



流复制中：同步和异步自动切换 --PostgreSQL Hot Standby模式下

姜瑞海

jiangruihai@highgo.com

山东瀚高基础软件股份有限公司

内容简介

- 简要介绍PostgreSQL流复制的工作原理
- 介绍流复制中：同步流复制和异步流复制的自动切换的实现。该方案满足了一些客户的需要。

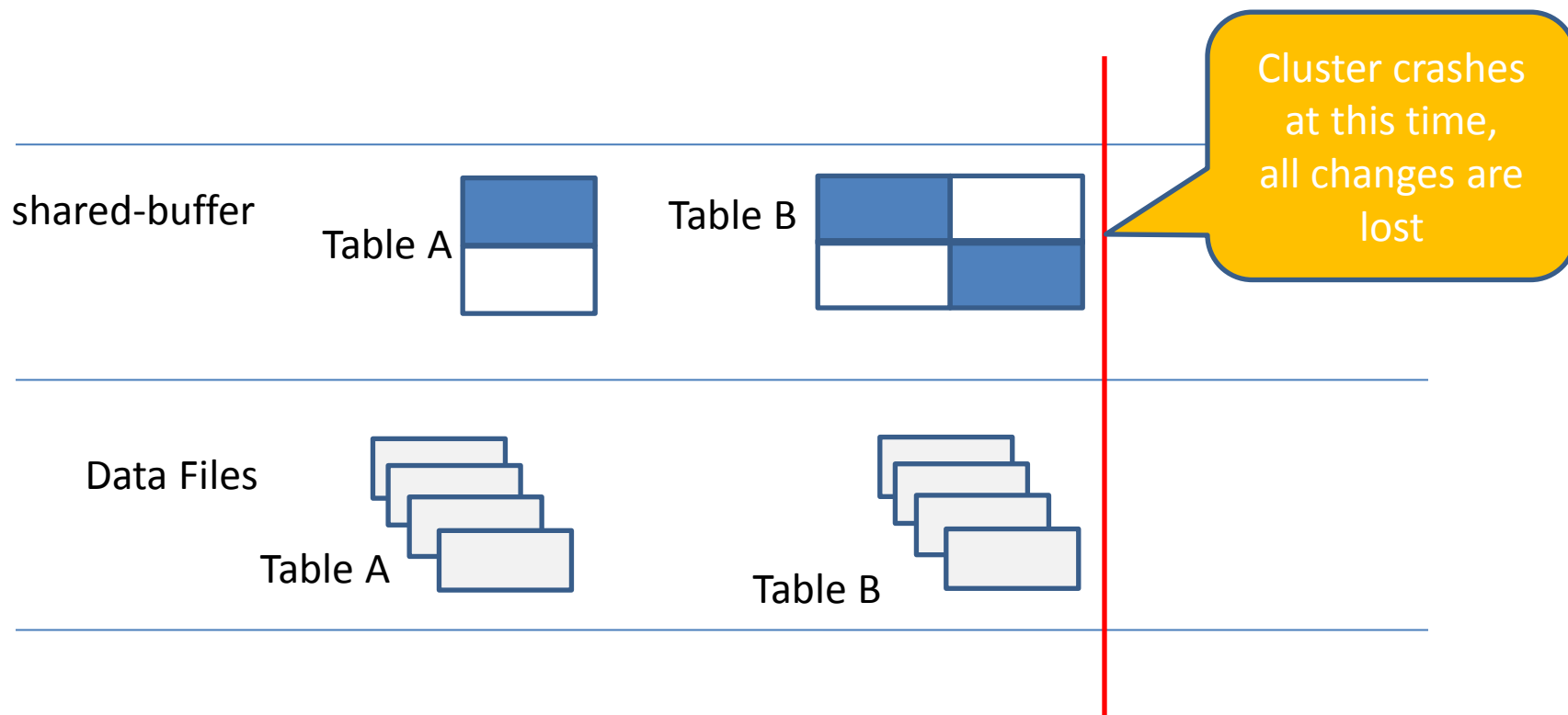


WAL: Write-Ahead Logging

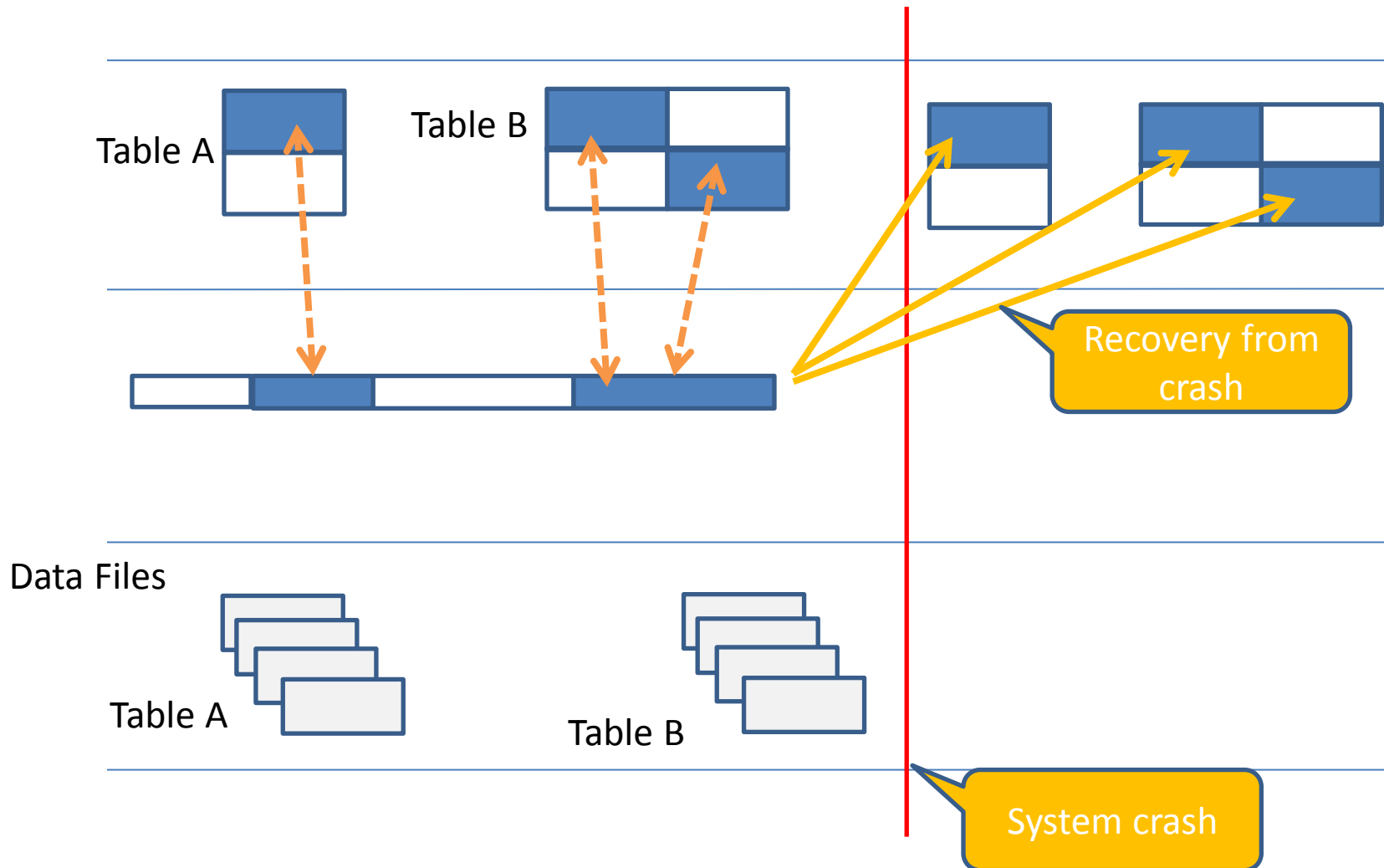
- 数据的变化首先在内存(shared buffer)发生，在后续的某个时刻，shared buffer里更改的数据被同步到数据文件中。
- 每一个数据的修改，都对应着至少一条WAL日志: WAL record。WAL record被写入WAL文件中。目录\$PGDATA/xlog/下产生很多WAL文件，每一个WAL文件叫做WAL segment。
- shared buffer里的数据更改被提交成功之前，以及同步到数据文件之前，其对应的WAL日志是先被写入WAL文件中。
- 当系统发生故障(例如掉电)，shared buffer中更改的数据，如果没有同步到数据文件中，会丢失。丢失的数据，能够从WAL日志文件中对应的WAL record恢复。



Data lost without WAL

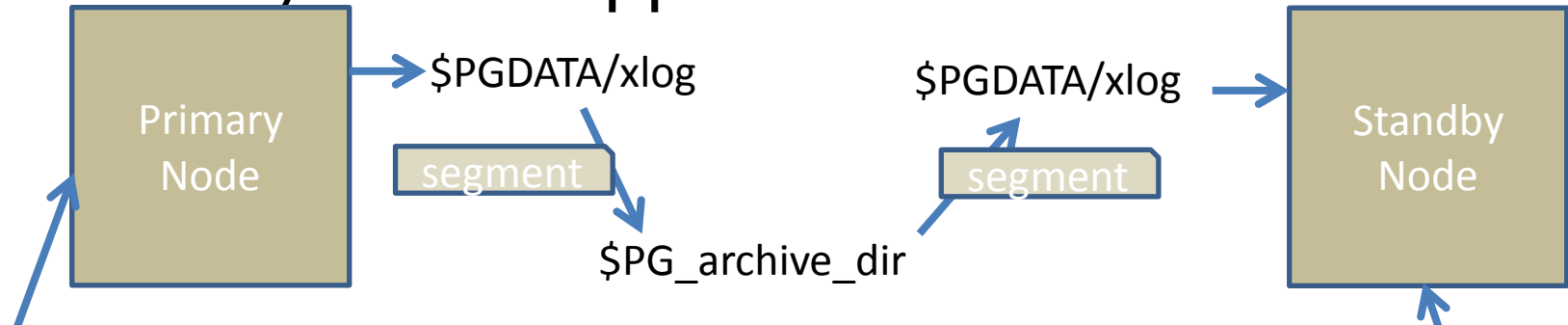


Data Recovery with WAL



PostgreSQL Warm Standby

- Primary node supports both read and write.
- Standby node supports neither read nor write.

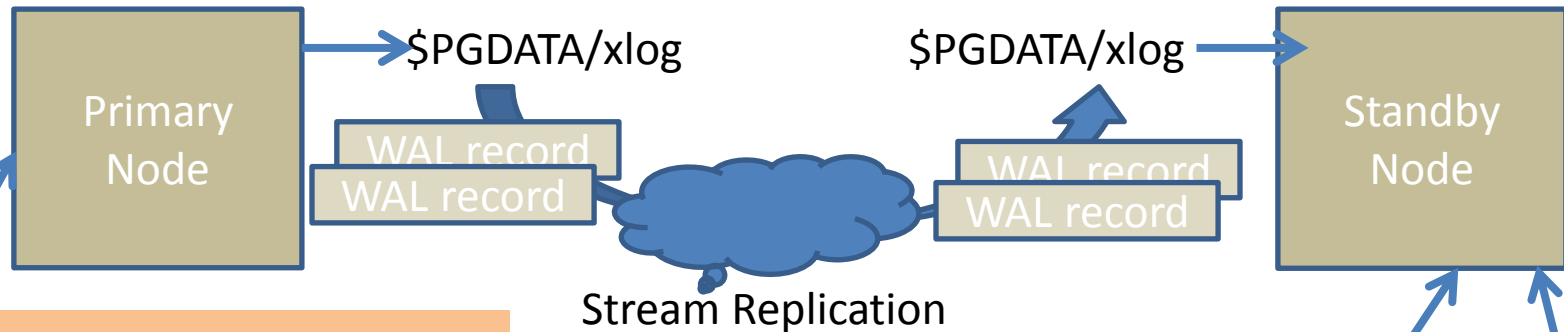


```
Primary node: postgresql.conf  
=====  
archive_mode = on  
archive_command = 'cp -i %p /shared/psql_wal/%f'
```

```
Standby node: recovery.conf  
=====  
standby_mode = 'on'  
restore_command = 'cp -f /var/lib/postgresql/data/archive/%f %p </dev/null'
```

PostgreSQL Hot Standby

- Primary node supports both read and write.
- Standby node supports read only.
- Use Stream replication to transfer WAL records.



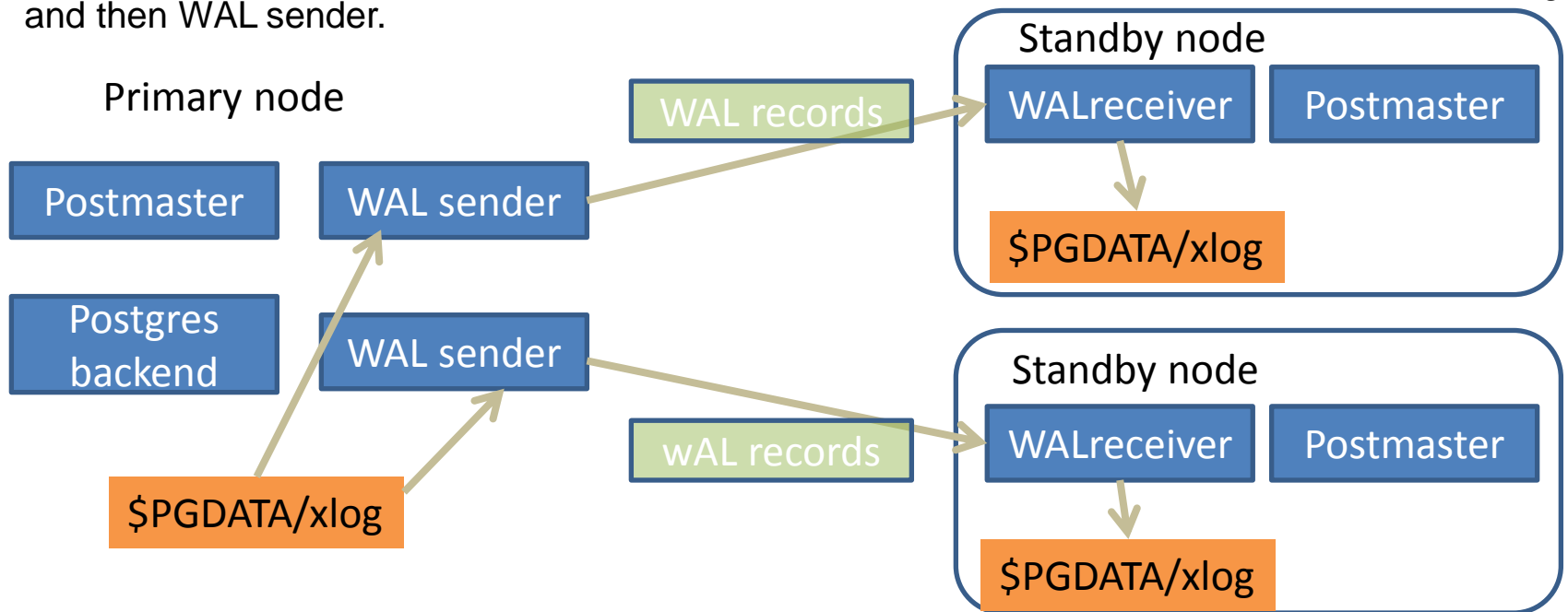
```
Primary node:postgresql.conf
=====
wal_level = hot_standby
max_wal_senders = 1
```

```
Standby node:postgresql.conf
=====
hot_standby = on
```

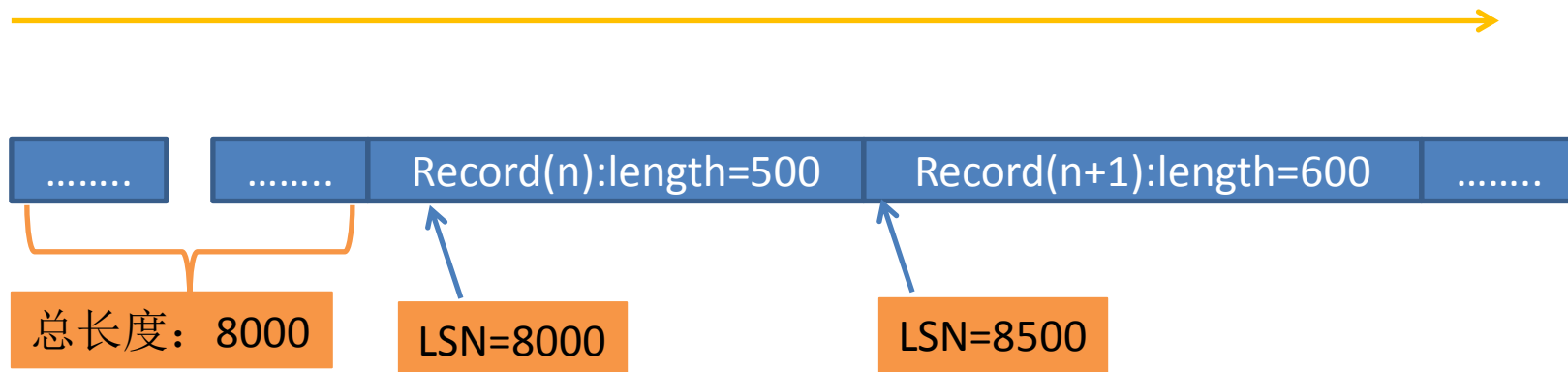
```
Standby node: recovery.conf
=====
primary_conninfo = 'host=192.168.0.1 port=5432 user=ruser password=password'
standby_mode = on
```

PostgreSQL Stream Replication

- Standby node: Recovery process notifies Postmaster to start the WAL receiver process.
- Standby node: Postmaster process starts the WAL receiver process.
- Standby node: WAL receiver process tries to connect to the Primary node.
- Primary Node: Postmaster accept the connection and spawn the WAL sender process.
- The WAL receiver and WAL sender begin to work.
- WAL receiver tells WAL sender the WAL location(LSN, Log Sequence Number) to start.
- WAL sender starts sending WAL records to WAL receiver.
- WAL sender and WAL receiver exit on communication timeout. WAL receiver will be started again and then WAL sender.



LSN(Log Sequence Number)示意

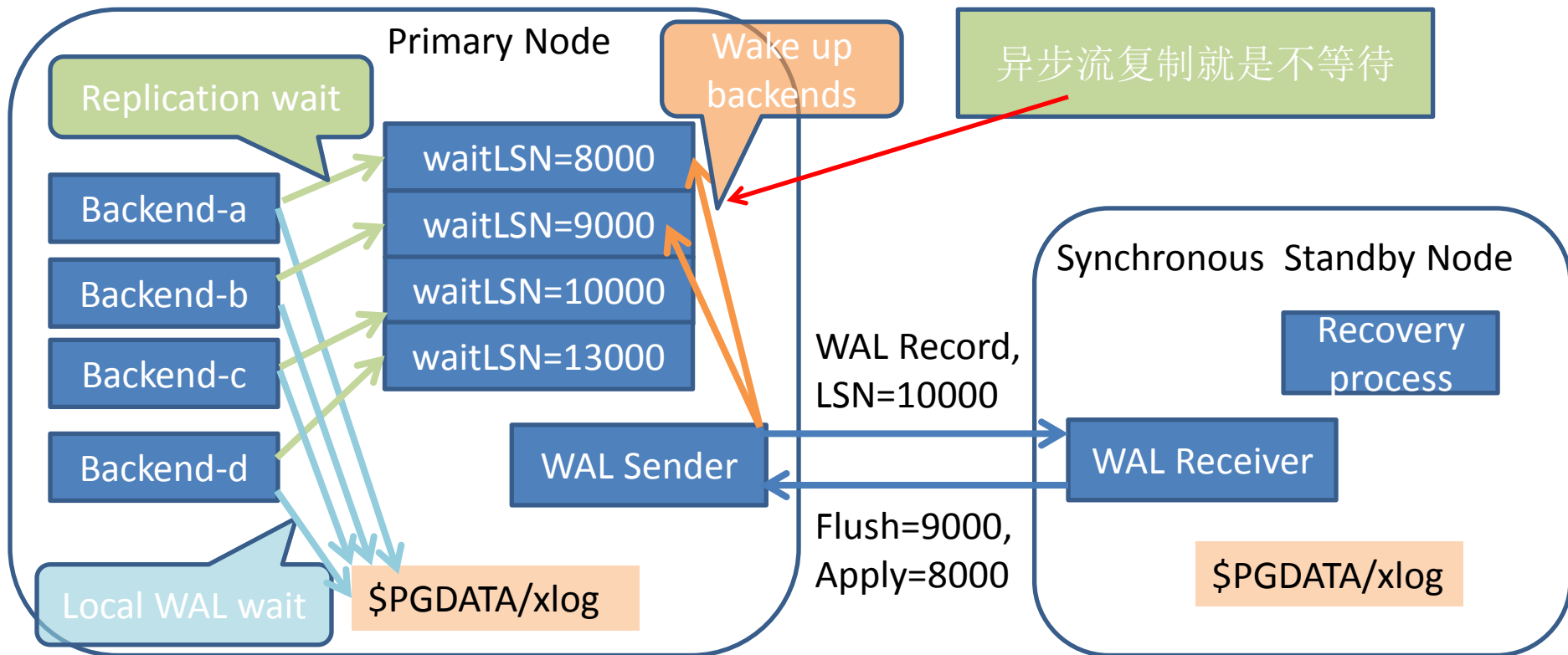


- LSN表示一个WAL record在WAL流中的偏移，如图所示。（实际的LSN计算比图示复杂）
- 每一个WAL record分配一个LSN。



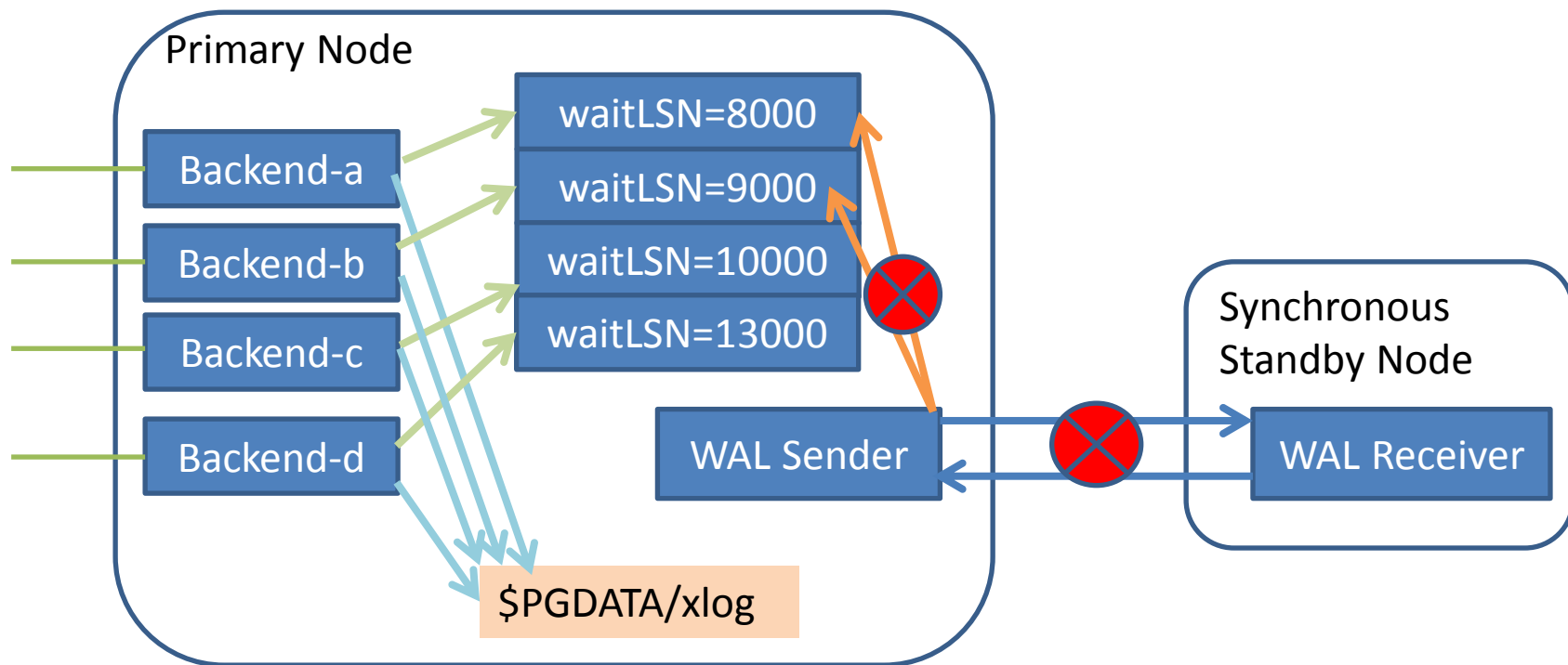
同步流复制：Synchronous Replication

- 同步流复制意味着Primary节点上，Postgres backend等待当前事务的WAL records被Standby接收并写入WAL日志文件或者被应用以恢复数据(replay)。这个等待使得Postgres backend阻塞，直到相应的LSN从standby应答。
- 当WAL sender收到standby应答的LSN，例如：10000，所有等待中的事务/backend，其waitLSN小于等于应答LSN，例如：8000,9000，会被唤醒。



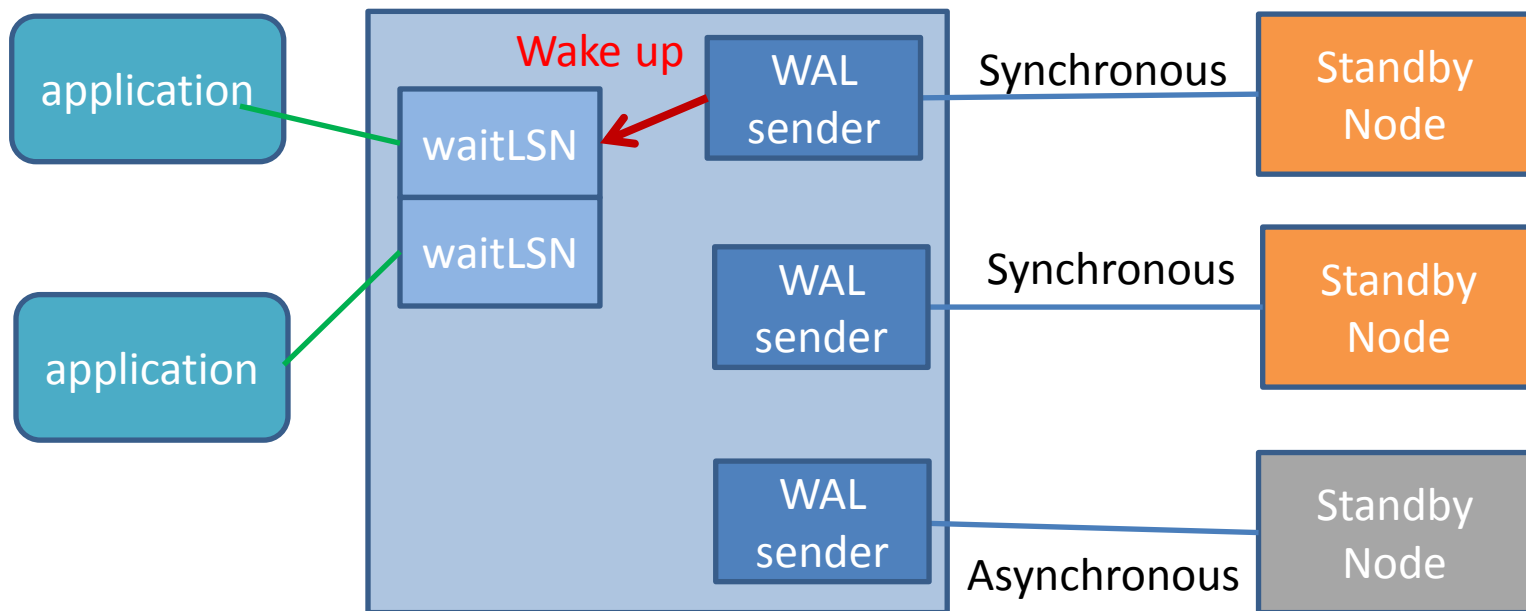
同步模式中：Standby 故障，导致 Primary 无法正常工作

- 当同步模式的standby崩溃，或者网络故障，导致WAL sender收不到任何来自WAL receiver的应答。
- 这会导致Primary端，没有任何Postgres backend会被唤醒，意味着进行中的事务被挂起，进而数据库客户端会超时，返回失败。
- 一些应用程序遇到这样的错误时，会不断放弃已有的数据库连接，而尝试新建连接。新建连接意味着新的Postgres backend进程不断产生，进而会耗尽系统资源。

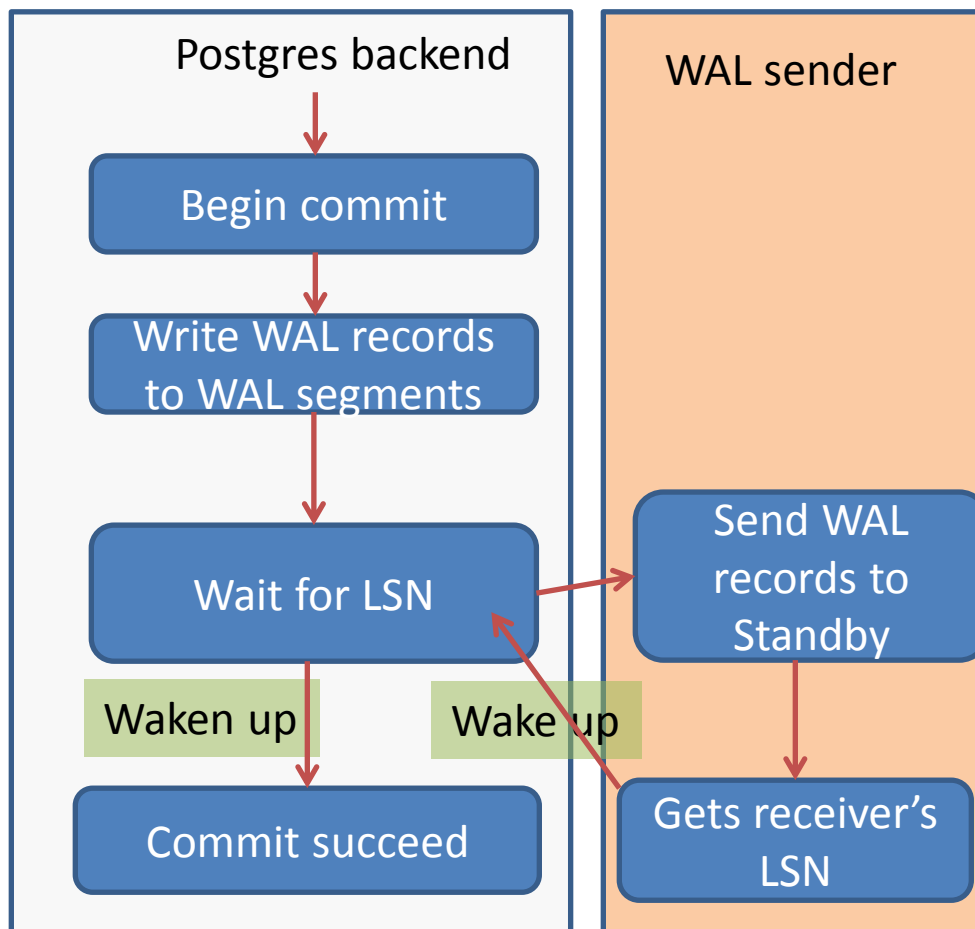


为什么需要同步/异步自动切换

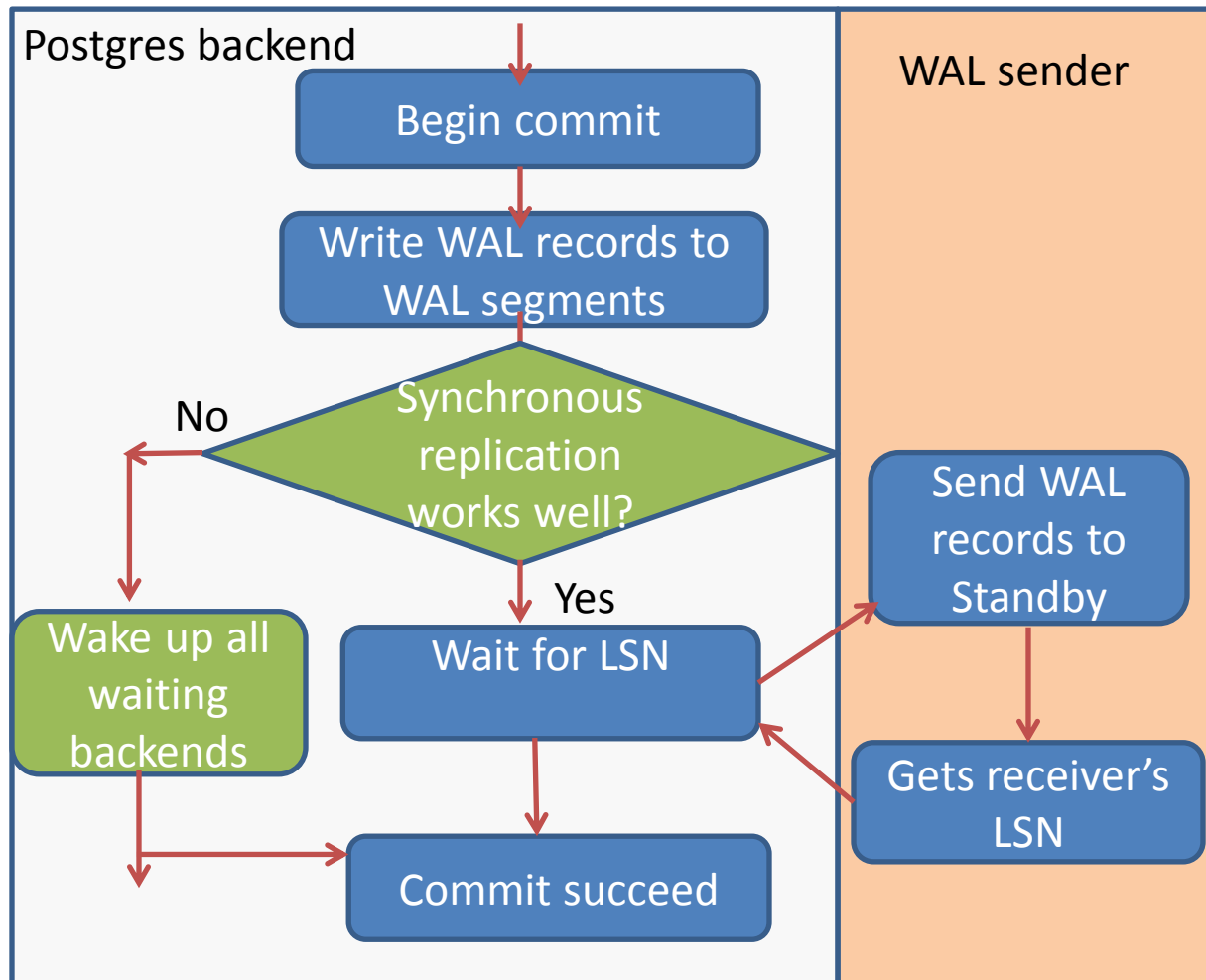
- 基于流复制的hot standby配置中，如果同步的Standby节点故障，会导致Primary node也无法工作。
- 我们的很多客户不接受。他们希望：Standby节点出现故障后，Primary节点能继续工作，整个应用程序应该能继续工作。**通过自动切换实现。**
- 有些人说：这是可以接受的。因为同步流复制的目的是为了Primary和Standby保持数据高度一致。一个出故障了，另外一个就不应该继续更改数据了。



同步模式的工作流程

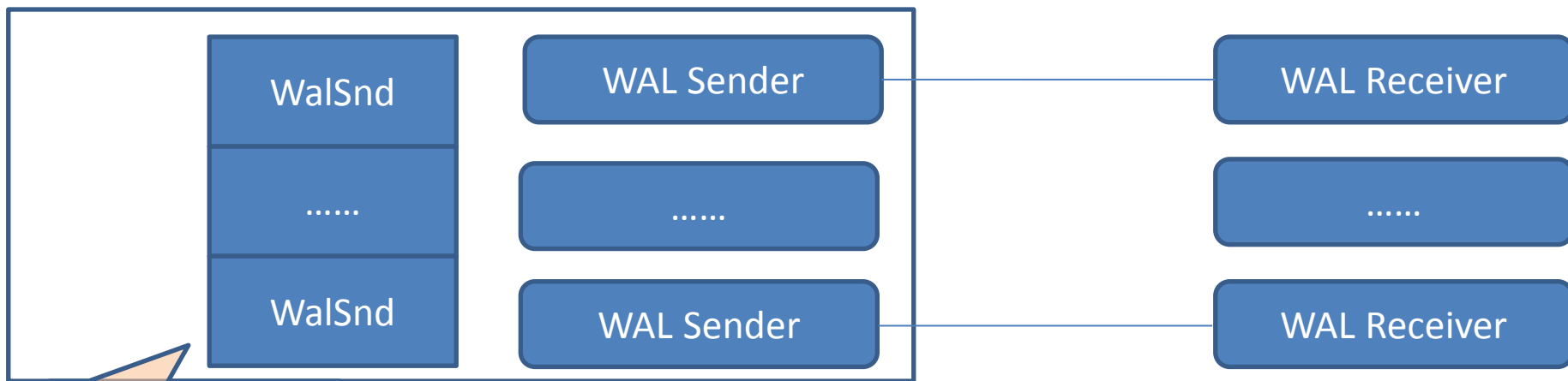


Switching between synchronous mode and asynchronous mode



同步流复制状态检查

- 流复制会话建立时，即新的WAL sender产生，对应的WalSnd.pid 设置成WAL sender的PID，非0。
- WAL sender通信超时，退出，其WalSnd.pid设置为0。
- 当所有同步的WalSnd.pid都为0时，**同步**流复制变为不可用。



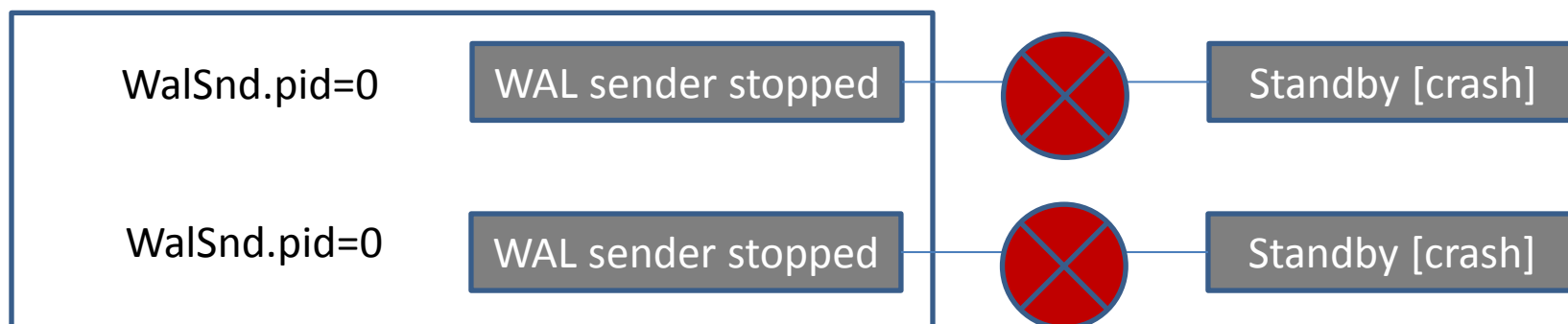
Shared memory

```
typedef struct WalSnd
{
    pid_t    pid;          /* this walsender's process id, or 0 */
    WalSndState state;    /* this walsender's state */
    XLogRecPtr sentPtr;   /* WAL has been sent up to this
point */
} WalSnd;
```



Synchronous to Asynchronous

- 条件检查和动作：如果WalSnd[...]中，**所有**同步模式的pid都是0，则唤醒所有等待中的事务，进入异步模式。
- 每次提交事务，做上述检查。
- 每当流复制通信超时，关闭该会话，WAL sender退出，做上述检查。



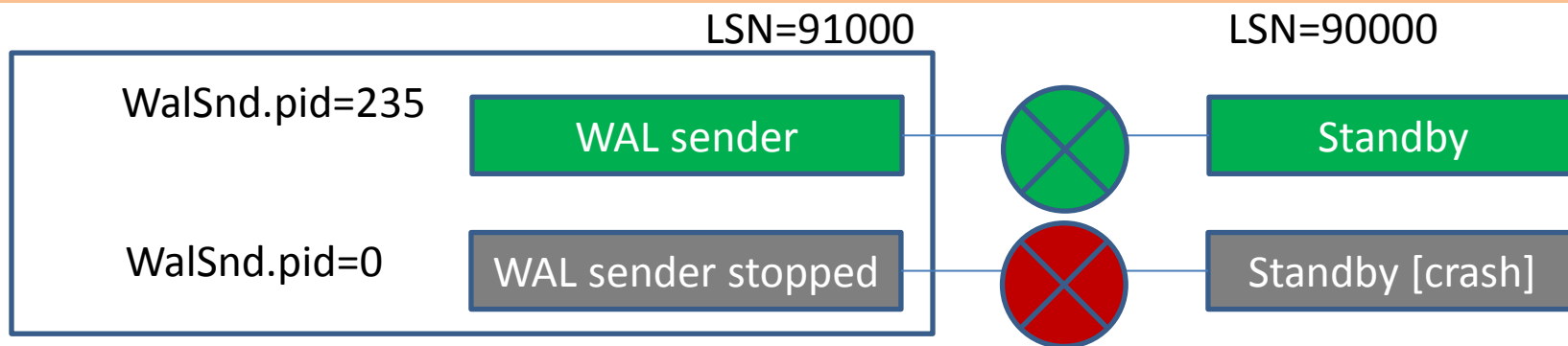
所有同步模式的standby都出故障了



Asynchronous to Synchronous

- 条件检查和动作：当至少一个同步的Standby连接上来。即：当一个WAL receiver发送请求，建立连接，对应的WAL sender进程产生，其对应的WalSnd.pid变为有效值（非0）。
- 如果刚刚连接上来的standby落后的WAL日志如果太多，还不能变为同步模式，必须等待该Standby通过流复制，追平Primary WAL日志。WAL sender和对应的WAL receiver之间的LSN差小于配置参数：
adaptive_sync_repli_lsn_catchup=8192。

- 每次提交事务，做上述检查。
- 每当一个新的WAL sender运行，做上述检查。
- 每当一个同步的WAL sender收到WAL receiver的应答。



至少一个同步Standby恢复了，并且LSN追上来了



实现细节:1

postgresql.conf

=====

`adaptive_sync_repli_flag=on/off` #这个开关控制一切功能，只有这个开关
#打开，新增功能才工作，否则，所有新增代码都不运行。

`adaptive_sync_repli_lsn_catchup=8192` #有异步切换到同步时，standby落后的LSN要
小于这个值

共享内存中的变量

=====

```
Bool adaptive_sync_repli_flag=true/false #对应配置参数adaptive_sync_repli_flag。
Int adaptive_sync_repli_lsn_catchup=8192 #对应配置参数adaptive_sync_repli_lsn_catchup。
Int adaptive_sync_repli_status=(adaptive_sync, adaptive_async, disabled)
WalSnd {
    XLogRecPtr standbyLSN; //WAL receiver收到的最新新的LSN，WAL sender就能知道
    Standby落后了多少LSN.
}
```



实现细节:2

SyncRepWaitForLSN(XLogRecPtr XactCommitLSN){

SyncRepWaitForLSN()进入等待前的检查

```
    如果adaptive_sync_repli_flag=true
    {
        检查所有的同步WAL sender, WalSnd[]
        如果存在至少有一个同步WAL sender,WalSnd.pid非0
        {
            if(adaptive_sync_repli_status==adaptive_sync/*自适应同步模式*/) {
                //本来就是同步模式,不做什么, SyncRepWaitForLSN()将执行旧的逻辑
            } else if(adaptive_sync_repli_status==adaptive_async/*自适应异步模式*/) {
                if( ( myLSN-WalSnd.standbyLSN )<adaptive_sync_repli_lsn_catchup ) {
                    adaptive_sync_repli_status==adaptive_sync; //异步到同步,
                    //SyncRepWaitForLSN()将执行旧的逻辑
                } else {
                    //仍然处于自适应异步模式
                    wakeup_all_transactions(); //唤醒所有等待中的backend进程
                    return; //不执行SyncRepWaitForLSN()将执行旧的逻辑
                }
            }
        }
        否则, 如果没有同步WAL sender存在
        {
            if(adaptive_sync_repli_status==adaptive_sync/*当前是自适应同步模式*/)
            { //同步变异步
                wakeup_all_transactions(); //唤醒所有等待中的backend进程
                adaptive_sync_repli_status==adaptive_async
            }
            return;//不执行SyncRepWaitForLSN()将执行旧的逻辑即不等待。
        }
    }
}
```

//SyncRepWaitForLSN后续正常处理

....
....
....

}



实现细节:3

如果adaptive_sync_repli_flag=true

```
{  
    检查所有的同步WAL sender  
    如果没有其它同步WAL sender存在  
    {  
        if(adaptive_sync_repli_status==adaptive_async/*自适应异步模式*/) {  
            //如果当前是异步模式，则有可能切换到异步模式  
            if( ( myLSN-WalSnd.standbyLSN )<adaptive_sync_repli_lsn_catchup ){  
                adaptive_sync_repli_status==adaptive_sync; //异步到同步，  
            }  
        }  
    }  
}
```

WAL sender进程产生，并开始工作时，做检查。
异步切换到同步。



实现细节:4

WAL sender进程退出之前，做检查。
同步切换到异步。

```
如果adaptive_sync_repli_flag=true
{
    检查所有的同步WAL sender
    如果没有其它同步WAL sender存在
    {
        if(adaptive_sync_repli_status==adaptive_sync/*自适应同步模式*/) {
            //如果当前时同步模式，则切换到异步模式
            adaptive_sync_repli_status=adaptive_async;
            唤醒所有等待中的backend进程
            wakeup_all_transactions();
        }
    }
}
```



Thanks!

Q & A

山东瀚高基础软件股份有限公司

