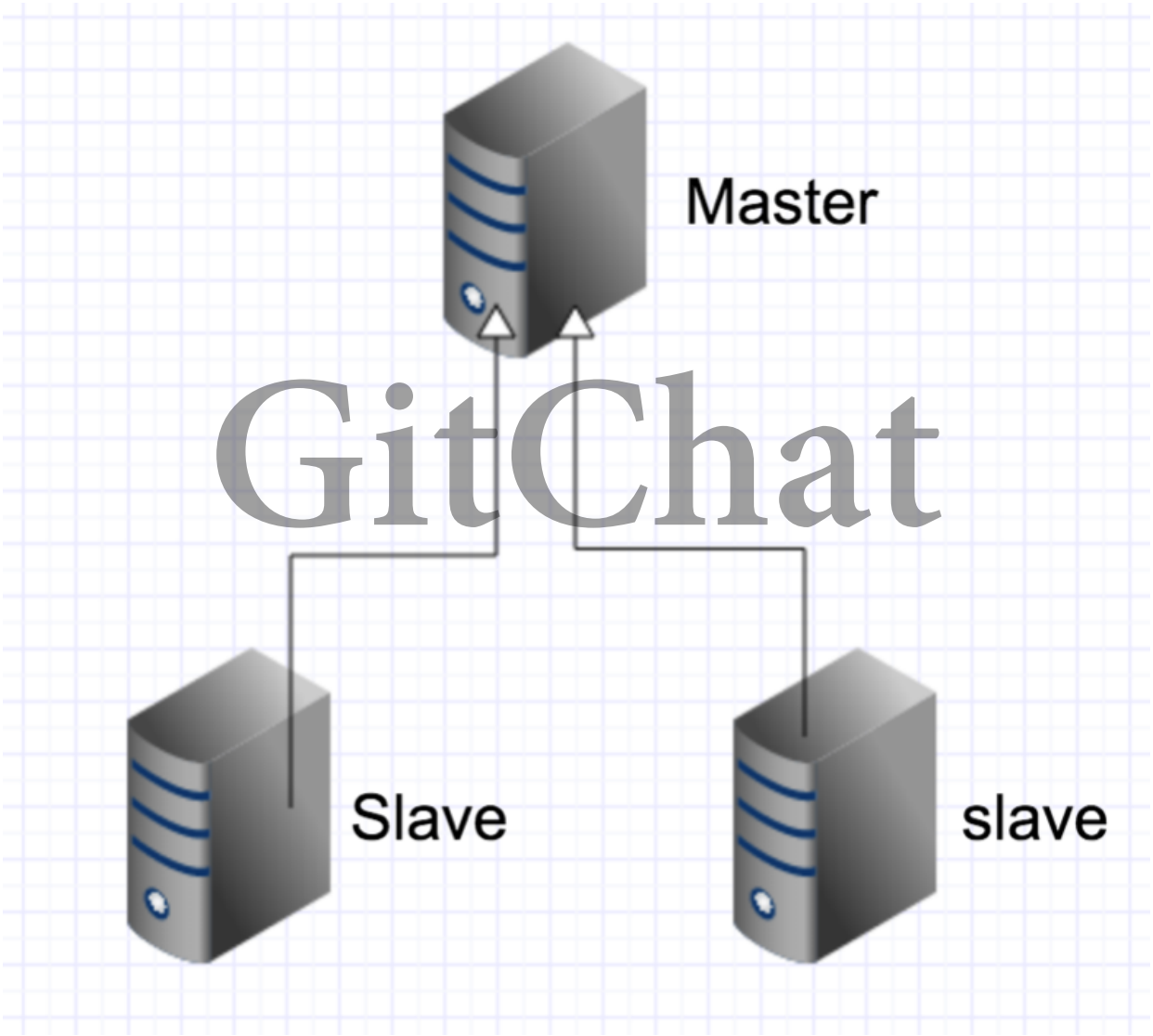
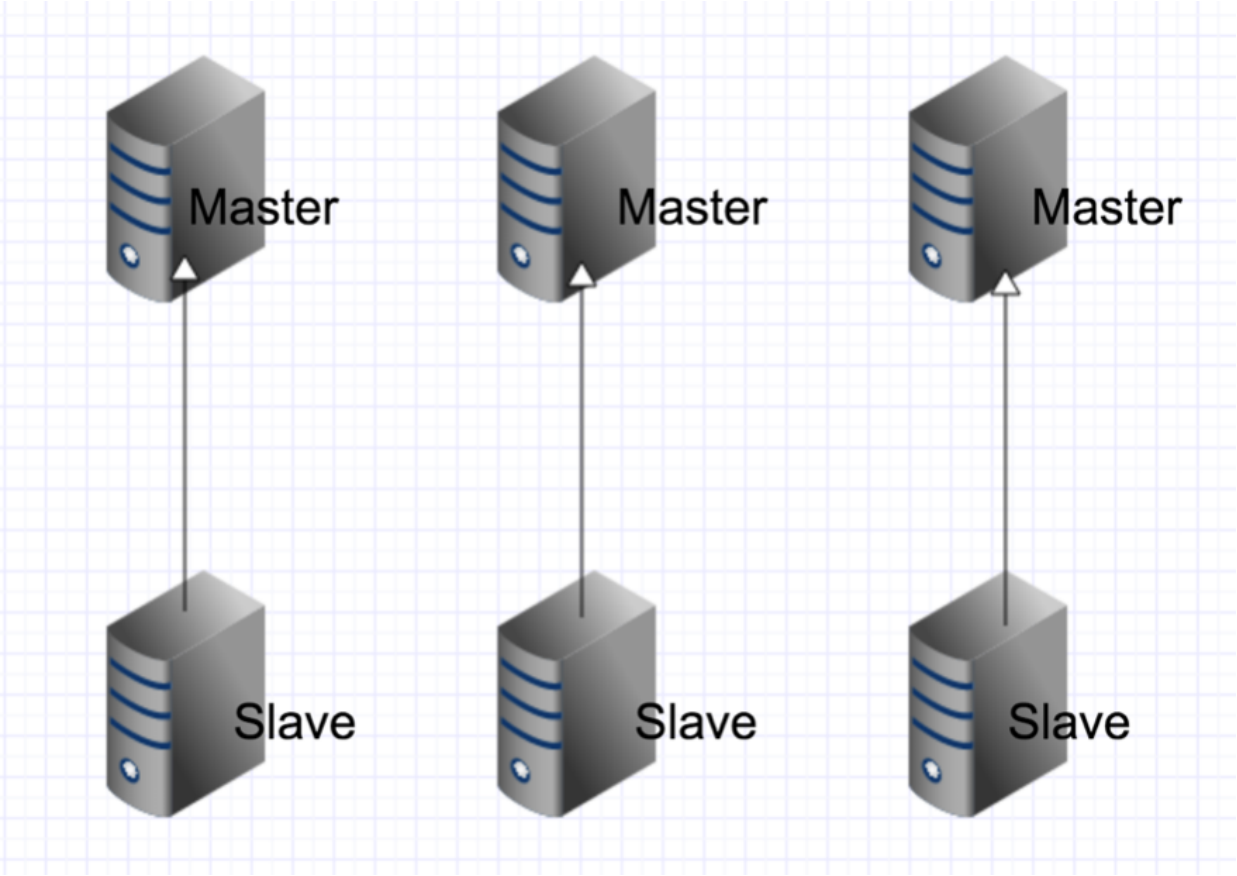
饿了么 PostgreSQL 优化之旅

1. 架构演变

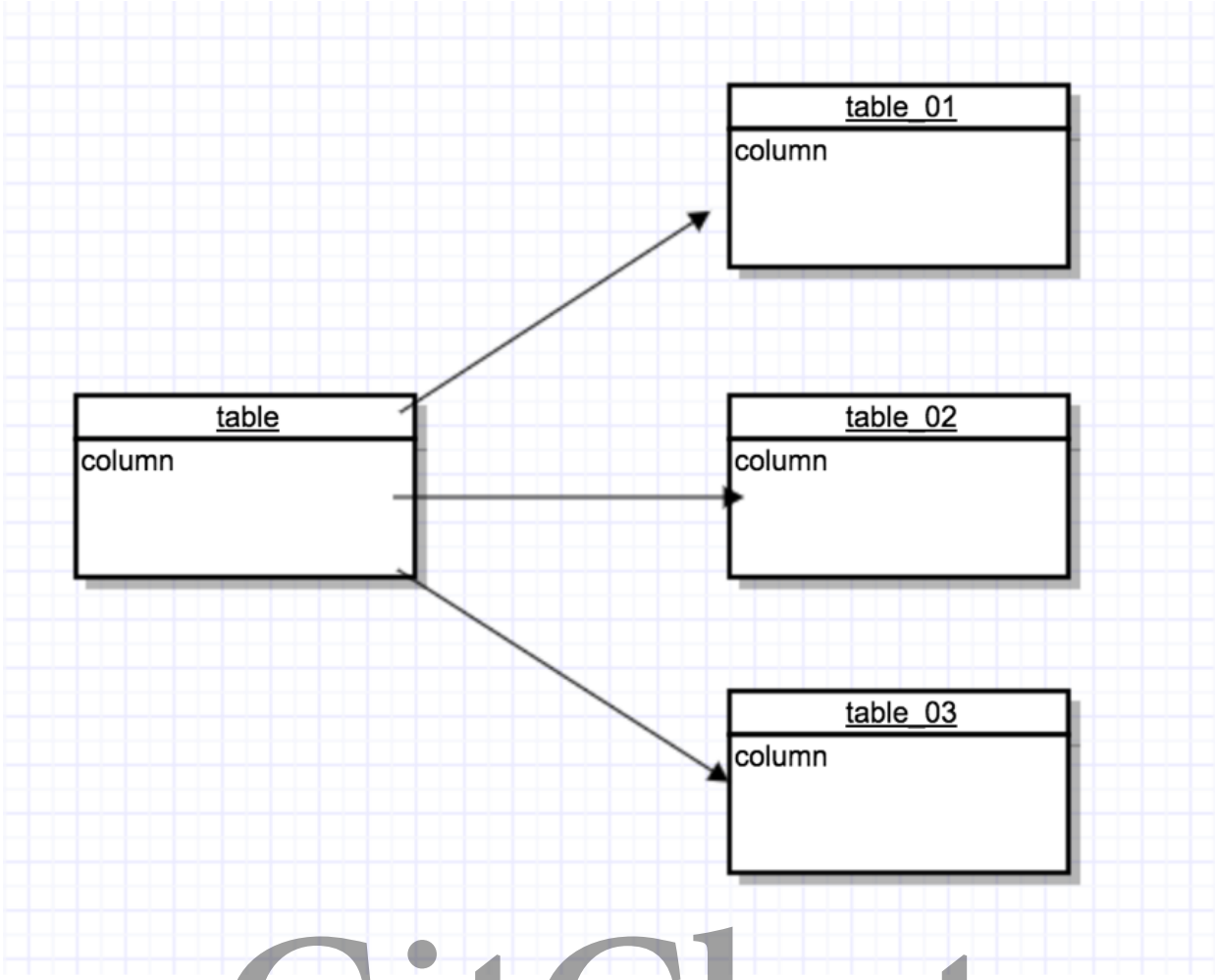
在O2O外卖领域，基于位置服务的需求越来越多，这就要求DB能够存储地理位置信息，而在开源数据库中，对空间地理数据支持比较好的要数PG的插件Postgi。



饿了么在使用PG的过程中，由于性能及容量的原因，DB结构也在不断发生变化。 在刚开始使用PG时，公司使用的是最简单的结构一主两从，读写分离，Master负责写，Slave负责读，一切都是那么快乐的运行着。 但过了一段时间，随着公司业务量的扩大，单台数据库的写遇到了瓶颈，所以需要对DB进行拆分，在水平拆分与垂直拆分中，垂直拆分是相对简单的，由于饿了么业务与地理位置相关性很大，自然想到了根据地域进行切分，结构如下：



经过拆分把单一Master， 拆分成多个Master， 此时数据库的写已不是瓶颈。 世间万物总是在不断变化，饿了么每天几百万的订单量，相关的地理位置信息数据达到几百GB的真是轻轻松松， 有时DBA不得不进行一些系统维护，比如导出数据，做历史归档，DDL变更等等，但如果一个表数据有100G+的时候，那些操作想想都是头疼，此时不得不对表进行水平拆分，把大表变成小表，结合业务，饿了么对数据时效性要求比较强，故我们采用每天一个轮询表的方式进行水平拆分，结构如下：



经过以上DB架构演变，后来我们还对历史数据做了归档处理等，至此，目前PG已能支撑饿了么对数据库的要求。

2. “坑”与优化

2.1 Disk queue与checkpoint

我们DBA在运维PG的过程中，有一段时间内总是不定期的观察到磁盘的disk queue有大量的等待，磁盘的IOPS也很高，后来通过日志发现记录有”checkpoints are occurring too frequently”，再结合checkpoint的时间点发现当时1分钟有70+个wal日志文件，1分钟写那么多的数据，当然会有disk queue了，可这是为什么呢?

大家知道，PG中也有和Oracle一样的checkpoint，其作用如下两点：

1. 保证数据库的一致性，这是指将脏数据写入到硬盘，保证内存和硬盘上的数据是一样的；
2. 缩短实例恢复的时间，实例恢复要把实例异常关闭前没有写出到硬盘的脏数据通过日志进行恢复。如果脏块过多，实例恢复的时间也会很长，检查点的发生可以减少脏块的数量，从而提高实例恢复的时间。

由以上可知checkpoint刷新脏数据到硬盘时，会导致PG把shared buffer中的dirty buffer刷新到磁盘上，试想一下如果 shared_buffer=30G ，如果dirty buffer占5%，则PG也要把1.5G的数据从内存刷新到磁盘，在业务高峰期，实际上远大于1.5G，因为要设置 full_page_writes=on (为了防止数据文件损坏，不得不把这个参数设置为on)，每一次checkpoint之后对每一个数据页的第一次修改都会导致在WAL日志中整页的写入。 现在是一次写入数据量太大， 但checkpoint又无法避免， 那有没有办法使PG一次不要刷新太多的数据，通过时间换速度，有的，可以通过如下三个参数控制：

```
checkpoint_segments=256
checkpoint_timeout=30min
checkpoint_completion_target=0.9
```

其中 checkpoint_segments 和 checkpoint_timeout 两个参数控制在什么条件下会发生checkpoint ，如上，则说明每当写入256个wal日志或者每30分钟间隔则发生一checkpoint， checkpoint_completion_target 则说明在两个checkpoint之间多长时间内完成，比如两个checkpoint之间有30分钟，当checkpoint_completion_target=0.9，则30*0.9=27分钟完成刷新脏数据到磁盘，其值越大，对IO压力越小。

调整之后disk queue已很少出现，日志输出如下：

```
2016-10-12 20:05:21.558 CST,,,156801,,57fcf9c7.26481,8421,,2016-
10-11 22:40:07 CST,,0,LOG,00000,"checkpoint complete: wrote 590555
buffers (9.2%); 0 transaction log file(s) added, 0 removed, 457
recycled; write=3239.416 s, sync=0.021 s, total=3239.905 s; sync
files=119, longest=0.004 s, average=0.000 s",,,,,,,,,""
```


2.2 表膨胀

在维护PG的过程中，表膨胀是无法避免的问题，在我们系统中有一张表每天要更新几千万次，表数据大概有10w+，但表所占磁盘空间却有10G+，这明显是不正常的，后来我们做了一次vacuum发现表占用空间大小一下子下降到了几十MB(vacuum并非像网上说的那样不能回收表所占的空间，当回收的页处于存储数据的文件尾部，并且页内没有事务可见的tuple（即整个页都可以删除）时，会做truncate操作，把尾部的这些页统一从文件中删除，文件大小和表所占空间随之减少)，由以上可知表中有太多的dead tuple。

```
db_name=# vacuum verbose t1;
INFO:  vacuuming "public.t1"
INFO:  scanned index "t1_pkey" to remove 6646578 row versions
DETAIL:  CPU 0.04s/0.29u sec elapsed 0.34 sec.
INFO:  scanned index "uk_t1_type_id" to remove 6646578 row versions
DETAIL:  CPU 0.03s/0.28u sec elapsed 0.32 sec.
INFO:  scanned index "gix_t1_location_geog" to remove 6646578 row versions
DETAIL:  CPU 1.31s/6.28u sec elapsed 7.60 sec.
INFO:  scanned index "idx_t1_created_at" to remove 6646578 row versions
DETAIL:  CPU 0.14s/1.90u sec elapsed 2.05 sec.
INFO:  scanned index "idx_t1_updated_at" to remove 6646578 row versions
DETAIL:  CPU 0.21s/2.88u sec elapsed 3.10 sec.
INFO:  "t1": removed 6646578 row versions in 25419 pages
DETAIL:  CPU 0.00s/0.20u sec elapsed 0.21 sec.
INFO:  index "t1_pkey" now contains 93901 row versions in 5898 pages
DETAIL:  878272 index row versions were removed.
1381 index pages have been deleted, 1265 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "uk_t1_type_id" now contains 94184 row versions in 4824 pages
DETAIL:  532758 index row versions were removed.
1092 index pages have been deleted, 541 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "gix_t1_location_geog" now contains 100733 row versions in 537294 pages
DETAIL:  6646578 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.60u sec elapsed 0.60 sec.
INFO:  index "idx_t1_created_at" now contains 103533 row versions in 86384 pages
DETAIL:  6646578 index row versions were removed.
59641 index pages have been deleted, 58311 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "idx_t1_updated_at" now contains 109136 row versions in 108020 pages
DETAIL:  6646578 index row versions were removed.
93923 index pages have been deleted, 75980 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "t1": found 52316 removable, 93751 nonremovable row versions in 26874 out of 80244 pages
DETAIL:  13065 dead row versions cannot be removed yet.
There were 1072701 unused item pointers.
0 pages are entirely empty.
CPU 1.90s/12.71u sec elapsed 14.63 sec.
```


首先，我们看一下表膨胀会造成什么问题呢?

1. 表膨胀，会造成数据膨胀，占用大量磁盘空间
2. 表膨胀会引用索引跟着膨胀(除非重建索引)
3. 表膨胀会引用SQL查询效率低下
4. 表膨胀会使PG进行vacuum时使用更多的CPU，IO，内存，进而引起整个DB性能下降

那是什么原因引起表膨胀呢，这要从PG的MVCC说起，PG为了实现多版本并发控制，当PG在更新数据时， 是不直接删除老数据的，一个update操作执行后，被更改的数据的旧版本也被保留下来，当PG进行删除数据时，也不是直接删除而是标记删除，那些旧版本数据被称为dead tuple。 那有没有办法回收那些被标记不再使用的数据呢，有的，PG提供了autovacuum，vaccum操作，当对表做vacuum操作的时候，才考虑回收，旧版本不及时回收就会造成表膨胀。

```
与autovacuum相关的参数:
autovacuum_work_mem = -1 # autovacuum所能使用的内存大小，当其为-1时，使用maintenance_work_mem参数的值，值越大，使用的内存越多
autovacuum = on # 是否打开autovacuum
autovacuum_max_workers =3 # 最多能够有多少个autovaccum进程运行，值越大，使用的内存越多
autovacuum_naptime = 1min  # autovacuum进程间隔多长时间对表进行是否需要autovacuum操作
autovacuum_vacuum_threshold = 50 # 当表上dml操作达到多少行时执行
```

```
autovacuum操作
autovacuum_analyze_threshold = 50  # 当表上dml操作达到多少行时执行
autovacuum analyze操作
autovacuum_vacuum_scale_factor = 0.2 # 当表上dml操作达到多少比例时执行
autovacuum操作
autovacuum_analyze_scale_factor = 0.1  # 当表上dml操作达到多少比例时执
行autovacuum analyze操作
autovacuum_vacuum_cost_limit = -1  # autovacuum 的cost超过此值时，
vacuum会sleep一段时间，使用vacuum_cost_limit参数的值，值越大对系统IO压力越
大
```

2.3 表膨胀原因

2.3.1 长事务

前面已经说了，由于PG的MVCC的原因，PG为了保证数据对事务的可见性，dead tuple是否能够被vacuum回收，要看当前系统里中是否有正在进行的事务需要查看dead tuple，如果需要则不会进行回收dead tuple，因而长事务会放大dead tuple数量，可以通过如下语句长询是否有长事务：

```
 select * from pg_stat_activity where state<>'idle' and
pg_backend_pid() != pid and (backend_xid is not null or
backend_xmin is not null ) and extract(epoch from (now() -
xact_start))  > 60s order by  xact_start;
```


一般情况下长事务结束，autovacuum就可以对dead tuple进行回收，从而避免表膨胀。

2.3.2 事条回绕

在我们DBA实际运维PG过程中发现DB中并没有长事务，也引起了表膨胀，这就很让人费解了，后来通过操作系统进程发现有如下进程：

```
autovacuum: VACUUM public.tb_20160919 (to prevent wraparound)
autovacuum: VACUUM public.tb_20160916 (to prevent wraparound)
autovacuum: VACUUM public.tb_20160915 (to prevent wraparound)
```

DB中已经有3个autovacuum进程在运行，前面我们说过 `autovacuum_max_workers` 决定了DB中最多可以有多少个autovacuum并发运行，其默认为3个，而现在刚好有三个autovacuum进程。 没有多余的autovacuum进程可以被调用，此时即使有hot table进行了大量的update或delete，有很多的dead tuple需要进行vacuum操作，但因为没有autovacuum进程可用，则也不得不进行等待，所以hot table表的膨胀会越来越大。

但是上图中三个表的数据在我们实际生产环境中，都没有再更新过，为什么还是需要autovacuum呢?这就与to prevent wraparound(防止事务回绕)相关了，因为PG的版本号是uint32的，是重复使用的，也即40多亿，如果一个表经历了40亿次事务操作就会溢出回绕，发生溢出回绕后数据表本身记录的事务号大于当前的系统事务号，会造成过去出现的似乎是来自于未来，因此需要一个措施来防止这种情况，PG会在表的事务号达到最大

xmin一半的地方（20亿）的时候就要强制标记所有行为“冻结”（freeze），如果没清理（freeze）就会夯住数据库。清理需要扫描表的所有行并更新行的xmin为2，然后更新pg_class的relfrozenxid为实施这次操作的事务号，由此即使表的xmin字段永远无法达到最大值发生回绕。对于一个高并发，且数据量大，事务小的系统而言，PG为了防止事务回绕而把autovacuum进程占用也不足为奇了。

那是不是调大了 `autovacuum_max_workers` 参数就可以防止hot table的表膨胀了呢?其实未必，调大了 `autovacuum_max_workers` ，只是增加了hot table被vacuum的概率，试想如果系统中有很多的大表，都要被to prevent wraparound，那hot table也有可能因为autovacuum进程可用，继续膨胀。而且调大 `autovacuum_max_workers` 参数可能需要更多的内存。后来还是写了一个脚本在凌晨对表手动执行vacuum维护。

2.3.3 Slave节点引起的表膨胀

```
postgres=# \c t
INFO:  "t": found 6587453 removable, 23556192 nonremovable row versions in 461962 out of 461973 pages
DETAIL:  23513371 dead row versions cannot be removed yet.

postgres=# select count(*) from t;
-[ RECORD 1 ]
count | 47046
```

在公司实际使用DB时一般都会设置一个delay节点，以备操作数据时恢复用，比如设置delay slave延迟master 12小时，则当在master上误操作造成数据丢失时，再通过修改delay的延迟时间到操作之前的那个点，进而快速恢复数据(通过备份恢复也可以，只是有点慢了)。PG 9.4版本开始提供支持delay节点，通过在slave的recovery.conf文件中指定 `recovery_min_apply_delay` 参数设定延迟时间。

Delay节点虽好，但如果使用不当，则会引起master上表的膨胀，比如当 `hot_standby_feedback=on` 时，slave节点会向master反馈当前查询的状态，如果是级联环境，则会反馈给最上层的主库，至于slave为什么需要反馈查询状态，其原因与长事务的原理一样，试想当Delay节点设置延迟12小时时，则可能会造成master节点上表有12个小时的膨胀，对于一个高并发的数据库，表膨胀几GB是很正常，如上图就是一个delay节点引起master膨胀的例子，表实际数据只有5w左右,但膨胀数据则有2000w+。那能不能避免呢?其实对于delay节点来说可以把这个参数关闭，因为一般delay节点也不用于查询。类似的还有 `max_standby_streaming_delay` ， `max_standby_archive_delay` 。但如果备库要被用于只读，在有大的查询的情况下，此时可以在session级别动态设置那些参数的值，而不宜设置为global级别。

3. 小结

在我们使用数据库过程中，前期也许只是简单的结构，但随着业务的扩张，数据量的增大，DB也在不断的演化，无论是架构变化还是性能优化，都是为了能够支撑业务的需求。每一种数据库都有各自的优点与缺点，期望我们能够扬长避短，发挥优势，避免采”坑”。