PostDB 3.0

Replica设计说明

# 概述

实现postdb计算节点支持主备复制的功能，主要包括：

* 支持一主多从
* 自动选主和切主
* 根据wal日志进行同步
* 保持一致性读

# postmaster状态机



基本保持postmaster现状状态机机制，在此基础上修改，要去除bgwriter的启动处理流程，replica不需要bgwriter系统任务。

增加CNODE\_LEADER\_ELECTION状态，postmaster启动后所有的计算节点，都先进入到CNODE\_LEADER\_ELECTION状态，在此状态中集群执行选主流程。选主完成后主节点进入到CLUSTER\_RECOVERY状态进行集群recovery流程，在完成后则进入到STARTUP状态。从节点直接进入到STARTUP状态。

startup任务中执行的流程：若为主节点则执行StartupPrimaryNode函数从存储节点ControlFile等状态信息，若为从节点则执行StartupReplicaNode函数用于获得主节点主动发送的ControlFile等初始信息，并持续进行日志回放的功能，获得主节点的ControllFile信息时，需要等待主节点完成集群恢复进入到STARTUP状态。

从节点回放日志后发现同步完从节点的初始化信息则进入到HOT\_STANDBY状态，此状态可以接收只读访问的连接。

当从节点发现和主节点连接断开时，从节点需要进入到PM\_WAIT\_BACKENDS状态，将所有正在工作的backend进程退出，清理内存的状态，包括已经接收到的wal日志，bufmanger中，relcache等等信息，并重新进入到PM\_INIT状态，之后若再此连接成功将从一个新的位置重新接收日志。

主节点发送某从节点断开后，将清除从节点的并尝试重新连接。

# ps\_cnode\_server

新增ps\_cnode\_server模块，类似于存储节点ps\_server模块的作用，主要负责启动计算节点网络的服务端监听，使用现有网络模块，开启监听，分发收到的网络请求等。

增加guc参数peer\_addresses用于配置计算节点之间监听的ip地址，peer\_port用于配置计算节点之间连接监听的端口。

节点刚启动时不知道自己是主节点还是从节点，需要主动建立和其他计算级节点的连接，在确定了自己是从节点则断开和主连接之间连接，因为主节点已经主动连接上从节点。



# Coordinator模块

现有Coordinator模块用于连接所有的存储节点，并根据存储节点的状态进行不同的控制流程，现在需要增加用于连接所有的计算节点并进行类似的状态控制过程。

仍然使用原有的状态值对计算节点的状态进行控制。

NODE\_OFFLINE: 表示此计算节点尚未建立连接。

NODE\_CONNECTED: 表示此计算节点已经建立好了连接。

NODE\_RECOVERY：表示此计算节点正在进行选主或集群恢复的过程。

NODE\_NORMAL：表示此计算节点可以正常工作。

NODE\_TOBE\_CLOSED：表示计算节点之间的有错误即将关闭，比如长时间没有心跳的情况下。

NODE\_SHUTDOWN：表示此节点的连接关闭，以后不会自动连接。

# 主节点新增加的wal日志

为了使从节点可以在连接到主节点后获得运行时一致的状态需要增加以下WAL日志，日志的资源类型为RM\_STANDBY\_ID。

* 活动事务列表的日志: XLOG\_RUNNING\_XACTS
* ~~一致性读日志: XLOG\_CONSISTENCY\_READ~~

Coordinator负责在某从节点连接上来，或某自己刚刚成为主节点后，就产生以上新的日志，从节点必须在回放完以上所有的日志后，表示在此时已经获得和主节点运行期的初始状态了，才可以进入到HOT\_STANDBY状态。



# 节点状态和心跳

计算节点主从之间需要发送心跳信息，但是从节点之间不发送状态信息，发送方向是从向主发送状态，主向从回复。心跳信息由从节点的ps\_hearbeat来发起执行。

从节点的状态信息基本上和存储节点的状态信息类似，NWL，NCL，NRL。

主节点回复PGCL, PGRL等信息。

若一段时间后主节点未收到从节点的状态信息，则认为连接断开，节点切换到NODE\_TOBE\_CLOSED状态。

若一段时间后从节点未接收到主节点的状态信息，则认为连接断开，从节点开启切主的流程。

主节点需要增加Rep\_NCL的状态，表示所有从节点已经收到的最后的NCL信息，Rep\_NRL表示所有从节点已经回放了NRL信息。此信息根据从节点收到的状态信息进行推进。从节点初始的NCL可以设置为刚连接上时主节点的dispatcher队列里最后一条日志的lsn号。

和主节点一样从节点也需要获得存储节点发送过来的心跳信息。

一般的在各个消息的头上都带状态信息，只有当一段时间后没有任何消息发送的时候才会主动发送心跳信息。

# wal日志发送

在现有WALDispatcher的基础上增加将wal日志发送给从的功能。

当新的从节点连接上来并进入到NORMAL状态后，将从此时开始记录发送的开始位置，之前的历史日志不再发送。

WALDispatcher仅将wal日志发送给状态为NODE\_NORMAL状态的节点，若当集群配置了多个计算节点则发送队列中的条目仅当Rep\_NCL和PGCL都推进后才能删除。

现有WALConsitencyManager增加同时负责接收对wal发送给从节点的回复信息，并将发送队列中对应条目的got\_ack设置为true，避免重新发送。



# wal日志接收

新增模块ReplicaWALRecevier，用于从节点接收处理主节点发送过来的系统任务，其为一个常驻的系统任务，在退出PM\_CLUSTER\_RECOVERY状态前知道自己是从节点的时候启动。负责接收wal日志，并进行排序，放入到内存，其主要的数据结构如下：

|  |
| --- |
| /\* one replica received lsn record \*/  typedef struct ReplicaWalRec  {  XLogRecPtr lsn;  XLogRecPtr pre\_lsn;  char\* data;  } ReplicaWalRec;  /\* control data of replica node wal receive and replay \*/  typedef struct ReplicaControllData  {  XLogRecPtr NWL;  XLogRecPtr NCL;  XLogRecPtr NRL;  List\* raw\_records; /\* raw received wal record \*/  slock\_t raw\_lock; /\* lock to protect raw\_records list \*/  List\* records; /\* wal records list in lsn order \*/  slock\_t rep\_lock; /\* lock to protect records list \*/  MemoryContext\* rep\_mem;  } ReplicaControllData; |

# wal日志回放

从节点的回放由startup系统任务来执行，需要从内存中回放接收到的wal日志，并更新回放位置的状态信息，回放过程中若发生错误则报错后直接退出。

需要增加模块WALReader，用于实现在内存中回放wal日志的功能，需要实现xlogreader相关的接口，在未接收到后续数据的时候的等待触发。

回放的目的是更新内存中的信息，若需要更新不在bufmanger中的数据页则不加载，仅回放生成此lsn的read consistency points结构中的信息。



# wal同步方式

保持原生postgresql的wal多种日志的同步方式，需要在不同的方式下等待不同的推进状态，原生pg的wal同步方式有：

|  |
| --- |
| typedef enum  {  SYNCHRONOUS\_COMMIT\_OFF, /\* asynchronous commit \*/  SYNCHRONOUS\_COMMIT\_LOCAL\_FLUSH, /\* wait for local flush only \*/  SYNCHRONOUS\_COMMIT\_REMOTE\_WRITE, /\* wait for local flush and remote write \*/  SYNCHRONOUS\_COMMIT\_REMOTE\_FLUSH, /\* wait for local and remote flush \*/  SYNCHRONOUS\_COMMIT\_REMOTE\_APPLY /\* wait for local and remote flush and remote apply \*/  } SyncCommitLevel; |

对于SYNCHRONOUS\_COMMIT\_OFF, 主节点只把wal放入发送队列即返回，SYNCHRONOUS\_COMMIT\_LOCAL\_FLUSH，主节点需要等PGCL推进后返回，SYNCHRONOUS\_COMMIT\_REMOTE\_WRITE和SYNCHRONOUS\_COMMIT\_REMOTE\_FLUSH主节点需要等PGCL推进和Rep\_NCL推进，SYNCHRONOUS\_COMMIT\_REMOTE\_APPLY需要等Rep\_NRL推进。

# 选主

新增加模块ElectRunner支持运行整个选主流程，该模块是一个自主运行的后台任务，在CNODE\_LEADER\_ELECTION状态下需要的时候启动。

## elect runner流程

ElectRunner模块内部的工作流程主要会在以下几个状态之间流转，如下所示：



Leader：primary role

Follower：replica role

Candidate：unknow role

voter：storage node

投票过程中，每轮投票的时间间隔是一个10到30秒之间的随机值，避免在多个节点之间同时发起投票导致termID增长的过快。

## 存储节点投票

存储结点负责进行投票，需要在本地保存投票过程的文件，记录了每term投给了谁。文件的内容以二进制方式进行存储，数据内容为连续的8字节termID+4字节的NodeID。内存中保存了此投票过程的信息，首次开始投票时从文件中读出数据全部放入内存中，之后每次投票都先修改内存中的数据并落盘后才回复投票成功。此文件路路径为ps\_term/votes。

需要新增加投票的协议PS\_request\_vote，request中有NodeID和termID, response中有termID和accepted表示是否接受。同时此存储节点对此协议需要做特殊的处理，不判断termID是否有效。

## 计算节点选主流程

此过程的流程图如下：

* 某节点启动到成为主节点





* 某节点启动到成为从节点：





## 计算节同时选主过程

举例说明选主中会发生的某复杂情况，两个计算节点同时开始选主，在选主的过程中，需要进行协商。选主成功后会主动通知其他计算节点自己是主，并询问其他计算节点是否接受。接受则继续发送replica的startup信息，不接受是因为对方的termID比较大则等待对方发送其选主成功的信息，自己要切换为从节点。其中主要流程示例如下所示：



# 主从切换

主从之间的状态切换属于完成选举过程中必须步骤，主要按照以下步骤进行：

* 工作了一段的时间从节点需要变为主节点



当主节点掉线，某一个从节点经过选举后需要提升为主节点，此时需要做的步骤：

1. postmaster将状态切换为LEADER\_ELECTION
2. 停止到原主节点的心跳
3. 开始主动连接其他的计算节点
4. ElectRunner开始选主，若当选为主则继续以下流程，否则参考其他流程。
5. 结束运行中的Startup, RepWalReceiver
6. 开启StorageRecoveryWorker，开启WalDispatcher等任务
7. postmaster将状态切换为CLUSTER\_RECOVERY，
8. StorageRecoveryWorker完成存储节点的recovery
9. postmaster将状态切换为STARTUP状态
10. startup开始进行计算节点的recovery。从所有的存储节点查询最大的 NWL，和自己已经回放的NWL进行比较，将其中相差的部分读上来，并进行回放。
11. 若读取相差的xlog时失败，比如读取不到NWL的xlog则报错退出。
12. 若发现自己的NWL比所有存储节点NWL都要大，则认为自己超前回放的日志存储节点没有，则报错退出。
13. 重新加载ContolFile 等状态信息。
14. 可以开始给其他计算节点发送初始化信息，开始同步给别的节点发送PGCL等状态信息
15. postmaster切换到PM\_RUN状态。
16. 对已经连接的session不做额外的处理，并开始允许此session上进行写操作，但在恢复过程中session的进行任何操作都会等待。
17. 若在recovery中发生任何异常进程结束，此时集群中其他计算节点重新开始选主流程。

以上过程的流程图如下：





* 工作了一段的时间的主节点需要变为从节点：



当主节点发现已经连接上的从节点的termID比自己大，此问题发生于网络问题，导致一段时间主节点和所有节点断开，之后以前的某个从节点成为了新的主节点，或可能还在选举的过程中。此时网络恢复重新连接上其他的节点，需要将自己从主切换为unkown，重新参与选主。之后要变为从节点工作，因为以前内存里面已经有很多信息，并且已经有session连接在上面了，此时切换为replica需要做的步骤：

1. 将node\_role切换为replica，通知postmaster进入到CLUSTER\_RECOVERY状态。
2. 目前暂不考虑写转发的情况下，通知已经连入的进行过写操作的session回滚并退出，但对于只读session不处理。但在恢复过程中只读session的进行任何操作都会等待。
3. postmaster停止WALDispatcher, WALConsistencyManager, StorageRecoveryWorker, 开启StartUp， ReplicaWalReceiver。
4. 等待主节点发来的PGCL, max\_NWL等信息。
5. 开启到主节点的心跳信息heartbeat。
6. 进入到计算节点recovery状态
7. 比较本地的NWL和max\_NWL尝试从存储节点读取缺失的xlog日志，并回放。
8. 若在补齐日志的过程中失败，则报错退出。
9. 完成xlog补齐后，通知postmaster进入到PM\_HOT\_STANDBY状态。

以上过程的流程图如下：





# 冲突处理

主节点和从节点的各种操作很可能发生冲突，比如主节点想删除某表，但此时从节点正在访问此表；主节点vacumm正在清理某数据页，但是从节点正在访问此数据页；主节点对某对象加排他锁，但从节点也要操作此对象等等。

利用原有解决冲突的逻辑，主节点和从节点都在努力的解决冲突。

当前从节点活动事务列表在wal回放过程中被更新，当从节点开始事务时，创建的snapshot里面包括最小运行事务的id即xmin，从节点定期的收集此信息发送给主，为此RepWALRecevier需要定期的向主发送feedback, 当hot\_standby\_feedback开启后，每wal\_receiver\_status\_interval时间就发送。主收到此信息后更新backend 的当前事务的xmin，此后在进行操作时会根据此来避免冲突，比如vacuum更新时会跳过大于等于xmin的页。为此每个从节点需要对应主节点上的一个backend结构，并且其pid不能为0。

在从节点回放日志的时候会根据具体其中修改的涉及到的信息，比如涉及到冲突的最小事务id 即xmin，或正在删除的dbid, tabspaceid等信息，来找到对应从节点上正在进行操作并等待其完成，若一段时间max\_standby\_streaming\_delay从节点的操作还没有结束则通知其报错退出，或不等待直接通知其退出，等所有冲突的操作都退出后回放操作才会继续进行。锁、snapshot冲突会等待，其他的不会等待。

利用原有逻辑，需要做到每个从对于主有一个对应的backends，从定期的发送feedback。

# write forward

实现将从节点接收到写请求发送给主执行，从节点不仅仅支持read only的请求。

当在从节点上首次执行写操作时，需要生成事务ID，这个时候作为写转发的起始点，将该请求转发给主。主节点接收到该请求时会生成对应的backend为其进行服务，之后从将此会话当前事务的所有操作都转发给主操作，需要增加协议发送sql并接收结果集，当事务提交或回滚后对应的主上的backend将自动结束。

遗留问题：

1. 锁，xacts等一系列问题的优化
2. consistency read point是否可以去掉
3. 切主的时候recovery，保持session不断开
4. select for update 行级锁的实现
5. forward后续的读操作如何处理，参考pg\_fdw