日志中的位置

说明：其实就是提前对GTID-日志的对应关系进行分析，用到的时候直接扫描这个索引文件即可快速定位。

1. 表定义缓存：反向SQL生成过程需要读取表定义，通过缓存查询表定义，可大幅提升反向SQL生成速度
2. Key值利用：生成反向SQL时，使用key值字段作为where条件，可大幅度提升反向SQL语句执行的速度

注：在GTM的log目录下有一个gtid的文件，同时还有一个对应的index索引文件，这个是为了提高性能的优化方案。

在GoldenDB中，事务的原子性通过各分片自动提交和基于binlog的自动补偿机制实现的：

各分片直接提交：各分片并行在本地执行子事务，并在本地各自记录子事务日志。当每个分片都完成后，提交事务全局状态

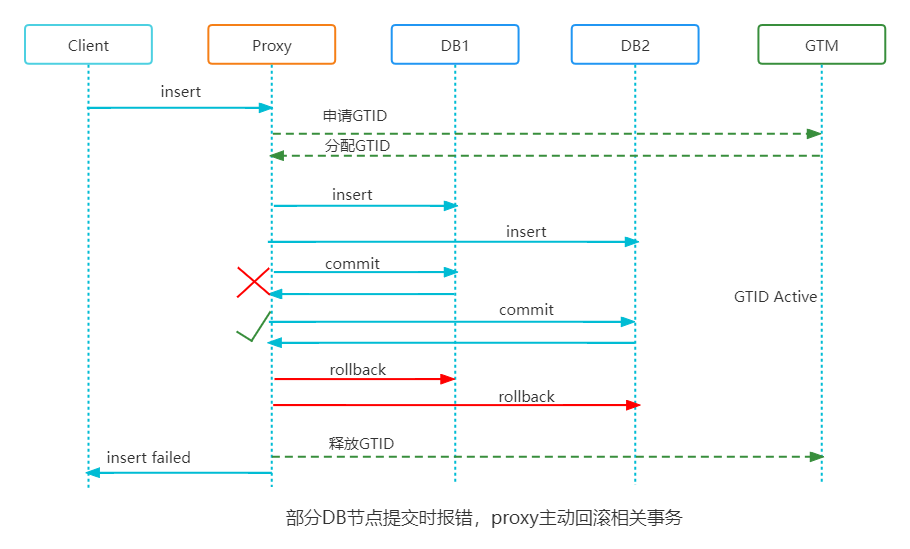
自动补偿机制：某个分片执行失败，其它分片会反向解析本地binlog日志，将数据还原到执行前的状态

对于未提交的子事务，单节点数据库自身会进行回滚；而对于已经提交的子事务，会基于binlog日志自动补偿回滚。这样在应用层无需实现补偿逻辑，而失败回滚是少数情况，整体性能较高。

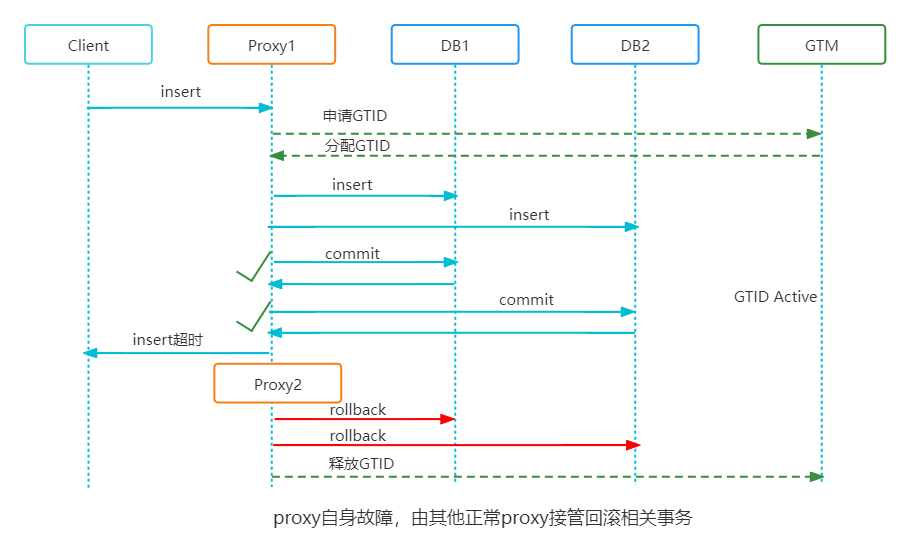
#### 异常事务原子性流程

异常事务原子性处理主要包括两种故障类型：部分DB节点提交时报，Proxy主动回滚相关事务；Proxy自身故障，由其它Proxy回滚相关事务。

##### DB异常



##### Proxy异常



#### ROLLBACK流程

GoldenDB中分布式事务的rollback分为几种情况：

在SQL执行阶段，当部分数据节点的SQL执行失败、执行超时或者计算节点异常，此时需要业务根据SQL的执行结果选择rollback或者选择其它分支继续执行其它SQL

在事务提交阶段，当部分数据节点的SQL执行失败、执行超时或者计算节点异常，需要在已经提交的DB上进行已提交事务的回滚

图形用户界面

描述已自动生成

Rollback处理过程如上图所示：

1、DBProxy将GTID发送给DB节点上部署的事务回滚组件DBagent

2、DBAgent解析该事务的Binlog，然后对数据进行回滚

3、当所有DB分片回滚完成后，再释放GTID

通过回滚进行事务补偿分为定位、遍历、生成和执行四个阶段：

定位：根据GTID相关信息定位要进行分析binlog日志文件的列表

遍历：遍历binlog日志文件，找到GTID对应的事务日志块

生成：分析日志块，为事务中每条SQL语句生成反向的SQL语句

执行：将所有反向SQL语句逆序执行，并保证在一个事务中

**在GoldenDB中还会利用binlog预分析、表定义缓存、key值利用等技术手段提升回滚性能**。

### 隔离性

**总体原理：**

结合数据中GTID信息及GTM中系统当前未提交事务列表，判断该数据所在事务是否已经全局提交，解决读写冲突和写写冲突。

冲突判断：数据的GTID要求不在GTM上的未提交事务列表中

预封锁机制：数据的GTID要求小于系统当前分配的最大GTID（GTM上的活跃事务列表和数据不再一个系统里，查询两者的数据无法保障严格一致性）

重试：上述条件不满足时触发重试，确保一致性要求

在读写操作前查看操作的数据是否处于活跃状态，如果是活跃状态，则等待事务结束后，在进行读写操作，这样的优势是有全局锁就能够彻底解决分布式环境下的脏读问题。

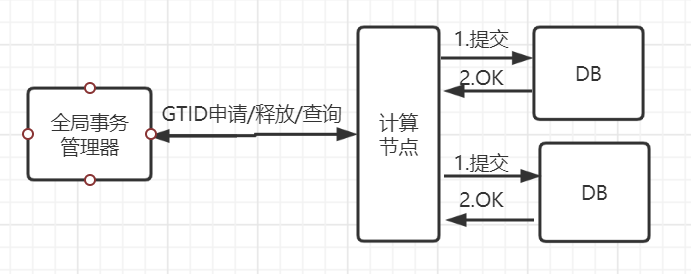
这样做是会牺牲一部分代价，在高并发情况下，会牺牲一部分读性能来保证数据的一致性，目前已经实现分布式MVCC，可以大幅度降低锁的范围。

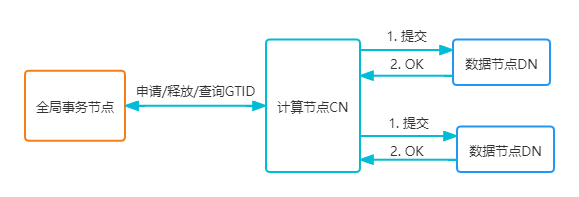
### 一致性

引入GTM实现全局锁，彻底解决分布式数据库脏读问题，保证全局事务的一致性。

### 架构图

基于GTID的分布式事务管理方案架构图：





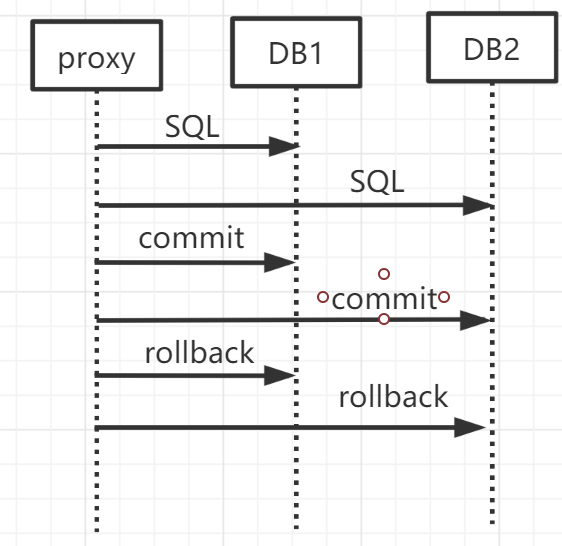
采用全局事务ID对分布式事务进行控制，其核心思想是全局事务控制和标签数据。前者为每一个分布式写事务分配一个全局唯一的有序事务ID（GTID），并根据事物的存活情况维护对应ID的生命周期。后者在用户表中增加对应用透明的GTID列，并在该列中维护操作本行数据的最近一次分布式事务对应的全局事务ID。

总体在流程交互上为一阶段提交+补偿事务（2PC有一个达成共识的过程，但是这里没有统一的共识，即直接通过语句中的GTID判断了）的方式，如果事务在提交阶段有部分节点失败，本方案将采用回滚已成功提交的事务。

## 设计

### 一致性设计

GoldenDB分布式数据库利用数据库普通事务来代替分布式事务，在所有的表中增加隐含列，存放该全局事务ID。在执行完所有的SQL后统一进行commit，各个DB节点的隔离级别需要设置为RC（读已提交）。



create table t1(col1 int,col2 varchar(20));

proxy转化为：create table t1(col1 int,col2 varchar(20),gtid int);

insert into t1 values(xx,’xx’)

proxy转化为：insert into t1 values(xx,’xx’,current\_gtid);

update table t1 set col1=yyy

proxy转化为：update table t1 set col1=yyy,gtid=current\_gtid;

select col1 from t1;

proxy转化为：select col1,gtid from t1;

说明：在创建表中增加隐含列gtid，每一个分布式事务分配一个全局唯一的事务ID，插入或者更新操作会更新gtid列为本地事务ID；为了保证数据一致性读，需要插入GTID列。

### 可靠性设计

#### 与GTM交互异常

处理过程中GTM故障，该如何处理？

1. 每个GTID创建和释放都有持久化日志
2. GTM由一主多备节点构成高可靠集群
3. 由MDS节点发起主备切换，并通知到所有的计算节点

如果是网络异常导致的链路闪断、消息丢失，定时器超时后会重试，超时时间和尝试次数可配，在超时尝试时间内网络恢复可以保证事务不失败。

GTM采用双机双活高可用机制，备机异常不影响业务，主机异常后，HA主备切换，Proxy会定时尝试备机一定次数，尝试时间间隔和尝试次数可配，尝试间隔\*尝试次数>HA切换时间可以保证事务不失败。

如果GTM双机异常，为整个分布式集群的数据强一致性，需将GTM持久化数据里的活跃事务列表及比最大GTID还大的所有事务全部回滚掉。

#### 部分节点提交失败数据一致性保证

当所有SQL执行结束后，proxy再向各个DB发起commit操作，此时如果有的节点提交失败有的提交成功，会造成数据不一致的情况；这种情况下proxy触发提交成功节点的DBAgent进行已提交事务回滚，回滚成功后才能向GTM释放GTID。

#### DB节点异常数据一致性保证

DB节点发生crash时，DB进行主备切换，在新主完成binlog replay的这个时间段该DB节点无法对外提供服务。对于DB异常有如下三种场景需要考虑：

1. 所有涉及该DB节点的新事务无法执行；
2. 所有涉及该DB节点已经执行还未提交的事务会给本事务涉及的DB节点发起回滚；
3. 事务正在提交节点DB异常，提交成功的节点会触发已提交事务回滚；

DB长时间无响应，OS会检测链路长时间无消息会自动断链，默认是7小时。

#### Proxy节点异常数据一致性保证

分布式事务正常运行时，proxy发生异常crash，此时proxy与所有db集群和客户端连接都会中断，客户端可以通过与集群下其他proxy建立连接进行数据库集群访问，不会影响发起的业务；但是未结束的残留事务有如下两种场景需要考虑：

1. DB上未提交的事务，由于与数据节点断链，MySQL会自动回滚；
2. Proxy异常宕机时，部分DB节点提交成功，部分节点未提交，由于GTID不会释放，其他事务访问异常数据会挂住，知道这些数据被回滚掉。

### 高性能设计

GTM作为一个独立的网元，用于分配全局事务ID，提供查询和释放功能，GTM的处理单次操作的耗时随单位时间内请求数增加而增加，并且存在性能上限。所以减少proxy频繁向GTM请求成为分布式数据库性能瓶颈的关键，目前方案是采用组申请、查询、释放。

GTM高性能机制：

1. 批量申请GTID

问题：每个事务都要申请一次GTID，与GTM交互频繁。

性能提升机制：计算节点批量申请GTID，GTM一次执行多个计算节点的批量申请。

1. 批量释放GTID

问题：每个事务都要释放GTID，与GTM交互频繁。

性能提升机制：计算节点批量释放GTID，GTM一次执行多个计算节点的批量释放。

1. 批量查询GTID

问题：每个事务都要查询活跃事务列表，与GTM交互频繁。

性能提升机制：计算节点汇总多次查询，GTM一次执行多个计算节点的汇总查询。

1. GTM横向扩展

问题：多集群共用GTM，导致GTM压力大。

性能提升机制：支持多GTM部署，最多一个集群独占一套GTM。

效果：减少计算节点与GTM交互次数（RPC通信），减少GTM日志落盘次数、主从复制次数。

## 隔离级别

### 单机事务隔离级别

在标准SQL规范中，定义了4个事务隔离级别，不同的隔离级别对事务的处理不同：

Read Uncommitted（读未提交内容）：

在该隔离级别下，所有事务都可以看到其他未提交事务的执行结果，本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读未提交的数据，也被称之为脏读（dirty read）。

Read Committed（读已提交内容）：

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的隔离级别），它满足了隔离的简单定义：一个事务只能看到已经提交事务所做的修改。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一个事务的其他实例在该实例处理期间可能会有新的commit，所以同一个select可能会返回不同结果。

Repeatable Read（可重复读）：

这是MySQL的默认隔离级别，它确保同一个事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致一个棘手的问题：幻读（Phantom Read）。简单地说，幻读指当前用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影”行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC）机制解决了该问题（因为对于RR隔离级别，MySQL采用InnoDB或Falcon出处引擎的时候就不会存在幻读，但是其他存储引擎仍然存在幻读）。

Serializable（可串行化）：

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

### 分布式事务隔离级别

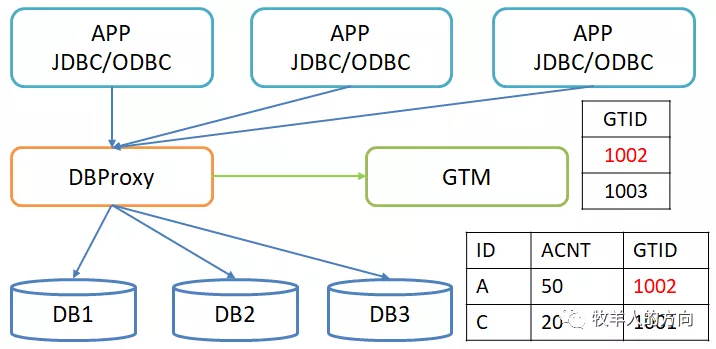
#### 概述

在实时一致分布式事务处理中，GoldenDB引入了全局事务管理器GTM，记录当前所有正在执行的全局事务标识GTID及其状态，并把GTID附加到每条数据记录中（DB中的表会增加GTID字段），作为分布式环境下的全局标志。通过全局事务机制实现全局事务的隔离性，并协助实现全局事务的原子性。

分布式事务更新：分布式事务更新数据时，Proxy会从GTM申请GTID，同时改写SQL语句，将申请的GTID更新到数据行中。

分布式一致性读过程：当事务隔离级别为CR时，查询事务会通过proxy查询GTID中当前的活跃事务列表，并进行活跃状态校验，如果该数据正在被修改则不会返回。

1）实时一致性事务的处理



事务开始时申请GTID，事务结束时释放GTID，所有活跃事务由GTM统一管理，称为活跃事务列表。

建表时会在表上自动增加GTID列（对应用不可见），更新数据行时同时会更新GTID列（为了提升性能，单次操作可以批量申请一批GTID）。如果数据行的GTID在活跃事务列表中，表明该数据正在被其它事务修改。

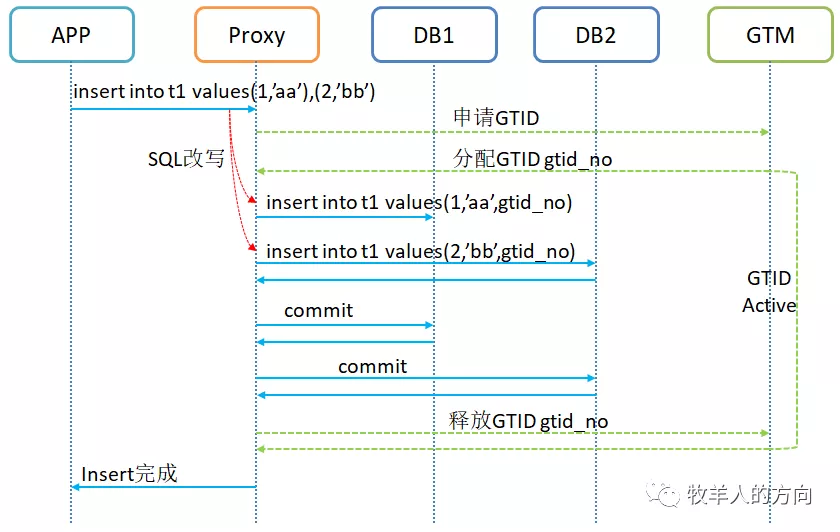
proxy计算节点会对事务的SQL语句进行修改，将GTID加入到语句中下推到数据节点。

GTID单调递增。

2）分布式事务更新流程

分布式事务在更新数据的时候，会改写SQL语句，将申请的GTID更新到数据行的GTID列中。比如：

insert into t1 values(1,’aa’),(2,’bb’);  
 语句会改写为：  
 insert into t1 values(1,’aa’,gtid\_no),(2,’bb’,gtid\_no);



事务一旦提交，GTID就会与数据一起写到DB的binlog日志文件中。

#### 分类

GoldenDB中数据节点的隔离级别是默认RC隔离级别，计算节点proxy分为读语句和写语句不同的隔离级别，如下所示：

|  |  |  |
| --- | --- | --- |
| 分类 | 隔离级别 | 描述 |
| 读语句隔离级别 | UR(Unconsistency Read) | 非一致性读，即不判断分布式读写冲突，适用于允许脏读或者不存在读写冲突的场景 |
| CR(Consistency Read) | 强一致性读，先查询活跃GTID，后查询数据，严格保证返回结果处于分布式事务已提交状态，不存在脏读的可能性 |
| 写语句隔离级别 | SW(Single Write) | 单事务写，即不判断分布式写-写冲突，适用于不存在多个事务同时写相同数据的场景 |
| CW(Consistency Write) | 强一致性写，需要判断事务写-写冲突，允许多个事务同时写相同的数据 |

事务执行的时候会先经过计算节点根据隔离级别判断并改写语句，然后下发到数据节点执行。

GoldenDB的存储节点层，还是遵守单机DB的事务隔离级别，GoldenDB在计算节点层增加了分布式事务的隔离级别，如下：

**UR（脏读）：**

查询时，查询的数据中允许包含脏数据，不校验来自各个存储节点的数据是否活跃，即不检查来自各个存储节点的数据是否来自同一个版本或同一时刻的副本数据。

**CR（强一致性读）：**

查询时，读取的数据中不允许包含脏数据，检验来自各存储节点的数据不能为活跃状态，即检查来自各个存储节点的数据必须为同一版本或同一时刻的副本数据。

**SW（单节点写）：**

不考虑分布式事务下的对部分已提交事务的数据的修改，即不检查修改的数据是否活跃。

下面有两种情况使用SW：

1. 修改的数据不会在两个及以上的事务中同时修改
2. 涉及本数据的所有事务都只会修改单条记录

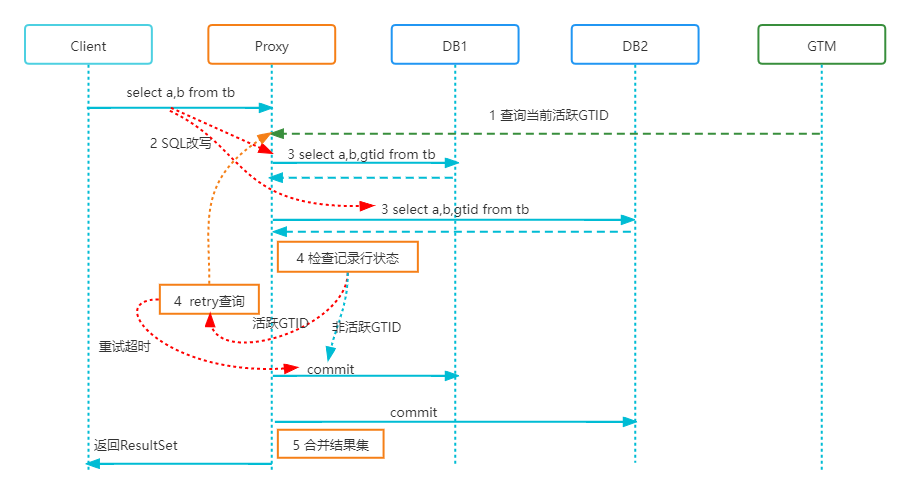
**CW（强一致性写）：**

需要考虑分布式事务下的对部分已提交事务的数据的修改，需要校验修改的数据是否处于活跃状态，活跃状态的数据不能修改，只能修改已经全局提交成功的数据。

#### 原理

##### 一致性读

当proxy隔离级别定义为CR时候，查询数据时，通过检查数据行GTID列对应的全局状态，来判断该数据行是否正在被其它全局事务修改。如果GTID在全局活跃事务列表中，则表明该数据正在被修改，不能返回给应用。在一致性读的过程中，如果事务已提交即GTID不在活跃事务列表中，则返回的是已提交的数据；如果事务未提交，即GTID在活跃事务列表中，则返回的是事务提交之前的数据，这样即满足了隔离性要求。



一致性读处理过程如下：

1、向GTM请求查询当前的活跃事务列表

2、DBProxy将查询的SQL进行改写

原语句select a,b from t1 where改写为

select a,b,gtid from t1 where

3、将改写后的SQL下发到数据节点进行查询

4、DBproxy根据查询后的表数据行的GTID和GTM查询的GTID进行活跃判断，如果GTID不在活跃事务列表中，则合并结果集返回给应用；如果在GTID活跃事务列表中，会进行尝试重新查询活跃事务列表GTID进行判断，超过重试次数后也会返回结果集，不过返回的是更新前的数据

5、由于数据节点的隔离级别为RC模式，下发到数据节点的SQL语句返回的是事务更新提交之前的数据

6、在dbproxy节点进行GTID判活及retry的时候，会有短暂的阻塞读的情况，时间会很短，大概是ms级别

备注：在一致性读过程中，先获取活跃事务列表，再去查询数据行，是因为如果先查询数据行，当查询期间其它事务正在修改数据并且在获取事务列表之前提交了，仍然会返回这些事务未提交的数据。同时要结合GTID单调递增的特性，来判断是否存在活跃事务列表之外更大的活跃事务，也就是看数据行中的GTID是不是比GTM查到的最大活跃事务列表还大。

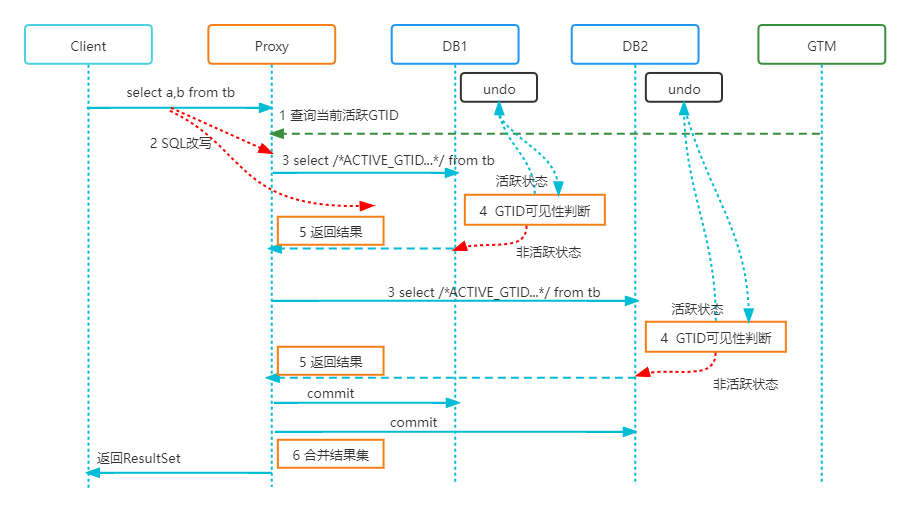
##### MVCC模式下的一致性读

在MVCC模式下，将全局活跃事务列表带到DB层进行活跃判断，判断规则如下：

DB节点使用proxy上收到的active\_gtm\_gtid\_list进行数据活跃判断

如果当前数据行上的GTID判断不活跃，直接返回数据

如果当前数据行上的GTID判断活跃，则通过undo构造全局的镜像数据



MVCC模式下一致性读的处理过程如上图所示：

1、DBproxy向GTM请求当前的活跃事务列表

2、DBProxy将原SQL进行改写，即将活跃事务列表下推到数据节点

将select a,b from t1 where改写为

select /\*+SET\_VAR(active\_gtm\_gtid\_list=活跃事务列表的BASE64编码) SET\_VAR(next\_gtm\_gtid=ulonglong变量)\*/ a,b,gtid from t1 where

3、将改写后的SQL语句和GTID活跃事务列表下发到数据节点

4、在计算节点判断对应的GTID是否全局可见

5、如果GTID不活跃，则直接返回数据；如果GTID活跃，则通过undo数据构造全局的前项数据返回

6、DBproxy将结果集汇总后返回给应用

MVCC模式下的一致性读的好处是通过MVCC多版本并发控制，能够**保证读一致性，并且写不阻塞读**。同时将GTID的活跃判断检测下推到了数据节点，**避免了计算节点获取所有结果集进行活跃判断而导致的Proxy内存开销增大的风险**。但是目前GoldenDB的MVCC功能没有经过测试验证，功能和性能上的效果如何有待进一步测试。

在分布式环境、计算存储分离架构下，事务处理技术如果采用MVCC，那么MVCC应该置于哪里？在解耦的需求下，可以考虑如下方式：

1. MVCC置于存储层

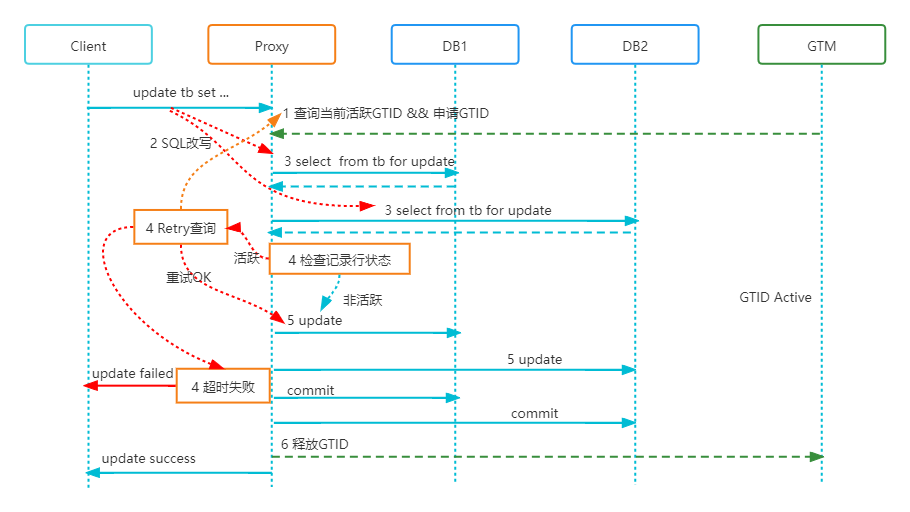
在传统的页面结构之上，进行MVCC的可见性判断（即可见性判断放置在存储层），这样的好处是，网络传输量少，但是事务处理技术和存储层耦合度高。

1. MVCC置于计算层

这样以MVCC技术为基础的事务处理可以和存储层解耦，这有助于实现多模数据库。

##### 一致性写

GoldenDB中分布式事务在更新或删除数据的时候，会通过select for update锁定+检查GTID活跃来解决写写冲突。



一致性写处理过程如上图所示：

1、DBProxy计算节点请求活跃事务列表GTID，同时请求创建GTID

2、DBproxy将update更新语句进行改写

将update set a=’xxx’改写为

select pkey,gtid from t1 for update和update t1 set a=’xxx’,gtid=gtid\_no

注：这个是悲观锁的控制逻辑，如果是乐观锁，则下发语句为update ... where ... and gtid not in (活跃事务列表) and gtid < 最大gtid;

3、将select for update语句下发到数据节点，并返回结果，select for update会申请加锁，如果出现等锁超时则直接失败

4、DBProxy对返回的结果进行GTID活跃检查，如果是非活跃GTID，则可以直接下发update语句；如果是活跃GTID，则尝试重新查询活跃GTID列表，检查OK后可以继续update，如果超时则失败

5、GTID活跃检查OK以后，将update语句下发到数据节点执行，update语句会更新gtid字段，当前时段该数据行的记录是被该事务更新，GTID为活跃状态

6、所有节点更新完成并提交后，释放GTID

7、DBProxy再将事务处理的结果返回给应用端

注：这里之所以有一个重试retry的过程，原因在于GoldenDB这种事务控制模型是基于这样一种假设的，就是计算节点发送给数据节点SQL之后，数据节点提交失败的情况概率比较低。所以，在遇到记录是活跃状态的时候，节点会重试一下，保证尽可能都会执行成功。

#### 总结

总结GoldenDB中的分布式事务，为了解决分布式数据库中事务的原子性和隔离性问题，引入了binlog自动补偿机制以保证原子性、GTM全局事务管理器以保证事务的隔离性，实现一致性读和一致性写。接下去中，GTM的性能如何保证，MVCC功能下的稳定性和性能，DB故障、DBProxy故障和GTM时候分布式事务的处理机制等，都是分布式事务使用过程中需要考虑的问题。

## 实现流程

涉及多个DB节点操作的事务为分布式事务。

1. 涉及多个DB节点写操作的事务，需要向GTM申请GTID
2. Proxy判断到仅涉及一个DB节点的写事务（非交互式事务），不用向GTM申请GTID
3. 交互式事务，即使事务仅涉及一个DB节点，proxy默认也会申请GTID，除非事务中带有hint（samedb），强制指定整个事务都下发到一个节点执行
4. 是否向GTM申请GTID，主要看proxy是否能够判断该事务是否只涉及一个DB节点，如果判断只涉及一个DB节点，不需要申请GTID，否则向GTM申请GTID
5. CR、CW隔离级别时，与事务涉及的DB节点数无关，都会检验数据是否处于活跃状态

### INSERT流程

|  |  |  |
| --- | --- | --- |
| INSERT语句 | 单节点 | 多节点 |
| SW | 不申请GTID | 申请GTID |
| CW | 不申请GTID | 申请GTID |

原语句：insert into t1 values(xx,’xxx’),(yy,’yyy’);

转换后：insert into t1 values(xx,’xxx’,current\_gtid),(yy,’yyy’,current\_gtid);

流程说明：

1. 客户端发起insert流程
2. Proxy向GTM申请全局事务ID（create gtid请求）
3. Proxy从GTM接收到全局事务ID响应
4. Proxy并行发送insert给相关DB，语句携带GTID
5. Proxy接收到各个DB语句执行结果
6. Proxy判断各DB insert执行成功，并行发送commit给相关DB（如果有DB执行insert失败或超时，proxy返回给客户端失败或超时消息，待客户端执行rollback命令回退事务）
7. Proxy接收到各个DB执行commit
8. Proxy判断所有commit都执行OK后，通知GTM释放该全局事务ID（如果有commit失败，proxy发起已提交事务回滚流程）
9. Proxy接收到GTM释放GTID结果
10. Proxy判断GTM释放GTID成功，给客户端回复执行成功结果（如果GTM释放GTID失败，给客户端回复执行失败结果）

注：如果能够保证当前事务仅含有insert，且只涉及一个节点，不需要申请GTID，走普通insert流程。

### UPDATE流程

update单节点不申请GTID是因为不存在已提交事务的回滚情况，由mysql自己回滚：

|  |  |  |
| --- | --- | --- |
| UPDATE | 单节点 | 多节点 |
| SW | 不申请GTID  不需要select for update | 申请GTID  不需要select for update |
| CW | 不申请GTID  需要select for update | 申请GTID  需要select for update |

update转换场景1：

原语句：update t1 set a=’xxx’ where b>’yyyy’;

转换后：

select gtid\_col,primary\_key from t1 where b>’yyyy’ for update;

update t1 set a=’xxx’,gtid=current\_gtid where primary\_key1 or primary\_key2;

（没有开启乐观锁开关，即proxy.ini配置文件中optlock\_switch=0）

流程说明：

1. 客户端发起update流程
2. Proxy向GTM申请全局事务ID，并获取当前的所有活跃事务列表请求
3. Proxy接收到申请的新GTID和当前活跃GTID列表及当前GTM中最大GTID值
4. Proxy向相关DB执行select for update，查询update数据的gtid列值，并锁住update涉及到的相关行资源
5. Proxy接收到select for update查询结果和加锁情况
6. Proxy判断select返回的gtid（不含本事务新申请的gtid）不存在步骤3的活跃事务列表，且都不大于当前GTM中最大gtid，即没有活跃，向各个DB节点下发update t1 set a=’xxx’,gtid=current\_gtid where primary\_key1 or primary\_key2;（如果select返回的gtid（不含本事务新申请的gtid）存在于3步骤3的活跃事务列表中，或大于当前GTM中最大gtid，重复步骤2~5（重复次数受配置值限制，重复时步骤2只用getActiveGtidList，不再create gtid））
7. Proxy接收各个节点update执行结果
8. Proxy判断各个DB节点的update执行结果成功，向各个DB节点下发commit语句（如果有节点update执行失败或超时，向客户端发送失败或超时消息，待客户端执行rollback命令，回退事务）
9. Proxy接收各个DB节点的commit结果
10. Proxy判断各个DB节点的commit都成功，通知GTM释放全局事务ID（如果有DB节点commit失败或超时，发起已提交事务回滚）
11. Proxy接收GTM释放gtid结果
12. Proxy判断GTM释放gtid成功，给客户端回复执行成功结果（如果GTM释放gtid失败，给客户端回复执行失败结果）

Update转换场景2：

原语句：

update t1 set a=’xxx’ where b=’yyyy’（b为主键和分发键）

转换后：

update t1 set a=’xxx’ where b=’yyyy’ and gtid not in (当前活跃列表) and gtid<=当前最大gtid;

（原语句为where条件中含有主键和分发键的等值条件，开启乐观锁开关，即proxy.ini配置文件中optlock\_swtich=1）

流程说明：

1. 客户端发起update流程
2. Proxy通过where条件判断含有主键和分发键的等值条件，结果集落在一个节点，然后向GTM发起查询活跃gtid列表和当前最大gtid请求
3. Proxy接收查询响应消息
4. Proxy向DB节点下发的语句为：update t1 set a=’xx’ where b=’yyyy’ and gtid not in (当前活跃列表) and gtid<=当前最大gtid
5. Proxy接收update结果，如果Rows matched>0，proxy执行步骤8，即返回update结果给客户端；如果Rows matched=0，proxy执行步骤6
6. Proxy下发check select(select count(1) from t1 where b=’yyyy’)，检查是否存在b=’yyyy’的行
7. Proxy接收到check select的结果，如果check select检验到结果大于0，重复步骤4、5，重复次数可以配置，如果重试期间步骤5的Rows matched>0，执行步骤8，超过重试次数，返回给客户端数据处于活跃状态；如果check select检验到结果=0，执行步骤8，即返回命中条数为0的结果集给客户端

注：

1. select for update是为了加锁和检验活跃gtid
2. Proxy如果本事务仅涉及一个节点，可以不生成GTID，但是select for update仍然可能要实施

### DELETE流程

|  |  |  |
| --- | --- | --- |
| DELETE | 单节点 | 多节点 |
| SW | 不申请GTID  不需要select for update | 不申请GTID  不需要select for update |
| CW | 申请GTID  需要select for update | 申请GTID  需要select for update |

Delete（CW）转换场景1：

原语句：

delete t1 where b>’yyyy’;

转换后：

select gtid\_col,primary\_key from t1 where b>’yyyy’ for update;

update t1 set gtid=current\_gtid where primary\_key1 or primary\_key2;

delete from t1 where primary\_key1 or primary\_key2;

流程说明：

1. 客户端发起delete流程
2. Proxy向GTM申请全局事务ID，并获取当前的所有活跃事务列表请求
3. Proxy接收到申请的新GTID和当前活跃GTID列表及当前GTM中最大GTID值
4. Proxy向相关DB执行select for update，查询update数据的gtid列值，并锁住update涉及到的相关行资源
5. Proxy接收到select for update查询结果和加锁情况
6. Proxy判断select返回的gtid（不含本事务新申请的gtid）不存在步骤3的活跃事务列表，且都不大于当前GTM中最大gtid，即没有活跃，向各个DB节点下发update t1 set gtid=current\_gtid where primary\_key1 or primary\_key2;（如果select返回的gtid（不含本事务新申请的gtid）存在于3步骤3的活跃事务列表中，或大于当前GTM中最大gtid，重复步骤2~5（重复次数受配置值限制，重复时步骤2只用getActiveGtidList，不再create gtid））
7. Proxy接收各个节点update执行结果
8. Proxy判断各个DB节点的update执行结果成功，向各个DB节点下发delete语句（如果有节点update执行失败或超时，向客户端发送失败或超时消息，待客户端执行rollback命令，回退事务）
9. Proxy接收各个DB节点的delete结果
10. Proxy判断各个DB节点的delete都成功，向各个DB下发commit语句（如果有DB节点update失败或超时，向客户端发送失败或超时消息，待客户端执行rollback命令，回退事务）
11. Proxy接收各个DB节点的commit结果
12. Proxy判断各个DB节点的commit执行结果成功，通知GTM释放全局事务ID（如果有的节点commit失败或超时，发起已提交事务回滚流程）
13. Proxy接收GTM释放gtid结果
14. Proxy判断GTM释放gtid成功，给客户端回复执行成功结果（如果GTM释放gtid失败，给客户端回复执行失败结果）

delete转换场景2：

原语句：

delete t1 where b=’yyyy’（b为主键和分发键）

转换后：

delete t1 where b=’yyyy’ and gtid not in (当前活跃列表) and gtid<=当前最大gtid;

（原语句为where条件中含有主键和分发键的等值条件，开启乐观锁开关，即proxy.ini配置文件中optlock\_swatich=1）

流程说明：

1. 客户端发起delete流程
2. Proxy通过where条件判断含有主键和分发键的等值条件，结果集落在一个节点，然后向GTM发起查询活跃gtid列表和当前最大gtid请求
3. Proxy接收查询响应消息
4. Proxy向DB节点下发的语句为：delete t1 where b=’yyyy’ and gtid not in (当前活跃列表) and gtid<=当前最大gtid
5. Proxy接收delete结果，如果Rows >0，proxy执行步骤8，即返回delete结果给客户端；如果Rows =0，proxy执行步骤6
6. Proxy下发check select(select count(1) from t1 where b=’yyyy’)，检查是否存在b=’yyyy’的行
7. Proxy接收到check select的结果，如果check select检验到结果大于0，重复步骤4、5，重复次数可以配置，如果重试期间步骤5的Rows matched>0，执行步骤8，超过重试次数，返回给客户端数据处于活跃状态；如果check select检验到结果=0，执行步骤8，即返回命中条数为0的结果集给客户端

注：

1. select for update是为了加锁和检验活跃gtid
2. Proxy如果本事务仅涉及一个节点，可以不生成GTID，但是select for update仍然可能要实施

### SELECT流程

读事务的控制：

|  |  |  |
| --- | --- | --- |
| SELECT | 单节点 | 多节点 |
| UR | 不查询GTID  不查询活跃事务 | 不查询GTID  不查询活跃事务 |
| SEMI-CR | 查询GTID  查询活跃事务 | 查询GTID  查询活跃事务 |
| CR | 查询GTID  查询活跃事务 | 查询GTID  查询活跃事务 |

原语句：select a,b from t1 where b>’yyyy’;

转换后：select a,b,gtid from t1 where b>’yyyy’;

流程说明：

1. 客户端发起select流程
2. Proxy先向GTM获取当前的所有活跃事务列表
3. Proxy在接收到GTM响应后，发送select给相关DB
4. Proxy判断返回DB中GTID是否处于GTM活跃gtid列表中，或大于活跃gtid列表中的最大值，如果有活跃事务，重复步骤2、3（重复次数限制），否则转下一步
5. 事务结束后，proxy给客户端回复执行结果

注：

1. 在UR隔离级别下，不需要实施和GTID相关的所有操作
2. 判断事务是否活跃，除了在列表中的事务外，还包含比最大事务ID大的所有事务
3. 先向GTM查询GTID列表完成之后才能查询DB

### 汇聚函数

|  |  |  |
| --- | --- | --- |
| 汇聚函数的控制 | 单节点 | 多节点 |
| UR | 不需要lock in share mode | 不需要lock in share mode |
| SEMI-CR | 不需要lock in share mode | 需要lock in share mode |
| CR | 不需要lock in share mode | 需要lock in share mode |

注：汇聚函数是无法保证强一致性的，这主要是为了防止select for update加锁扩大导致全表锁，所以汇聚函数采用共享锁lock in share mode，这就导致会出现多个同时访问一组数据，并进行修改，这个是最终一致性而不是强一致性。

原语句：select count(\*) from t1 where b>’yyyy’;

转换后：select count(\*) from t1 where b>’yyyy’ lock in share mode;

流程说明：

1. 客户端发起select count(\*)流程
2. Proxy并行发送select in share mode给相关DB
3. 事务结束后，proxy给客户端回复OK响应

注意：

1. 在UR隔离级别下，不需要转换成lock in shared mode
2. 如果判断出select仅仅涉及单个DB节点，不需要转换成lock in share mode
3. 汇聚函数CR隔离级别的查询结果不是强一致性的，在有事务并发的情况下，可能会得到不一致性的结果

注：汇聚函数为什么要用lock in share mode，而不用for update，这是为了降低加锁范围，因为汇聚函数大部分都是全表扫描，使用for update就会全表锁，代价太高。

### 混合流程

混合是指由多条语句组成的长事务，基本流程：

1. 客户端发起混合事务流程
2. Proxy同单独的insert流程类似，插入数据
3. Proxy同单独的update流程类似，更新数据
4. Proxy同单独的select流程类似，查询数据
5. 并行提交事务
6. 事务全部结束后，向GTM释放GTID
7. Proxy给客户端回OK响应

注：

1. 在UR隔离级别下，select不需要实施和GTID相关的所有步骤
2. Proxy对于非交互事务，proxy可以预先判断是否需要申请gtid，对于交互式事务，proxy碰到第一个写操作即认为本事务需要申请gtid

### 异常流程

#### Proxy异常

##### Proxy进程异常后被自动拉起，没有对等回滚

Proxy异常时，可能存在部分业务未正常结束，存在活跃事务，在proxy被自动拉起后向GTM查询本proxy所有活跃GTID（在proxy成功获取到活跃GTID前不对外服务，避免回滚掉新执行的事务）。

基本流程：

1. proxy宕机重启后首先到PM注册，PM导致proxy需要启动回滚
2. Proxy向GTM查询本proxy的残留活跃事务ID列表
3. 成功获取活跃事务列表开始对外服务，否则一直尝试
4. Proxy后台通知所有节点DBAgent回滚所有残留活跃事务ID
5. 等待所有节点回滚成功后，向GTM释放GTID

注：proxy启动后，PM会通知所在集群是否有对等proxy正在帮你回滚，如果存在，需要去GTM上查询此集群下本proxy的活跃GTID是否回滚完，如果没有回滚完，不能对外提供服务。

##### proxy进程异常后无法被拉起，对等回滚

Proxy进程异常后无法被自动拉起，PM检测到proxy心跳异常通知对等proxy回滚，对等回滚流程和重启回滚执行流程相同。

##### Proxy进程异常无法被拉起，也没有对等回滚

Proxy进程异常无法拉起，也找不到对等proxy，如果后续集群有新增proxy，PM需要通知新增的proxy回滚掉异常但没有执行回滚的proxy。

##### Proxy进程异常后被拉起，对等已经在回滚

Proxy进程启动后，向PM注册时，被告知已经有对等proxy在回滚中，proxy会重复尝试向GTM查询自身是否还存在活跃GTID，直到对等proxy回滚成功后对外服务。

##### PM检测proxy异常导致对外回滚异常

目前方案是PM检测到proxy异常，在界面告警提示，人工确认proxy进程是否异常且无法被拉起，人工触发对等prxoy回滚。

#### GTM异常

##### Proxy启动阶段GTM异常

Proxy启动查询活跃GTID阶段，GTM主备切换或者消息丢失，proxy都会定时重试，重试次数不限直到成功，在重试过程中，proxy不对外服务。

##### 业务执行阶段GTM异常

1. GTM主备切换或消息丢失，proxy都会定时重试，重试次数和重试间隔都可配，在重试时间内，业务挂住，GTM恢复后业务可执行成功
2. GTM主备双机异常，需要人工恢复，正常服务前需要手动执行命令回滚掉全部活跃GTID

#### DB异常

##### 执行阶段异常

1. proxy收到DB返回的错误响应
2. 交互式（非自动提交）返回DB错误信息到客户端，并提示用户需要手动执行rollback
3. 语句块（自动提交）收到DB错误信息后，proxy向所有已执行的DB节点自动下发rollback后返回DB的错误信息到客户端
4. Proxy未收到DB响应

本会话挂住，OS检测到链路长时间无消息（默认9小时）后前端链路主动断链，proxy与DB后端链路断开（写事务在一定时间内不结束会产生告警，可以手动结束本次执行，告警阈值可配）

写事务长时间未结束手动强制中断流程：

Proxy根据设置的事务执行超时时间，检测到事务长时间未结束会产生告警，告警信息中显示此事务所属的客户端IP和端口，需要人工判断事务执行是否正常，如果确认夯住，可以执行：

dbtool -p -xc

根据告警信息中客户单IP（client\_host）找到对应的会话信息dialogid，在MySQL客户端执行kill dialogid，强制中断会话。

##### 提交阶段异常

1. proxy收到DB返回的错误响应
2. 下发commit的所有DB节点均返回错误，返回客户端提交失败响应后释放GTID
3. 下发commit的DB节点部分返回成功，部分返回失败，返回客户端提交失败响应，同时通知后台回滚线程到执行commit成功的DB节点执行已提交回滚

**基本流程：**

2.1）客户端发起commit流程

2.2）proxy执行线程执行发现部分节点commit成功，部分节点commit失败，向客户端提示commit失败，可能数据存在不一致，正在自动恢复中

2.3）proxy执行线程通知回滚线程待回滚的GTID和需要回滚的grouplist

2.4）回滚线程把待回滚GTID加入到本次回滚任务执行队列中，并向该节点DBAgent发起已提交回滚

2.5）proxy收到所有DBAgent回滚成功响应后通知GTM释放

注：如果本次回滚任务中有节点回滚失败，重新向失败的节点发起回滚，一直尝试直到成功。

1. Proxy位收到DB响应

同执行阶段异常，链路超时后proxy前后端链路都断开，提交成功的DB节点无法回滚掉，可以存在数据不一致。

注：由于GTID未释放，数据不会不一致。

##### 回滚阶段异常

1. DBAgent执行反向SQL时DB返回异常

DBAgent立即返回错误到proxy，proxy认为该DB节点回滚异常，重新尝试回滚未成功的GTIDS。

1. DBAgent执行反向SQL时DB无响应

DBAgent在执行反向SQL后，会一直等待DB执行结果，等待过程中会持续回复proxy中间响应，proxy会一直等待DBAgent的最终响应。如果DB异常一直无法回响应，本次回滚会一直持续下去。

已提交的事务，可能出现待回滚的数据被别的事务select for update加锁，是否可以正常处理？

说明：回滚的数据被加锁，DBAgent执行反向SQL时会等待锁释放，等待过程中会定时向proxy回中间响应避免proxy超时，如果锁一直不释放超过了DB设置的innodb\_lock\_wait\_timeout，DBAgent就会向proxy报错。

## 对比

原生2PC的缺陷：

1. 性能问题

无论是在第一阶段的过程中，还是再第二阶段，所有的参与者资源和协调者资源都是被锁住的，只有当所有节点准备完毕，事务协调者才会通知进行全局提交，参与者进行本地事务提交后才会释放资源。这样的过程会比较漫长，对性能影响比较大。

1. 单节点故障

由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（虽然协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

GoldenDB的分布式事务实现方案最大的特点即为其在获得事务强一致性的同时还能具备较高的性能（主要是GTM的高性能设计），可以与常用的两（三）阶段提交做对比：

两（三）阶段提交的核心思想是通过前期的多次准备和协调，尽可能的让最后的提交操作能够成功。而在实际场景中SQL执行成功，最后commit阶段失败的概率极低。本方案认为大部分事务可以一次提交成功，因此采用一阶段提交+补偿事务的方式，与两（三）阶段提交相比，本方案在大部分情况下减少了与数据节点的交互次数，降低了锁冲突的概率，提升了事务处理效率。

虽然如此，分布式事务相比非分布式事务还是会占用更多的资源，GoldenDB提供了灵活的分布式事务控制，应用可以根据实际情况控制是否启动分布式事务。控制参数具备多粒度能力，可以在系统级、会话级、事务级三个维度控制，同时也提供了多种读隔离级别：

UR（uncommited read）：未提交读，即计算节点不做任何分布式事务控制，业务要么允许脏读，要么不存在读的时候同时写；

CR（consistency read）：强一致性读，在高并发读写时，严格杜绝脏读的可能性。

## 待优化

### 存在非必要的网络时延消耗

写入数据后，立刻可以查询到，这个是分布式数据一致性的要求之一。

GoldenDB当前的技术架构，需要计算节点接收到GTID从GTM节点上释放完成后，才能算分布式事务执行完成。计算节点与GTM全局事务节点，当前实现不在一个服务器上，存在网络时延消耗。

改进方向：尽量将全局事务GTID的申请和释放都放在计算节点上完成，至少要做到释放可在计算节点上完成。

注：最新的架构应该已经完善了。

### CR检测冲突时性能下降较大

当有CR检验冲突时，会间隔一定时间，再次发起检验，两次检验之间的时间间隔，就是分布式事务的时延加大的值，也就是导致性能下降明显的原因。

改进方向：待研究

### 被拆分的SQL语句，存在时差

例如，多表join或嵌套子查询，当语句不能整体下发DB时，dbproxy会做拆分，将一条SQL拆分为多条SQL，这个多条SQL之间的查询是有时差的，不可能同时到达所有DB节点，而且DB隔离级别为RC，这个时差的影响更加明显，存在一定的风险性。

改进方向：待研究（目前仍存在该问题）

注：update分发键或者全局索引等涉及SQL拆分的场景，都会存在这种问题。

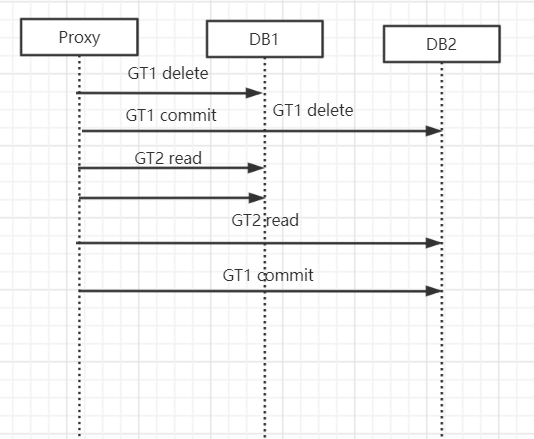
### delete+汇聚函数的实现不是强一致性

1. delete分布式事务，在被已提交事务回滚的期间，其他客户端可能会看到脏数据
2. 汇聚函数统计的数据不是强一致性

例如，客户端A并发执行一个事务，事务中一次insert 10个values，客户端执行select count()，查询出来的count值可能不是10的倍数。

改进方向：待研究（可以采用分布式MVCC方案解决）

### delete流程引起的脏读



如上图所示，GT2读取的数据刚好在GT1的两次cmmit之间的，这样GT2读取的就是脏数据。

解决方法：应用规避，主要措施是严格限制使用delete+小批量删除+非分布式事务的delete。

### delete操作回滚

若分布式事务中包含delete操作，由于在binlog里面无法找到该delete操作对应的GTID，因此delete操作无法进行已提交的回滚。

解决方法：应用规避，主要措施为严格限制使用delete+小批量删除+非分布式事务的delete。

### 分布式死锁

单条语句分解到各个节点上并行执行，和事务涉及多条语句都有可能引起节点间的全局死锁。

解决方法：MySQL上设置session级别的锁等待超时，一旦出现超时，通知客户端失败，业务回退事务。

# 主从复制/一致性

## 主从复制

## 一致性检查

# 备份恢复

## 备份

GoldenDB的备份属于物理备份，备份的数据包括每个分片的数据文件、每个分片的binlog日志及每个时刻的全局事务列表快照，后两种数据在进行集群的一致性数据恢复以及数据恢复到任意时刻时使用。

对GoldenDB的数据备份功能概括如下：

1. 备份任务管理。

GoldenDB支持在OMM上进行备份任务的管理，包括备份任务的发起、备份历史操作的查询等。

备份任务的发起一般有两种模式：

一种为实时备份，即用户配置好后立即会发起一次备份任务；另一种为定时备份，用户可以设置定时备份策略，典型的策略为每周备份全量数据，其他时间分别在周日全量备份的基础上做一次增量备份，如此，可以通过全量数据和其他任一增量备份数据，快速恢复出想要的那天的数据。

1. 备份文件存储。

GoldenDB的备份文件可以在分片节点本地挂载NFS共享目录，将备份结果文件统一存放；同时GoldenDB也支持与IBM TSM系统对接，可以通过TSM直接将备份结果灌入到磁带库。

## 恢复

### 操作

1. 自动恢复
2. 在OMM界面，通过恢复管理功能自动恢复响应的数据；
3. 选择需要恢复的集群、Group、是否一致性回滚、恢复的时间以及需要恢复的DB，单击“自动恢复”，开始恢复。

说明：

1. 支持恢复一个备机或一个位置入集群的未管理及其；
2. 对没有备机的主机进行恢复，结果不保证正确，谨慎操作；
3. 支持自动恢复和手工选择备份文件恢复。
4. 手动恢复（拷贝数据）
5. 如果环境没有安装NFS服务，那么就需要人为拷贝数据到需要恢复的DB上；

拷贝备份文件

拷贝binlog文件

拷贝管理节点保存活跃事务文件

拷贝备份结果文件

1. 数据拷贝完成后，通过OMM界面完成对需要恢复DB的恢复操作

### 功能

GoldenDB的数据恢复功能概括如下：

1. 可恢复到任意时刻。

由于GoldenDB有数据节点的全量及增量备份文件，同时有其运行过程中的binlog日志，借助这些数据可以将系统恢复到任意需要的时刻，但需要注意全量及增量备份文件恢复的速度要大大快于binlog日志的回放速度，因此无特殊要求，建议选择恢复到某次备份的结束时刻，以便更快的完成数据恢复。

1. 一致性的数据恢复。

由于GOldenDB备份了运行过程中的binlog日志及每个时刻的全局事务列表快照，因此可以根据这些信息恢复到一个全局一致性的数据副本。

1. 恢复任务管理。

GoldenDB支持在OMM运维平台上进行节点的一键恢复操作，用户直接在OMM上选定要恢复的节点以及要恢复到的时间点，即可以进行自动化的数据恢复。

# 兼容性

## MySQL兼容性

## Oracle兼容性

### 数据类型

|  |  |  |
| --- | --- | --- |
| 数据类型 | 说明 | 映射Oracle类型 |
| INT | 4字节整数 | NUMBER(10,0) |
| INTEGER | 同INT | NUMBER(10,0) |
| SMALLINT | 2字节整数 | NUMBER(5,0) |
| TINYINT | 1字节整数 | NUMBER(3,0) |
| MEDIUMINT | 3字节整数 | NUMBER(7,0) |
| BIGINT | 8字节整数 | NUMBER(20,0) |
| FLOAT | 4字节单精度浮点数 | NUMBER |
| DOUBLE | 8字节双精度浮点数 | BINARY\_DOUBLE |
| DECIMAL | 数值型，n为数值总长度，范围1-65，s为小数点后长度，范围0-30，s必须不大于n | NUMBER([1-38],[-84-127]) |
| NUMERIC | 同DECIMAL范围 | NUMERIC |
| CHAR | 定长字符串（1~255） | CHAR（2000字节） |
| VARCHAR | 变长字符串（256~2000） | CHAR（2000字节） |
| ENUM | 枚举，其值只能从值列表values1、values2...NULL中选择一个值，最大值为65535 |  |
| SET | 枚举，可取多值，其合法取值列表最大为64 |  |
| BINARY | 定长 | RAW(1-2000) |
| VARBINARY | 变长 | RAW(1-2000) |
| TEXT | 最大长度为65535（2^16-1）字符的TEXT列 | CLOB |
| BLOB | 最大长度为65535（2^16-1）字节的BLOB列 | BLOB |
| DATETIME | 日期和时间，范围：10000-01-01 00::00:01.000000~9999-12-31 23:59.59.999999 | DATE |
| TIMESTAMP | UTC日期和时间，范围：1970-01-01 00::00:01.000000~2038-01-1903:14:07.999999 | DATE |
| DATE | 日期，范围1000-01-01~9999-12-31 | DATE |
| TIME | 时间，范围-838:59:59~838:59:59 | DATE |
| YEAR | 两位或四位格式的年，默认是四位格式。  在四位格式中，允许的值是1901到2155和0000。在两位格式中，允许的值是70-69，表示从1970到2069年 | NUMBER(3,0) |

### 表

GoldenDB提供一个Oracle的存储过程，将Oracle表结构导出成MySQL表结构，我们可以基于生成的MySQL的表结构按照对应的规则进行语法改造。

GoldenDB要求所有表必须有主键（Oracle没有此限制）。对于没有主键的表，可以根据业务规则在原表字段建立联合主键，或者新增一个使用sequence或者自增列作为主键列。

根据业务规则对表结构增加分发规则语句。如果表只涉及单表操作，则可以选取一个离散度较高的列作为数据分片键即可；如果表存在多表联合查询，则需要尽量让这些表以相同的分片键进行数据分片，以便语句的执行也具有分片性，减少运算节点做JOIN运算的压力。

### 关键字替换

### sequence

#### 背景

Sequence用来在多用户环境下产生唯一整数的数据库对象。序列产生器生成数字，它可用于自动生成主键值，并能协调多行或者多表的主键操作。没有sequence，顺序的值只能靠编写程序来生成。先找出最近产生的值然后自增长。这种方法要求一个事务级别的锁，这将导致多用户并发操作的环境下，必须育人等待下一个主键值的产生，而且这种方法很容易产生主键冲突。

还有一个问题，那就是完成生成主键的程序（一般情况包含PLSQL块）本身对于并发调用也是一个瓶颈，因此这样的程序段往往是提供给好多用户调用，如果代码写的不够优化（比如没有使用绑定变量等），或者此代码存在问题，那么它所影响的是系统全局。我们应该提倡开发人员使用sequence，它笑出了序列化问题，而且改善了应用的并发能力。

#### 概述

在oracle中sequence就是序号，每次取的时候它会自动增加。Sequence与表没有关系。

##### 表

Sequence的功能就是产生分布式系统单调变化的序列：为了分布式系统能够像单机系统一样产生可靠的全局单调序列，因此选择系统中控制全局的网元GTM承担sequence用户数据表的创建、维护和消耗功能，但是不维护sequence的元数据。

GTM使用一张表来保存系统用户创建的所有sequence实体，表结构如下：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| clusterId | SeqPrefix | Seqname | starval | currval | nextval | step | minval | maval | cache | cycle | order |
| 1 | Prefix1 | Seq1 | 10 | 12 | 14 | 2 | 1 | 65535 | 20 | 1 | 0 |
| 2 | Prefix2 | Seq2 | 1 | 2 | 3 | 1 | 1 | 65535 | 100 | 1 | 0 |

注；这里涉及cache可以提高并发，但是这样就不能严格递增。

##### 功能

Sequence的作用主要有两个方面：

1. 作为代理主键，唯一识别；
2. 用于记录数据库中最新动作的语句，只要语句有动作（insert/update/delete等）sequence号就会随着更新，所以我们可以根据sequence号来select出更新的语句

##### 特点

将sequence的用户查询数据放到GTM的优点：

1. 复用GTM设计的初衷思想，不需要再引入其他网元；
2. 可以保证整个分布式系统sequence值全局唯一和申请有序。

#### 操作

##### CREATE

创建语句：

CREATE SEQUENCE seq

INCREMENT BY 1 --每次加几个

START WITH 1 --从1开始计数

NOMAX VALUE --不设置最大值

NOCYCLE --一直增加，不循环

CACHE 10; --设置缓存cache个序列，如果系统down掉了或者其他情况将会导致序列不连续，也可以设置为NOCACHE

INCREMENT BY用于定义序列的步长，如果省略，则默认为1，如果出现负值，则代表序列的值是按照此步长递减的。

START WITH定义序列的初始值（即产生的第一值），默认为1。

MAXVALUE定义序列生成器能产生的最大值。选项NOMAXVALUE是默认选项，代表没有最大值定义，这时对于递增序列，系统能够产生的最大值是10的27次方，对于递减序列，最大值是-1。

MINVALUE定义序列生成器能够产生的最小值。选项NOMINVALUE是默认选项，代表没有最小值定义，这时对于递减序列，系统能够产生的最小值是-10的26次方，对于递增序列，最大值为1。

CYCLE和NOCYCLE表示当序列生成器的值达到限制后是否循环。CYCLE代表循环，NOCYCLE代表不循环。如果循环，则当递增序列达到最大值时，循环到最小值；对于递减序列达到最小值时，循环到最大值。如果不循环，达到限制后，继续产生新值就会发生错误。

CACHE（缓存）定义存放序列的内存块的大小。NOCACHE表示不对序列进行内存缓冲。对序列进行内存缓冲，可以改善序列的性能。

**缓存说明：**

Cache参数是为了应对并发访问的。Cache参数高速oracle预先分配一个sequence number的集合，并且保留在内存中，以便sequence number能够被快速的访问。这个内存的大小就是cache所指定的大小，当多个用户同时访问一个sequence的时候，是在oracle SGA中读取sequence当前的合理数值，如果并发访问太大，cache的大小不够，那么就会产生sequence cache相关的等待，影响系统性能。既然cache涉及到了内存，那么就会想到oracle实例恢复的问题。

**如果数据库shutdown abort，sequence会如何？**

当然会有问题，sequence number保存在内存里的但是没有被应用到表中的会丢失！

如果指定cache值，oracle就会预先在内存中设置一些sequence，这样存储比较快。Cache里面的取完了，oracle会自动再取一组到cache。使用cache或许会跳号，比如数据库突然不正常down掉（shutdown abort），cache中的sequence就会丢失，所以可以在sequence的时候用nocache防止这种情况。

注：在GoldenDB中对于sequence是设置为1~255，然后循环。

##### 获取值

##### ALTER

alter sequence [schema.]sequence increment by num

##### DROP

drop sequence [schema.]sequence

#### 限制

1. sequence默认cache为100
2. Sequence前缀含义与oracle不一致，oracle是用户概念，GoldenDB是类似db概念
3. Sequence目前不支持ORDER|NOORDER属性
4. Sequence修改步长时会导致当前cache丢失
5. Sequence默认最大值与最小值与oracle不一致
6. Sequence负的最小值应该比理论int64最小值大1

### synonym

### hint

#### 背景

Hint是oracle数据库的特色功能，是很多DBA优化中经常采用的一个手段。那么oracle为什么会考虑引入优化器呢？

基于代价的优化器是很聪明的，在绝大多数情况下会选择正确的优化器，减轻DBA的负担。但是，有时候它聪明反被聪明误，选择了很差的执行计划，使得某个语句的执行变得奇慢无比。此时，就需要DBA进行人为的干预，高速优化器使用指定的存取路径或连接类型生成执行计划，从而使语句高效地运行。

#### 概述

Hint是oracle提供的一种机制，用来告诉优化器按照告诉它的方式生成执行计划。

##### 注释

提示是oracle为了不破坏和其他数据库引擎之间对SQL语句的兼容性而提供的一种扩展功能。Oracle决定把提示作为一种特殊的注释来添加。它的特殊性表现在提示必须紧跟着DELETE、INSERT、UPDATE或者MERGE关键字。

换句话说，提示不想普通注释那样在SQL语句中随处添加。且在注释分隔符之后的第一个字符必须是加号。

注：OceanBase采用注释的形式加hint信息，GoldenDB采用结尾关键字的形式。

##### 功能

Hint提供的功能非常丰富，可以很灵活地调整语句的执行计划。通过hint信息，我们可以调整：

1. 优化器类型
2. 优化器优化目标
3. 数据读取方式（访问路径）
4. 类型转换类型
5. 表间关联顺序
6. 表间关联类型
7. 并行特性

##### 弊端

1. Hint是比较“暴力”的一种解决方式，不是很优雅，需要开发人员手动修改代码；
2. Hint不会去适应新的变化。比如数据结构、数据规模发生了重大变化，但是使用hint的语句是感知不到变化的，无法获取更优的执行计划；
3. Hint随着数据库版本的变化，可能会有一些差异、甚至废弃的情况。此时，语句本身是无感知的，必须人工测试并修改。

注：比如我们采用storagedb g1，如果数据分布改为g2，那么这个SQL就有问题了。

#### 使用

当遇到SQL执行计划不好的情况，应优先考虑统计信息等问题，而不是直接加hint了事。

如果统计信息无误，应该考虑物理结构是否合理，即没有合适的索引。只有在最后仍然態SQL按照优化的执行计划执行的时候，才考虑使用hint。

毕竟使用hint，需要应用系统修改代码，hint只能解决一条SQL的问题，并且由于数据分布的变化或其他云因（如索引名变更）等，会导致SQL再次出现性能问题。

#### GoldenDB Hint

GoldenDB支持的hint包括：

1. storagedb
2. storagecluster
3. samedb
4. MULTI\_STEP\_QUERY
5. WRITE\_SAMEDB
6. CHGDISKEY
7. NOGTID
8. KEEPALIVE
9. REDISTRIBUTING
10. write\_consistency\_option：SW、CW
11. read\_consistency\_option：UR、CR、Semi-CR
12. read\_write\_split\_option：READMASTER、READSLAVE、READBALANCE

### 存储过程

按照MySQL的语法改写Oracle存储过程。如遇到MySQL不支持的语法，根据业务逻辑进行改写。

改写过程中，有几点细节需要注意：

1. 定义游标时，所有的定义必须在存储过程的一开始（即定义阶段）完成，不能再使用游标时再临时定义；
2. 使用游标进行循环操作时，注意不要遗漏“if not done then”，否则当游标查询为空时，循环操作有可能会出错。

语法差异：

|  |  |  |
| --- | --- | --- |
| 功能 | Oracle语法 | MariaDB语法 |
| 存储过程定义 | create or replace procedure SP\_NAME(v\_in\_trDate in date) is | drop procedure SP\_NAME;  create procedure SP\_NAME(in v\_in\_trDate date) |
| 存储过程调用 | P\_STATUS(para\_1,para\_2) | call P\_STATUS(para\_1,para\_2) |
| DDL | EXECUTE IMMEDIATE ‘TRUNCATE TABLE TABLE\_A’; | TRUNCATE TABLE TABLE\_A; |
| 动态SQL | EXECUTE IMMEDIATE  ‘BEGIN P\_IMP\_’||AU\_TYPE.CODE||’(TO\_DATE(‘’’||DATADATE||’’’,’’YYYY-MM-DD’’));END;’; | set sql\_str=concat(‘call P\_IMP\_’,code\_var,’(date\_format(‘’’,DATADATE,’’’,’’%Y-%m-%d’’));’);  set @p=sql\_str;  prepare s1 from @p;  execute s1; |
| 跳出存储过程 | return | 定义label，并使用leave label跳出 |
| 赋值 | A:=’aaa’ | SET A=’aaa’ |
| 变量长度 | 可以不定义，如varchar2 | 必须定义，如varchar(10) |
| 游标 | FOR rec IN (SELECT ....)  LOOP  ......  COMMIT;  END LOOP; | DECLARE done INT DEFAULT 0;  DECLARE cur1 CURSOR FOR SELECT ...  DECLARE CONTINUE HANDLER FOR SQLSTATE’0200’ SET done=1;  open cur1;  repeat  fetch cur1 into v\_tab;  if not done then  ......  commit;  end if;  util donen end repaet;  close cur1; |
|  |  |  |

### 自定义函数

1. 创建语法

MySQL和Oracle函数在语法创建上有不同，Oracle不要求字段类型都带上长度，但是MySQL要求。

1. LANGUAGE SQL

说明routine\_body部分是由SQL语句组成的，当前系统支持的语言为SQL，SQL是LANGUAGE特性的唯一值。

[NOT] DETERMINISTIC：指明存储过程执行的结果是否正确。

DETERMINISTIC表示结果是确定的。每次执行存储过程时，相同的输入会得到相同的输出。

NOT DETERMINISTIC表示结果是不确定的，相同的输入可能得到不同的输出。如果没有指定任意一个值，默认为[NOT] DETERMINISTIC。

CONTAINS SQL|NOT SQL|READS SQL DATA|MODIFIES SQL DATA：指明子程序使用SQL语句的限制。

CONTAINS SQL表明子程序包含SQL语句，但是不包含读写数据的语句；

NO SQL表明子程序不包含SQL语句；

READS SQL DATA：说明子程序包含读数据的语句；

MODIFIES SQL DATA：表明子程序包含写数据的语句。

默认情况下，系统会指定为CONTAINS SQL。

1. 自定义函数中如果再使用自定义函数，需要添加自定义函数所在的库名，否则会创建失败。
2. 自定义函数不支持动态函数语法。

### 视图

### 系统函数

|  |  |  |
| --- | --- | --- |
| Oracle函数 | MySQL函数 | 功能 |
| instr(v\_tmp,’,’)  instr(v\_tmp,’,’,5,2) | instr(v\_tmp,’,’) | 从字符串中获取字符位置，但是MySQL只能获取第一个位置 |
| VMSYS.VM\_CONCAT | group\_concat | 聚合 |
| pTarget||’%’ | concat(pTarget,’%’) | 拼接 |
| nvl | ifnull | 判断字段是否为空 |
| to\_char() | CAST(123 AS CHAR(3)) | 转换为char类型 |
| to\_number() | CAST(‘123’ AS SIGNED INTEGER) | 字符串转换为数字 |
| to\_date() | date\_format() | 字符串转换为时间格式 |
| cast(a AS numeric(24,6)) | cast(a AS decimal(24,6)) | 转换函数 |
| ADD\_MONTH(sysdate,2) | DATE\_ADD(sysdate(),interval 2 month) |  |
| length() | char\_length() |  |

### MERGE INTO

# 拓展性

## 多级扩展

常见的系统扩展方式有Scale up和Scale out两种；

1. scale up（纵向扩展）主要是利用现有的存储系统，通过不断增加存储容量来满足数据增长的需求。

但是这种方式只增加了容量，而带宽和计算能力并没有相应的增加。所以，整个存储系统很快就会达到性能瓶颈，需要继续扩展。

这个时候有两种方法：

1. 采用更强性能的存储引擎（EMC的Clarion系统和NetAPP的FAS系列，采用增加控制器CPU或内存的方式来提供更强的性能），但是价格昂贵；
2. 额外购买独立的存储系统，这样会增加管理的复杂度

注：OceanBase采用的就是使用单独分布式文件系统，这样不需要采用划分分片，只需要使用分区打散数据，使用paxos做数据分布。

1. Scale out（横向扩展）通常以节点为单位，每个节点往往将包含容量、处理能力和I/O带宽。一个节点被添加到存储系统，系统中的三种资源将同时升级。

Scale-out架构的存储系统在扩展之后，从用户的视角看起来仍然是一个单一的系统，这一点与我们将多个相互独立的存储系统简单的叠加在一个机柜中是完全不同的。

所以，scale-out方式使得存储系统升级工作大大简化，用户能够真正实现按需购买，降低TCO。

GoldenDB的架构设计使得其具有良好的扩展性，可以从如下几个角度分析：

1. Proxy计算节点是无状态的，可以动态任意扩展；
2. 数据库集群可以根据需要进行如下维度的扩展：
3. 在安全组内部添加数据节点增加数据副本数。一方面可以获得更高的可靠性，同时也可以通过读写分离功能达到读能力的线性扩展；
4. 增加安全组数目。获得数据节点存储和计算能力的线性扩展，在增加数据的分片时需要考虑如何将数据重新迁移到新分片上，即表数据的重分布；
5. 增加数据节点集群数目，扩展组建新的集群。该方法可以在得到处理能力扩展的同时做到不同业务间数据的隔离性，用户可以根据要求选择。

GoldenDB计算节点和数据分片的数目配比推荐为1:1，性能可以随计算节点和分片数目的增加线性扩展，损耗控制在10%内。若业务为CPU密集型，则计算节点书目要适当增加，若为IO密集型，计算节点数目可以适当减少。

注：GoldenDB动态扩容的实现是基于proxy的路由管理和一致性哈希算法实现的。

## 数据重分布

业务的增长不可避免的需要对资源进行扩容，由于使用了分片技术，数据被切分橙细小的分片分布在数据节点集群中。集群扩容后，原有的数据分片就面临着被打散重新分配的过程，这个过程就是数据重分布（resharding）。

GoldenDB数据重分布功能的基本原理是对重分布表，建一个和原表字段结构相同但是分发策略为目标分发策略的新表，对于需要重分布的数据分片，将其上的数据导出，然后将该部分数据导入到新表，导入过程会使用新表的分发策略，最后将新表改名为原表。其中数据的导出导入过程又是个多次反复渐进的过程，每次导出的数据均是在前一次导出导入过程中产生的增量数据，直至最后仅剩少量差距时对应用进行短暂禁写，最终完成切换。

GoldenDB的数据重分布对在线业务影响小、且课操作性强，具备以下特点：

1. 执行时间可控制，对业务影响秒级。重分布过程中数据的迁出迁入是循序渐进的过程，仅在原分片和目标分片数据存在少量差距时才会禁写，因此对在线业务影响秒级，并不影响读服务；
2. 支持多表并发重分布。在分布式数据库的数据模型中，会将有关联关系的表采用相同的分片策略。在数据重分布过程中，这些相关表可以选择同步进行处理，以避免在数据重分布过程中由于相关表的分片策略临时不同而导致该SQL语句无法直接下压，影响SQL执行性能；
3. 重分布数据智能识别。数据重分布过程中会涉及到数据从源分片至目的分片的移动，开销很可观，GoldenDB支持对表的源分片信息和目标分片信息进行详细的分析比对，最大程度的减少数据在分片间的移动，提升重分布效率（采用更加细粒度的bucketid，作为隐藏列存入表中）；
4. 可操作性好。GoldenDB在统一运维界面OMM上提供重分布操作界面，可全程可视化操作，包括执行、暂停、继续、取消、异常情况下的重试等；
5. 产品通用性好。对比需要预先做好分片规则的预sharding方案，GOldenDB的重分布方案对设计人员的要求相对较低，GoldenDB支持任意分片策略间的重分布，彻底与业务模型解耦，无需在系统设计初期就要精确规划好未来的数据分布情况。

## 异构数据库

采用slot、o2g（oracle to goledendb）、g2o（goldendb to oracle）、loadserver、datatransfer实现异构数据库之间的在线业务和离线业务对接。

### SLOTH

### LDS

### DTS

# 高并发

## LVS

## Proxy

### 池化技术

#### 线程池

#### 连接池

Proxy对前端采用TCP长链接，客户端一次性将语句下达至proxy；

Proxy对后端采用连接池的方式处理，连接池的数量及用户密码可配置，和端口是一一对应的关系；

Proxy根据应用访问的端口号来选择对应的集群以及具体的连接；

Proxy连接池耗尽后是否可以动态申请可以配置。

### 缓存

GoldenDB通过构建执行计划缓存、SQL缓存，提高SQL解析效率、提高数据读取效率，在高频度读写系统中可以减少磁盘I/O负担，提升整体系统效率。

#### 元数据缓存

Proxy本地会缓存元数据，当MDS发生元数据变更的时候会推送到proxy。

缓存信息包括表结构信息、SQL语句缓存、结果集缓存。

#### 执行计划缓存

#### 结果集缓存

### 并发控制机制

GoldenDB在事务处理上通过采用成熟的主流技术来实现高效的事务管理，这些技术包括：以行级为主表级锁为辅的锁技术、多版本并发控制技术、全乐观锁+自动补偿机制。这些技术在保证事务ACID特征的前提下大大提高了事务的并发处理能力。

### 流控

#### 限流

根据消息积压数进行计算，设置流量限定阈值。

Proxy对上做局部流控，如果当前连接数超过配置的最大连接数，要等某连接上的在线事务全部处理结束后再关闭该连接。

TDSQL对于热点数据处理是在计算节点采用一个hash表预先判断更新数据的分布，GoldenDB采用限流的方案（根据消息积压数计算），如果检测到分发到某个节点的写语句过多，则执行限流，这个粒度（针对group）相对比较大一些。

#### 黑名单

对于某些耗时比较久的SQL，会加入黑名单中。

#### 分包/分布式批处理

对于大结果集进行分包处理（结果集在内存中缓存），防止一次性处理卡顿。

分布式批处理。提供分布式架构下批处理功能，满足金融、政企、运营商等行业日终大数据批处理的要求，通过分布式FetchSize和存储过程功能对数据进行批处理，减少客户端与DB的交互次数，批量返回数据集并进行批量处理。

### 热点数据

使用重分布解决热点数据问题。

### 结果集透传

### 读写分离

读写分离是指利用数据节点集群安全组多副本，将部分读请求发往备节点，提升系统的读能力。

在启动读写分离时，GoldenDB的计算节点在受到应用SQL请求时，根据当前的语句类型和负载策略选择SQL下发的数据节点，将写操作发往主节点，将读操作发往备节点。

注意：GoldenDB的读写分离是事务间的读写分离，如果一个事务内同时存在读写请求，该事务的所有SQL都会发往主节点。

在多个应用接入一个数据节点集群时，为了满足不同应用的需求，GoldenDB支持对同一集群不同的连接实例，设置不同的读写分离模式，包括以下三种：

1. 读主节点。读操作默认发往主节点，当应用强制指定将读请求发往备节点时，读请求即在备机间做负载均衡；
2. 读备节点。读操作仅在备机间根据配置的权重做读负载均衡。

除了上述连接实例级别的读写分离模式设置，GoldenDB还支持SQL级别的读写分离模式指定，应用可以在SQL语句后面添加hint信息强制发往主节点或备节点，SQL级别的优先级高于连接实例级别。常见的使用场景如下：

1. 由于数据在安全组内部的节点间同步存在时延，因此对实时性要求比较高的SQL请求，应用希望将其发往主节点；
2. 对于一些SQL如分析聚合类SQL，应用希望将其发往备节点，减少对主节点的影响。

结合上述连接实例和SQL级别的读写分离模式设置，应用可以根据自己的希望设计合理的读写分离策略。

### group优先级

一般先读同城，再读异地。

### 负载均衡

可以设置不同group的优先级，同城机房高于异地灾备机房。

## SQL引擎

具体参考《分布式数据库SQL优化器》

### 合并下压优化

#### 表层次

#### 主子查询

#### UNION

#### Hint

采用samedb、storagedb的hint信息优化下发db。

Samedb表示自此SQL以后的所有SQL都是采用相同分片，不需要计算分片（适用于跑批业务），storagedb适用于业务侧已知分片信息的情况。

### SQL引擎优化

#### 条件繁殖

#### OR索引失效优化

#### 并行执行

#### AVG优化

#### JOIN优化

##### MULTI\_STEP

存在两表或多表关联操作时，可以使用MULTI\_STEP\_QUERY表示语句被拆分为多步骤执行，目的是提高查询性能。

SQL查询语句中表不需要一定出现在MULTI\_STEP\_QUERY中，如果MULTI\_STEP\_QUERY中定义了表，表的先后顺序表示在SQL语句的执行计划中的执行顺序。

示例：

SELECT

s.s\_name,c.c\_id, concat(c.c\_last, c.c\_first), o.o\_all\_local, o.o\_ol\_cnt

FROM

customer c inner join oorder o

on c.c\_id = o.o\_c\_id

left join warehaouse w

on o.o\_w\_id = w.w\_id

where

w.w\_id = 193 and c.c\_id >= 435 and c.c\_id <= 445

order by w.w\_id, c.c\_id, o.o\_id;

数据量：

customer表：1500万

oorder表：1500万

warehouse表：500万

首先使用原始语句执行，单条语句在16C32G虚拟机上执行报错：

ERROR 10435(HY000):ERR Write IO\_CACHE Fail!

内存监控发现内存写满了，使用count查看结果集，显示为565000条。

**优化方案一：**

修改hash分发方式，保证语句群发，join操作在DB层面执行，proxy层制作排序：

Customer表修改为DISTRIBUTED BY HASH(c\_id)(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(c\_w\_id)(g1,g2,g3,g4);

Ooder表修改为DISTRIBUTED BY HASH(o\_c\_id)(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(o\_w\_id)(g1,g2,g3,g4);

warehouse表修改为DISTRIBUTED BY DUPLICATE(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(w\_id)(g1,g2,g3,g4);

但语句耗时8.89秒

注：该方法不可行，业务不能为了一条语句的提升而导致整体业务性能下降。

**优化方案二：**

分析3张表的数据情况，发现可以使用分布式数据库特有的功能MULTI\_STEP\_QUERY强制先用小表进行JOIN：

SELECT

s.s\_name,c.c\_id, concat(c.c\_last, c.c\_first), o.o\_all\_local, o.o\_ol\_cnt

FROM

customer c inner join oorder o

on c.c\_id = o.o\_c\_id

left join warehaouse w

on o.o\_w\_id = w.w\_id

where

w.w\_id = 193 and c.c\_id >= 435 and c.c\_id <= 445

order by w.w\_id, c.c\_id, o.o\_id MULTI\_STEP\_QUERY(w,o,c);

#### 下推优化

##### WHERE条件下推

##### order by下推

##### distinct下推

##### limit下推

#### 常数折叠

#### 非逻辑优化

#### 死代码消除

#### 全局(唯一)索引

#### 锁

采用乐观锁（重试）和悲观锁（select for update）两种控制策略，针对不同场景设置不同锁类型。

#### force index

为了避免update、delete中where条件索引失效，造成全表锁（对于悲观锁，proxy会先下发select for update where锁住对应数据，然后执行更新），采用在where条件中增加force\_index。

#### Hint

##### NOGTID

##### READMASTER

##### READSLAVE

##### READBALANCE

##### storagedb

不需要计算分片，直接到对应的group。

##### samedb

如果可以确定某一个操作全部是对一个group的操作，前面第一个计算分片信息后，后面的全部添加samedb的hint信息，这样就可以直接用前面缓存的group信息了。

### 执行器

#### MPP

采用presto大数据组件，对于复杂的SQL进行计算。

## DB

### 复制优化(快同步)

分布式数据库通过增加副本数来提高系统的可用性，为了数据的安全可靠，数据必须在满足拥有一定数量的副本之后，才返回处理结果给客户端。GoldenDB采用自研的gSync复制技术，在配置的副本同步策略满足后，主库返回操作结果给客户端；通过线程池、非阻塞式同步、并行复制等关键技术可以实现在确保数据RPO为0的同时保证系统的吞吐量。

### 大容量数据处理

GoldenDB具备完备的海量数据管理功能，提供PB级的数据容量支持，具有高效索引和查询优化技术，具备海量数据处理能力并支持大数据量用户的并发访问。

## GTM

### 多线程

Proxy分别采用三个线程线程分别执行申请、提交、回滚GTID的操作。

GTM线程结构设计为一个分发线程加若干可配置的执行线程。

GTM高性能机制：

#### 批量申请GTID

问题：每个事务都要申请一次GTID，与GTM交互频繁。

性能提升机制：计算节点批量申请GTID，GTM一次执行多个计算节点的批量申请。

#### 批量释放GTID

问题：每个事务都要释放GTID，与GTM交互频繁。

性能提升机制：计算节点批量释放GTID，GTM一次执行多个计算节点的批量释放。

#### 批量查询GTID

问题：每个事务都要查询活跃事务列表，与GTM交互频繁。

性能提升机制：计算节点汇总多次查询，GTM一次执行多个计算节点的汇总查询。

### GTM横向扩展

问题：多集群共用GTM，导致GTM压力大。

性能提升机制：支持多GTM部署，最多一个集群独占一套GTM。

效果：减少计算节点与GTM交互次数（RPC通信），减少GTM日志落盘次数、主从复制次数。

## 待优化

# 高可用

图示

描述已自动生成

## 方案

## 组件高可用

## 部署

### 机房内高可用

即组件高可用

### 同城双活

### 两地三中心

# 数据压缩

GoldenDB支持表数据压缩和备份压缩。表数据压缩时通过对表及分区表进行数据压缩来减少磁盘存储空间，在进行压缩操作前，可以对压缩率进行提前估算，从而做到有针对性的有效压缩，压缩算法采用zlib算法，压缩比在2~11之间。

备份压缩是指备份出来的数据会进行要说来减少空间占用，GoldenDB采用quicklz算法对备份文件进行压缩，压缩比在2~10之间。

GoldenDB同时支持对表的数据空间进行归并整理，可以实现对通过数据压缩和归并整理优化出来的表空间进行释放到表空间，从而实现最大限度的存储优化能力。

# 数据安全

## 备份恢复

## 加密审计

为了确保数据的安全性，数据库需要对数据的全生命周期进行数据安全保证。分布式数据库系统支持数据安全功能包括：

1. 访问鉴权

分布式数据库需要对来访用户进行鉴权，通过IP白名单的方式过滤连接请求，支持通过国密算法对用户名密码进行解密校验。

1. 通道加密

分布式数据库对客户端和数据库的连接进行SSL通道加密，防止通过抓包等手段查看业务数据。

1. 数据加密

分布式数据库支持表级别和列级别的加密。通过列级别的加密，表中仅存储加密后的字符串，在不知道加密串的情况下，即使查询出来也会是乱吗；表级别的加密可以加密整张表，在没有秘钥文件的情况下，即使拷贝走表数据文件，也无法破解表中数据。

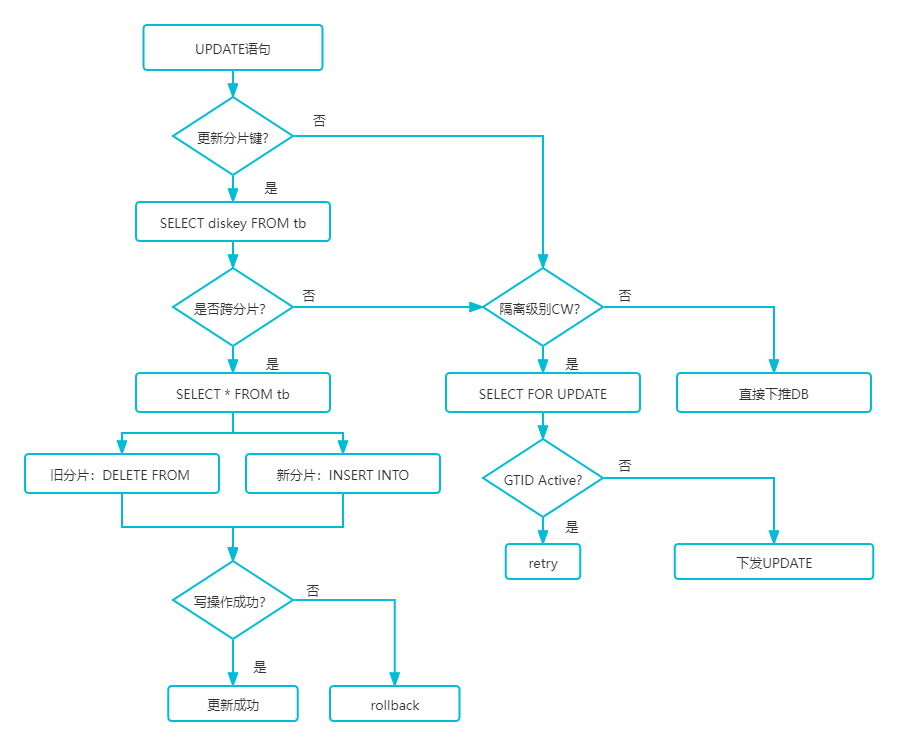
1. SQL日志审计

分布式数据库支持所有的业务SQL写入审计日志文件，以便对SQL语句进行统计和审计，找出潜在的威胁。

# 数据迁移

## COPY TABLE

## UPDATE分片键



## 重分布

对于少量数据的迁移可以采用重分布。

## 导入导出

数据导入导出一般用于系统间的数据迁移，小到导入导出一部分数据，大到数据割接、数据库迁移式升级、数据的分库、海量数据的迁移等，使用场景十分丰富。

GoldenDB支持将集群中符合查询条件的记录导出到指定的数据文件中，也支持将外部数据文件导入到集群合适的数据分片中。

导入导出功能：

1. 外部接口。GoldenDB导入导出的外部接口为文件，待导入的格式要符合要求，一般为字段间采用分隔符隔离的文本文件，导出结果也同样为文本文件。
2. 工具插件。导入导出可以认为是系统的一个功能插件，部署灵活，与在线交易流分离，可以降低对联机交易的影响。
3. 原始文件自动导入数据库集群。GoldenDB的导入导出工具，能自动根据导入表的分片规则，自动对原始数据文件记录分拣，将数据下发至对应数据节点，最终完成导入。这其中的各环节均支持并发处理，提升整体导入效率。工具也支持带条件导入，即仅将满足条件的记录导入至系统中。
4. 多分片数据统一出口导出。GoldenDB的导入导出工具，能够自动控制各个分片并发进行数据导出，并可以选择将导出文件上传至同一个服务器，进一步的，也可以将多个导出文件汇总成单个大文件，用户可以灵活选择控制导出过程。
5. 容错处理。导入操作耗时长，对系统资源占用多，且通常是将外部系统的数据文件导入到分布式系统中，而外部的文件往往会有异常数据。因此，导入操作经常会遇到错误，GoldenDB充分考虑导入异常的错误处理，举例如下：
6. 导入过程中遇到异常数据时能够将异常数据分拣出来，待后续处理；
7. 流程处理的各个阶段都有重试机制，最大限度保证导入成功；
8. 系统异常时能够从异常位置继续执行导入流程；
9. 支持对原始文件记录末尾的空值进行处理。

### SLOTH

### LDS

#### 场景

1. 数据批处理
2. 表结构发生变化
3. 业务迁移

#### 功能

1. 支持结构化的文本文件导入导出
2. 支持导出库表等数据字典信息
3. 支持部分字段导入导出
4. 支持对数据行进行处理后导入导出
5. 支持断点续传
6. 支持导入导出错数据转SQL语句
7. 黑名单权限控制

#### 特性

1. 工具灵活可视，易用性好
2. 数据迁移流与交易流分离，降低对联机业务的影响（一个走前端proxy，一个走后端的dbagent）
3. 容错性高，最大程度完成数据导入导出

### DTS

### Goldendumper

**说明：**

使用loadserver或datatransfer进行数据导入导出，走的是后端链路（DB），导入导出成功后，需要在前端连接proxy设置一下sequence自增列的初始值（因为后端链路导入后sequence的值proxy并不知道）。

使用goldendumper走proxy执行导入导出，这个性能可能会消耗比较大（因为是采用insert+delete的方案），可以采用一个专门数转的proxy做这个操作，配置也做相应的优化。

## 日切卸数

在金融领域，针对应用使用传统数据库存在批前卸数丢失跨天事务，分析报告不准确的问题，需要事后进行手工修复数据。GoldenDB提供日切、卸数插件，可以获得准确的当日业务数据，更快得到准确的分析报告，无需在事后手动修复数据。

以导出日切时刻的数据快照为例，使用GoldenDB日切卸数功能导出数据无需经过ODS处理，既能满足事务全局一致性，又能满足具有业务含义的数据一致性（在日切前发生的事务都保留在快照数据中，日切后发生的事务操作都不体现在快照数据上）。

# 分布式查询优化器

分布式查询优化器是GoldenDB计算节点的核心功能，计算节点在接收到SQL语句后，首先由解析模块转成等价的可识别的语法树，然后由优化模块转成等价的执行计划树，最后由执行模块按照步骤执行该计划树，从而完成整个SQL语句的执行。

## 影响因素

优化器的优化工作主要体现在计划树的生成上，GoldenDB的查询优化器设计实现主要考虑以下两个方面：

1. 代价模型的选择。

GoldenDB采用分布式系统代价估算模型，考虑节点间传输数据的代价，以减少数据传输的次数和数据量作为查询优化的目标，提高数据节点之间计算的并行度、减少计算节点的计算量。这主要考虑在分布式数据库系统环境中，表结构被水平或垂直拆分到多个数据节点，因此需要考虑语句如何分拆、分片之间数据如何移动、结果如何计算与合并的问题，网络通信开销不可忽视。

1. 考虑数据一致性开销。

在分布式数据库系统中，数据全局一致性机制相较于单机数据库需要更为复杂的控制。因此，如何降低数据全局一致性保证的开销，也是GoldenDB查询优化器的设计要求。

总体来讲，GOldenDB的分布式查询优化器遵循了上述的设计原则，以基于规则的优化为主，基于成本的优化为辅，在提升系统的灵活性的同时控制系统实现的复杂性。优化器内部内置大量的优化规则，通过查询重写的方式进行经验性优化。在优化规则的选择上，重点分析分片剪枝、并行执行、合并下压、条件下推、条件繁殖、排序消除、去重消除、排序下推等。

内置大量的优化规则，对上百个场景进行优化，复杂SQL语句兼容性和处理性能好，同时支持prepare预处理、执行计划缓存、数据集透传等功能，保证数据一致性条件下实现高性能SQL处理。

支持的典型优化包括：

1. 分片剪枝
2. 合并下压
3. 并行执行
4. 条件下推及条件繁殖
5. 排序下推、limit下推等
6. 聚合函数优化

得益于完善的优化器设计，使得GoldenDB对单节点、跨节点的复杂SQL的兼容支持程度很高，包括跨节点SUM、COUNT、AVG等汇聚类操作，跨节点WHERE、FROM等子查询，跨节点JOIN，跨节点GROUP BY、ORDER BY、LIMIT等。这是GoldenDB将Proxy命名为计算节点而非中间件的原因之一，也是其和很多分布式数据库产品中间件的主要区别。

## 优化策略

### 条件繁殖

条件繁殖是指优化器对已知条件进行推断，从而衍生出其他条件进行改造，缩小数据检索的范围。繁殖后的条件，或推入基表、或下压到数据节点执行。

SQL示例1：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col>100 UR;

优化后执行计划示例：

优化器会识别出本例中的条件传递从而推断出T2.col>100，并将该条件推入基表。语句被重写：

select T1.col,T1.col1 from {T1 where col>100} join {T2 where col>100} on T1.col=T2.col UR;

注：在等价语句改造的时候，也可以利用ON条件等做等值的传递。

SQL示例2：

select T1.col, T1.col1 from T1 join T2 on T1.col=T2.col where T1.col1 in (100,200) and T2.col1=T1.col1 UR;

优化器执行计划示例：

优化器会识别出本例中的条件传递从而推出T2.col1 in (100,200)，并将该条件推入基表。语句被重写为：

select T1.col, T1.col1 from {T1 where col1 in (100,200)} join {T2 where col1 in (100,200)} von T1.col=T2.col UR;

### 并行执行

并行执行是指执行计划在各个分区间进行并行执行，从而提升执行效率。

当SQL查询在分区剪裁后，仍然涉及多个分区时，会生成一个分布式执行计划，该分布式计划会被调度到分区所在不同机器上进行执行。GoldenDB在判断语句需要下发到多个分区时，会将语句拆分成多个同时下发到对应的节点并行执行。

SQL示例：

假定T1、T2在g1,g2,g3三个分区上：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col1=200 UR;

优化后执行计划，语句被拆分如下：

select T2.col from T2 order by T2.col ASC;

select T1.col,T1.col1 from T1 where T1.col1=200 order by T1.col ASC;

同时下发到对应的三个节点上并行执行后，将结果汇总到proxy层做join。

### AVG优化

在GoldenDB中，AVG被自动重写成SUM和COUNT两个计算，在每个数据节点上，只返回本数据节点的SUM、COUNT；在计算节点层面，再对各数据节点返回的SUM、COUNT进行累计，然后再用SUM/COUNT得到AVG的最终结果。

SQL示例：

select avg(col) from T where col>100 UR;

优化后执行计划示例：

select sum(col),count(col) from T where col>100;

### WHERE条件下推

在分布式数据库系统实现中，为了尽量减少数据节点向计算节点移动的数据量，系统被设计为尽可能将where条件下推到数据节点。

SQL示例：

假定T为单分发键表，且分发键为col：

select col,col1 from T where col=100 and col1>10 UR;

优化后执行计划示例：

根据where子句可以将查询数据定位到某（几）个GROUP，执行时将语句直接下发到对应的group节点上执行：

select col,col1 from T where col=100 and col1>10;

### order by下推

GoldenDB对于排序处理通常会优先考虑推入数据节点完成。利用数据节点的计算能力并行完成排序操作；涉及结果合并的，计算节点再对有序数据集进行合并排序。

SQL示例：

select col,col1 from T where col1>100 order by col UR;

优化后执行计划示例：

将语句下发到各个节点上并行执行，并在proxy层汇总结果，如果下发为多节点，则需要执行sort merge排序操作。

select col,col1 from T where col1>100 order by col;

### distinct下推

GoldenDB中，遇到不能合并下发的SQL语句，如果其中含有distinct，则计算节点在拆分语句时，会考虑将distinct下推入数据节点执行。以减少从数据节点提取到计算节点的数据量。

UR场景下，如果查询数据分布在同一个节点上或者select list为分发键的情况下，将distinct下推入节点执行，汇总结果不需要在proxy层再做distinct。

SQL示例：

假定T为多分发键表，且分发键为col1,col1：

select distinct col,col1 from T where col>100 and col1=20 UR;

优化后执行计划示例：

select distinct col,col1 from T where col>100 and col1=20;

### limit下推

在GoldenDB中，limit下推的主要目的是在需要计算节点进一步计算的场景下，尽量减少从数据节点提取到计算节点的数据量。

优化原则：

1. SQL语句能下发到一个db group执行的，limit子句不用调整
2. SQL语句下发到多个db group执行的，需要在proxy层汇总数据，做limit操作：
3. SQL语句能下发，但是需要下发到多个db group执行的，limit子句需要调整，调整格式如下：limit x,y --> limit 0,x+y
4. SQL语句不能下发，需要把数据拉到db proxy层计算的，limit子句不变

SQL示例：

查询数据分布在多个节点上，且查询语句能下发：

select col,col1 from T where col1>100 limit 2,2 UR;

优化后执行计划：

select col,col1 from T where col1>100 limit 0,4;

### 常数折叠

在GoldenDB中，为了减少对确定值的反复计算而先进行计算的优化方法。此过程一般发生在S（Select）F（From）W（Where）中SW阶段。

SQL示例：

假定T为单分发键表，且分发键为col：

select col,col1 from T where col=50+50 and col1>10 UR;

优化后执行假话示例：

在此查询中，50+50会被先计算成100。查询重写后，条件变为where col=100 and col1>10。根据重写后的where条件可以将查询语句定位到某（几）个group上执行，执行时下发原始条件。

select col,col1 from T where col=50+50 and col>10;

### 非逻辑优化

在GoldenDB中，会针对NOT运算进行处理，通常是将其下推，将表达式整体取反变为表达式分量补集的运算。

|  |  |
| --- | --- |
| 处理前 | 处理后 |
| NOT (col!=5) | col=5 |
| NOT(col1<=4 OR col2>0) | col1>4 AND col2<=0 |
| NOT(col1<=4 AND col2>0) | col1>4 OR col2<=0 |

经过变换，可以减少一次逻辑运算并在一定条件下使范围扫描可用。

### 死代码消除

GoldenDB分布式优化器中的处理逻辑通过判断出为恒指或者逻辑冗余的条件，然后在运行时减少不必要的逻辑判断，从而提升执行效率。

SQL示例：

假定T为range表，col为单分发键：

select col,col1 from T where col>0 and col > 200 UR;

优化后执行计划：

此例子中，根据where条件确定下发group（col>200所在group）：

select col,col1 from T where col>0 and col>200;

SQL示例：

select col,sol1 from T where col<0 and col>200 UR;

优化后执行计划：

此例中，根据where条件获取下发group为0，选择一个group下发执行：

select col,col1 from T where col<0 and col>200;

### 合并下发

GoldenDB的分布式优化，很重要的一个努力方向就是尽量利用数据节点的计算能力进行计算，避免不必要的从数据节点向计算节点的数据移动，并减少和数据节点交互的次数。计算节点分析语句后，尽量把能够一起执行的语句下发到数据节点。

注：这里涉及到表层次合并、主子查询合并以及JOIN等合并下发规则的判断。

SQL示例：

假定T1和T2分发属性相同：range表、分发键为col、分布在g1,g2,g3节点上，将“条件繁殖”部分的示例语句改写如下：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col=200 UR;

见“where条件下推”部分，根据where子句可以将查询数据定位到某个group上，则可以直接将语句下发到group节点上执行。

优化后执行计划：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col=200;

# 集群构建

## 设备资源管理

**基本步骤：**

创建城市信息->新增机房信息->手动注册组件信息

**创建城市信息：**

各类资源通过机房为单位进行管理，机房通过城市进行划分。

**新增机房信息：**

机房又分为本地机房与异地机房，根据城市区分。

## 集群管理

GTM管理->Cluster管理->DBGroup管理->DB管理

## 连接管理

DBProxy管理->添加绑定ROOT->集群用户管理->连接实例管理

# 监控运维

## 流水号

系统表information\_schema.INNODB\_TRX主要记录了innodb事务的相关信息，需要增加2个字段用于保存事务流水号信息及GTID信息。

新增字段信息如下：

trx\_serial\_num varchar(32) DEFAULT NULL,

trx\_gtm\_gtid varcahr(32) DEFAULT NULL

事务流水号信息和GTID信息都是以特殊HINT信息的方式携带在SQL语句中的，如：

事务流水号：/\*+TSN=abc123\*/ START TRANSACTION;

事务流水号：/\*+GTID=123456\*/ START TRANSACTION;

**为什么要做？**

事务流水号可以用于全链路的监控，以及问题的排查过程。

而GTID已经应用在映射binlog位置的功能中，大大提高已提交事务的回滚效率。

## 日志

**系统日志**

**OMM操作日志**

记录所有OMM下发的操作信息。

**Moni日志**

使用dbmoni命令时产生的日志，记录程序的启动/停止，日志文件为dbmoni.log，在安装用户的$HOME/log目录下。

**程序运行日志**

程序运行日志在安装用户的$HOME/log目录下，日志文件达到100MB后自动切换到下一个文件，前一个文件会更名为“模块名-日期-序号.log”。

**慢日志**

**Proxy慢日志**

Proxy慢查询开关、阈值、日志文件路径都可以通过配置文件设置。Proxy慢查询日志关键内容分析：

TotalExecTime：从应用接收SQL到返回应用响应的总时间

SQL：执行的SQL语句

MsgToExecTime：从消息线程发送到执行线程的时间

ParserSQLTime：解析SQL语法的时间

PlanTreeCreateTime：创建执行计划的时间

GetGTIDTime：获取GTID的时间

GreeGTIDTime：释放GTID时间

PlanTreeExecTime：执行计划树的执行时间

SubSQL[N]：第N条子查询

ExecTime：子查询的执行时间

FinishTime：proxy处理子查询的时间

g1 num:1,duration:104577us：执行子查询的某个分片的执行时间

DB connection\_id:2534578,duration:5467us：DB上执行时间

**DB慢日志**

# Time: 2018-12-18T05:55:15.941477Z

# User@Host: root[root] @ localhost [] Id: 53

# Query\_time: 2.000479 Lock\_time: 0.000000 Rows\_sent: 1 Rows\_examined: 0

#Req\_wait\_time:0.0 Pre\_dispatch\_time:0.000058 Parse\_time:0.000127 Execute\_time:0.019889

#Exec\_prep\_time=0.000009 open\_table\_time=0.006844 Mdl\_req\_time=0.00001 Innodblock\_req\_time:0.00000

#Order\_commit\_time:0.012669 Flush\_time:0.011767 Sync\_time:0.000044 Commit\_time:0.000044 Ack\_wait\_time:0.000005 Commit\_queue\_wait\_time:0.000000

SET timestamp=1560131835;

insert into t1 values(10);

**参考如下解释：**

# Time：当前打印的时间

# User@Host：当前用户

id：connection\_id

# Query\_time：查询总时间，从事件响应到执行结束的时间

Lock\_time：“锁”时间，mysql源码定义，时间从解析命令开始打获取到需要的锁的时间，也包括缓存表的时间等，并非单独获取锁的时间

Rows\_sent：返回记录数

Rows\_examined：扫描的行数

#Req\_wait\_time：线程池队列等待时间，单位为ms

Pre\_dispatch\_time：出线程池队列到命令开始解析的时间

Parse\_time：命令解析时间

Execute\_time：命令执行时间

#Exec\_prep\_time：执行前准备时间

open\_table\_time：打开缓存表和数据字典的时间，包含获取MDL锁时间

Mdl\_req\_time：获取MDL锁的时间

Innodblock\_req\_time：获取innodb锁的时间

#Order\_commit\_time：提交的总时间

Flush\_time：写binlog到进入缓存，sync redo落盘

Sync\_time：sync binlog落盘

Ack\_wait\_time：备机响应的时间

Commit\_time：存储引擎提交时间以及等备机响应

Commit\_queue\_wait\_time：提交队列中等待的时间

SET timestamp：表示当前的时间戳

insert into t1 values(10)：当前慢查询日志所执行的语句

**DB锁等待日志**

经过一次锁等待日志格式化后的锁等待日志格式如下：

[2020-06-10T01:00:00.1]||||0||||#WARN DESC=lock\_wait\_time:more than 200ms

req\_thd\_id:2, req\_trx\_id:123456, req\_trx\_seq:0, req\_gtm\_gtid:0, req\_sql:[update t1 set id=1 where name=’x’]

blk\_thd\_id:3, blk\_trx\_id:123457, blk\_trx\_seq:0, blk\_gtm\_gtid:0, blk\_key\_len:[211,2]

说明：

第一行：[时间戳]+固定格式+请求事务交易流水号+固定格式+#WARN DESC=+锁等待时间

第二行：

req\_thd\_id：请求事务thread\_id

req\_trx\_id：请求事务trx\_id

req\_trx\_seq：请求事务流水号，字符串最大长度32

req\_gtm\_gtid：请求事务gtm\_gtid，字符串最大长度32

req\_sql：请求事务SQL语句，字符串最大长度2048

第三行：

blk\_thd\_id：阻塞事务thread\_id

blk\_trx\_id：阻塞事务trx\_id

blk\_trx\_seq：阻塞事务流水号，字符串最大长度32

blk\_gtm\_gtid：阻塞事务gtm\_gtid，字符串最大长度32

blk\_key\_len：阻塞事务的索引键值，字符串最大长度256

## 监控和故障诊断工具

### gdbpd

GoldenDB问题诊断工具：提供从应用经过计算节点到数据节点的全链路交易跟踪及问题诊断能力。



GoldenDB PD工具是针对GoldenDB数据库设计的一种工具。该工具可用于故障检修、问题确定、数据库监控、性能调优和帮助应用程序 的开发设计。

工具GoldenDB PD设计为命令行方式，用于分布式数据库二线支持人员对数据库进行监控个故障诊断。

#### DB诊断

显示所有锁，包括innodb行锁及MDL锁

显示所有锁等待

显示最近一次死锁

显示所有线程

显示数据库恢复进度

显示所有表RUID次数：显示各表的读次数、插入次数、更新次数、删除次数；各表中各索引扫描次数；显示各表全表扫描次数

显示历史慢查询

显示所有数据库进程堆栈

显示所有表统计信息

显示所有事务：显示数据库当前正在执行的所有事务

DB诊断示例：显示所有表RUID次数

gdbpd -ruid -u dbproxy -p ‘xxxxx’ -d test

#### Proxy诊断

Proxy慢查询日志：

gdbpd -slowlog：显示慢查询日志中客户端IP

#### CM诊断

显示主备复制状态：

gdbpd -repl-state -c 1

主要参考两个值：

SyncLogGap：备机同步日志落后位置

SyncLogDelayTime(s)：备机同步日志延迟时间，单位为秒

显示所有表RUID次数：

gdbpd -ruid -d test -t user\_tb | more

显示数据倾斜度，显示表数据分布情况：

gdbpd -grad -从1 | grep-C1 ‘loadmoni.c1\_t1\_in’

### dbtool

dbtool工具是GoldenDB系统提供给系统操作人员和运维人员的命令行工具。通过dbtool工具，可以帮助系统操作人员和运维人员快速对系统内部的检查运行状态、动态更改参数、问题定位和故障恢复等。

dbtool工具通过系统内部集成的专用接口连接内部进行操作，保证系统内部运行的安全，快速响应，提供高效安全的处理方式。

工具通用功能：

修改、查看、删除配置项信息

版本信息查看

动态加载配置项

进程状态查看

进程内存信息查看

进程网元链路

查看指定级别日志

查看指定uuid日志

查看日志关键字及描述信息

查看线程实时消息积压

查看线程消息统计

#### 查看网元链路

dbtool -mds -ls

#### 查看线程实时消息积压

dbtool -mds -thdmsg

说明：

pname：表示线程的名称

pno：表示线程号

MsgQueueNum：表示当前时刻线程的消息积压数

#### PM工具

PM重新获取全量元数据（因为网络异常原因导致PM与MDS中元数据不一致，不需要重新启动进程去获取最新的元数据，通过命令实时更新本地的元数据信息即可）

通过PM回滚掉指定集群上的某个GTID

通过PM追踪指定集群上的事务日志

异常DBProxy活跃事务回滚

#### MDS工具

启动/禁用集群上的连接实例

查看密码版本

查看当前活跃GTID

禁用指定集群上的库表

查询不同集群上的最大GTID

指定某个机房的GTM为主机

指定某个GTM为主机

#### CM工具

强制指定集群中的主机

检测主备机数据一致性

查询集群元数据信息

查询最小一致性时刻

回滚到指定时刻

持久化分片的GTID

同城人工演练

异地人工演练

城市间数据一致性检查

将DB的GTID进行purger到上一次快照的值

根据角色生成差异化sql语句

根据IP和Port生成差异化sql语句

查询group中最新备机

强制/非强制指定主切换

查询指定集群全局活跃GTID

#### DBAgent工具

人工进行回滚失败的GTID

人工进行binlog备份

查看DB的基本信息

实时查看已回滚的全局事务

查看实时元数据

查看DB心跳表最近一次时间戳

启停DB监控服务

查找全局事务信息

#### Proxy工具

##### Proxy运维命令

查看proxy基本信息

查看proxy链接集群信息

查看proxy绑定连接实例信息

查看proxy连接实例会话信息

##### 执行模块运维命令

查询客户端会话信息

查看配置管理模块会话信息

查看当前正在处理的事务

查看执行时间超过指定时长的事务

查看超时定时器信息

查看所有事件号或指定事件号信息

查看超时会话

查看外部网元交互统计

查看会话内存变量值

会话级慢查询功能

性能统计分析功能

统计指定时间段内指定的DML语句

##### Proxy路由运维命令

查询路由的task缓存信息

查询路由的sql缓存信息

查询路由的会话信息

查询路由的监听队列信息

查询路由的垃圾队列信息

##### 连接池运维命令

查询连接池的使用情况

查询连接池的链路断开情况

查询连接池的垃圾队列信息

Proxy配置项查看

Proxy动态修改配置

Proxy不同维度语句统计查询

Proxy回滚失败信息

动态设置日志跟踪

#### GTM工具

动态配置日志级别

进程状态详情显示

GTID查询

GTID清理

特定维度GTID查看

查看配置项信息

查看proxy信息

查看集群信息

查看事件号信息

查看sequence信息

性能统计分析功能

二进制增量文件转换为文本形式

## 巡检脚本

## OMM

GoldenDB支持的运维能力包括：

1. 一键式安装、升级。
2. 配置可视化。
3. 统计监控。
4. 故障自动检测和告警。
5. 日志。
6. Explain查看执行计划。
7. Dbtoll管理工具。
8. OMM统一运维平台。提供网络拓扑、集群管理、元数据管理等功能。

## Insight

### 概述

OMM的升级版本，提供了更加详细的监控信息。

Insight是GoldenDB分布式数据库的监控系统，采用了目前流行的开源组件Elasticsearch、Zookeeper、kafka等开源组件，结合GoldenDB分布式集群系统而开发的一套包含各种监控指标采集、存储、展示的高性能的监控系统。

该监控系统包含服务器的系统资源，以及各个组件的指标监控，以及SQL的相关统计监控。

采集系统采用非侵入式的信息采集，对原系统的运行不会造成影响，采用kafka高性能消息队列，zook多节点部署，满足监控系统高可用、高性能。

### 监控

#### 服务器列表

服务器列表包含分布式系统所有机器资源，采用不同级别颜色显示资源的占用等级。

#### 网络监控信息

网络监控页面可以对两地三中心的网络流量进行实时监控，通过对丢包率、最大时延、最小时延的配置，实时采集符合配置的网络数据显示，历史查询中可以按照组件进行过去时间段的网络流量统计情况。

#### 组件详细信息

每个组件详细信息中包含性能、空间、配置、日志、详情、属性等数据统计。

##### 数据组件

数据组件按照集群现实DBGroup数、DB数、连接实例分类现实。

数据组件的详细信息包含详细的DB表锁冲突、行锁平均等待、缓冲区脏页、缓冲区命中率、临时表使用率、文件临时表使用、binlog磁盘使用和重做日志等待等指标的采集统计。

##### 组件性能数据

##### 组件节点日志

### 运维

#### 组件管理

#### SQL诊断

SQL诊断包括慢SQL分析、TOP SQL、事务统计、SQL统计、SQL执行计划等相关指标的采集和显示。

#### 会话诊断

会话诊断包含连接信息统计，DB节点的锁信息，性能统计包含集群和计算节点两方面的SQL指标统计和GTID申请、释放以及活跃的GTID统计趋势等。

#### 空间诊断

空间诊断是对数据组件的表空间信息进行采集统计信息，包含DB空间诊断和表空间诊断，DB空间显示节点、目录、数据量大小、使用量和剩余量，近一周的数据增长量，预测空间可使用时间，表空间统计相关的表的数据行数、数据空间，索引空间以及碎片空间占用统计。

#### 日志分析

日志分析包含各个组件的日志，按照日志产生时间，组件模块、日志级别、日志中的关键字以及相关业务功能UUID和交易的流水号等条件进行查找 信息可以对组件日志进行下载本地详细查看。

#### 告警管理及统计

告警管理包含分布式系统的各个级别的告警信息统计趋势图，按照告警时间、类型和告警级别的查询，具体告警信息原因及解决建议。

管理员通过告警配置可以设置相关的告警阈值，显示达到阈值设置的告警信息。

# 使用规范

## SQL规范

### 隔离级别

### 数据类型

### 语法

### 特有SQL

## JAVA开发指导

### JDBC

### 连接池