1. 背景

现如今，许多云数据库都采用存算分离的结构，如Amazon Aurora、HuaWei Taurus等。计算节点运行数据库程序，存储节点通过装载NFS等网络文件系统暴露出文件系统接口供计算节点使用。在这种存储与计算解耦的架构下，网络成为了新的瓶颈。Buffer Pool中频繁的写操作所造成的写放大加剧了问题的严重性。如何优化网络带宽，减少节点之间数据传输是数据库领域所面临的新的难题。

1.1 Buffer Pool与写放大

Buffer Pool是大多数OLTP数据库用来提升数据库读取性能的重要组件。InnoDB将数据组织成页持久化在磁盘上，为了缓解内存与磁盘访问速度的巨大差异，一部分数据页会被缓存在内存中并由Buffer Pool Manager进行管理。这些缓冲页面被前台线程共享访问。Buffer Pool Manager的目标是将频繁使用的页面尽可能久的缓存在内存中，而很少被访问的页面会很快被淘汰掉。为了最大化利用数据访问的空间局部性，在有限的内存中提高Buffer Pool的命中率，Buffer Pool Manager将数据页以缓冲帧（Buffer Frame）的形式组织成链表，并使用缓存替换算法（如LRU、Clock等）来管理缓冲帧链表。InnoDB使用了LRU算法的变体，可以将缓存命中率保持在 90% 左右。

在数据库运行过程中，Buffer Pool的脏页会不断地写入磁盘，总的来说，Buffer Pool中存在三种不同类型的写操作。
（1）由于缓存大小是有限的，在某些情况下Buffer Pool Manager必须要将一部分脏页写回到磁盘；
（2）为了脏页写回操作不对前台用户线程造成影响，许多数据库都使用了预刷新策略；
（3）数据库系统通过Checkpoint机制清理WAL（Write Ahead Log）的空间，Checkpoint会造成大量的脏页面写入。

接下来，我们将介绍这三种不同类型的页面写入操作，下图1展示了三种类型的页面写入操作的过程。

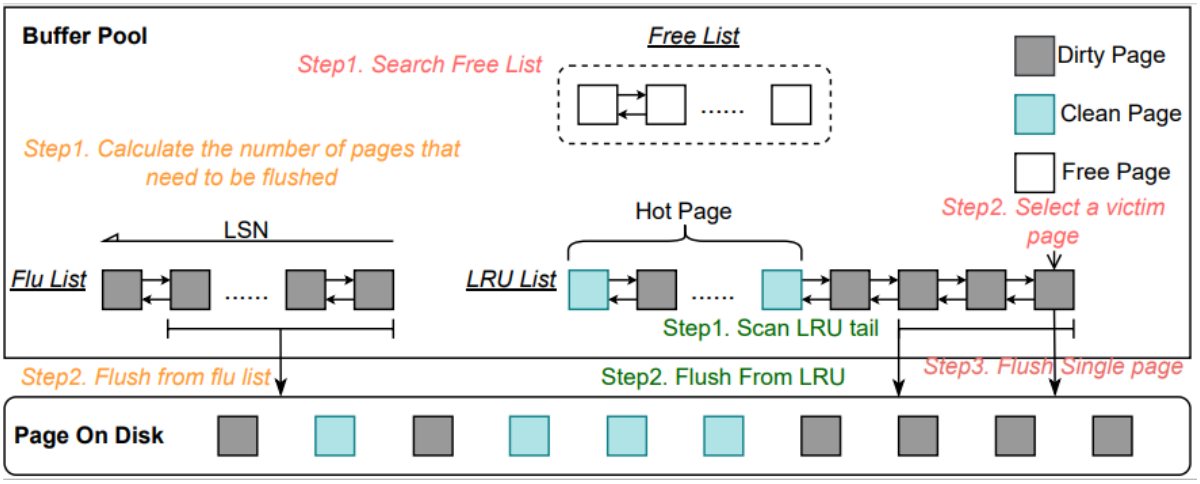

图1. Buffer Pool写操作

内存中的数据页被称为缓冲帧，缓冲帧中不仅维护磁盘页面的副本，还维护了磁盘页面在内存中需要的元数据信息，比如引用计数等。缓冲帧在内存中有三种状态：
（1）Free。该缓冲帧是空闲的，没有存储磁盘页。
（2）Clean。该缓冲帧存储了磁盘页面的副本，但是该页面未被任何事务所修改。
（3）Dirty。该缓冲帧存储了磁盘页面的副本，该页面被至少一个事务所修改过。

1.1.1 BUF_SINGLE_PAGE

Buffer Pool Manager将空闲的缓冲帧组织成链表（Free List），当前台事务请求的页面遇到Cache Miss时，将首先检查Free List中有无空闲缓冲帧（***Step1***）。否则当Buffer Pool已经没有空闲缓冲帧可使用时，需要根据缓存替换算法从后向前扫描LRU List寻找状态为Clean的缓冲帧（***Step2***）。一旦遇到一个状态为Clean的缓冲帧，就可以将磁盘页读取进该缓冲帧中。否则需要将LRU尾部的脏页同步刷新到磁盘

（***Step3***），以获得一个可供使用的空闲帧，然后将缺失的页面读取到该空闲帧中。

这种操作被称为BUF_SINGLE_PAGE，当Buffer Pool已满且没有状态为Clean的空闲帧时，它必须遵守Read-After-Write(RAW)协议，这是一个昂贵的操作，前台事务必须暂停，等待脏页面同步写入完成，才能发出读取请求。在Buffer Pool利用率很低，前台事务繁忙的情况下，每一次页面读取都会触发RAW操作，这将会造成严重的性能下降，对于数据库系统来说是不可接受的。

图1中红色文字展示了BUF_SINGLE_PAGE的过程。

1.1.2 BUF_LRU_PAGE

为了缓解RAW所造成的影响，通常现代数据库都采用了Preflush预刷新机制，数据库系统后台会有Dirty Page Writer线程，这种页面刷新方式称为BUF_LRU_PAGE。InnoDB使用Redo Log（WAL）保证已提交事务的持久性，使用Undo Log保证未提交事务的原子性，所以数据页面的刷新可以采取Steal、No-Force策略，在事务执行的任意时刻都可以将数据页面写入磁盘而不影响数据库的ACID特性。后台线程会定期LRU链表尾部（***Step1***），使用异步IO接口如Linux AIO将一定数量的脏页面写入磁盘（***Step2***），从而减少前台事务触发RAW操作的次数。

图1中绿色文字展示了BUF_LRU_PAGE的过程。

1.1.3 BUF_FLUSH_PAGE

数据库系统使用预写日志（WAL）存储事务对数据库所作的任何修改来保证事务的持久性。当数据库崩溃时，只需要扫描WAL，将其应用到数据库系统，即可恢复崩溃之前的状态。通常恢复时间和日志大小成正比，为了保持较短的恢复时间，需要定期执行Checkpoint强制刷新脏页来截断日志。这种页面刷新方式称为BUF_FLUSH_PAGE。

InnoDB将脏缓冲帧按照LSN从大到小组织成FLU List，当后台线程执行Checkpoint时，会根据当前日志产生速度和日志剩余容量计算出需要从FLU List刷新的脏缓冲帧数量N（***Step1***），总的来说，日志产生速度越快，N的值越大。然后将脏缓冲帧刷新到磁盘（***Step2***）。

图1中橙色文字展示了BUF_FLUSH_PAGE的过程。

1.2 Redo Log与崩溃恢复

由于Buffer Pool的存在，事务对磁盘数据的修改也许会落后于内存，如果数据库进程或机器崩溃，会导致内存数据丢失。现代的数据库大多使用ARIES算法或其变体来保证事务的持久性和原子性，数据库严格遵守Write Ahead Logging（WAL）规则，事务对数据页所做的所有更改在反映到磁盘之前需要先记录到预写日志中，并保证日志先于对应的Page落盘。当故障发生导致内存数据丢失后，数据库在重启时，通过重放预写日志，可以将Page恢复到崩溃前的状态。InnoDB中的预写日志称为Redo Log。

如下图2所示，InnoDB采取了Physiological Logging的方式来组织Redo Log，以Page为单位记录日志，日志内部记录对Page在逻辑上所做的修改。所有日志都用Log Sequence Number（LSN）来唯一的标识，此外数据库的Page的头信息会记录对该页面所作最后修改的日志的LSN。根据事务所做修改操作的不同，Log Record对应有不同的类型，例如当向Page的User Record中插入一条记录时会产生INSERT类型的日志，而更新Page中的记录时对应会从产生UPDATE类型的日志。日志的Log Body部分存储了日志重放时所需要的数据，根据日志类型的不同，该部分存储的内容也是不同的。为了达到可预期的崩溃恢复时间，数据库系统后台会不断地进行Checkpoint，1）从Flu List刷新一定数量的页面，我们假定这些页面中最大的LSN为Checkpoint LSN；2）Checkpoint LSN会被记录到Log文件中，这表明处于Checkpoint LSN之前的日志都可以被截断，其对应的修改已经被持久化到磁盘上。

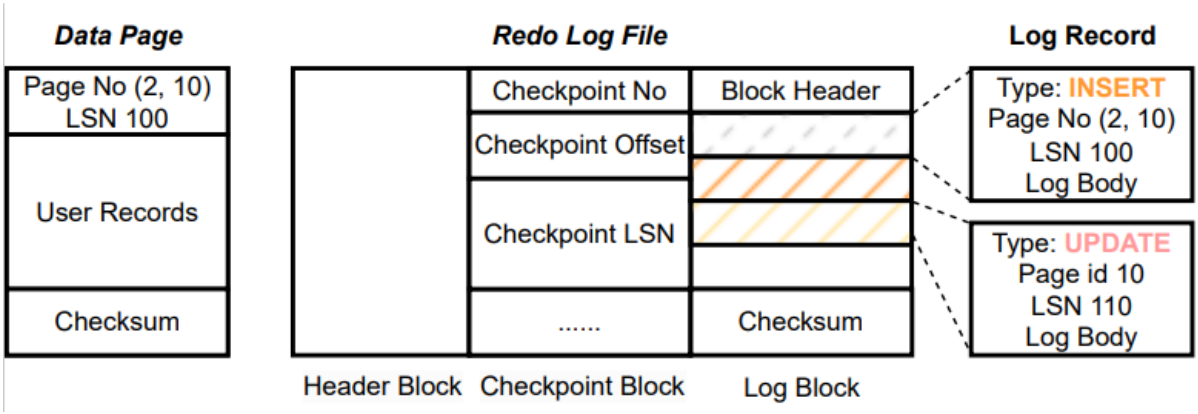

图2. Redo Log结构示意图

当数据库重启时，会扫描Redo Log进行崩溃恢复，这分为两个阶段。

1.2.1 Redo阶段

这个阶段负责将数据库恢复到崩溃之前的状态，该阶段不会区分事务的提交状态，已提交事务和未提交事务所做的修改都会被恢复。任何位于Checkpoint LSN之前的日志对应的修改都已经被持久化，因此，该阶段只会从Checkpoint LSN开始扫描Redo Log，比较Page LSN和当前Log Record的LSN，只有当Page LSN较小时才会将日志重新应用到该页面上。

1.2.2 Undo阶段

InnoDB会将未提交事务所做的修改存储到类型为Undo的Page中，Undo Page也是受到Redo Log的保护的，因此在Redo阶段会将Undo Page恢复到崩溃之前的状态。在Undo阶段会利用Undo Page来回滚崩溃时仍然处于活跃状态的事务。

2. 解决方案

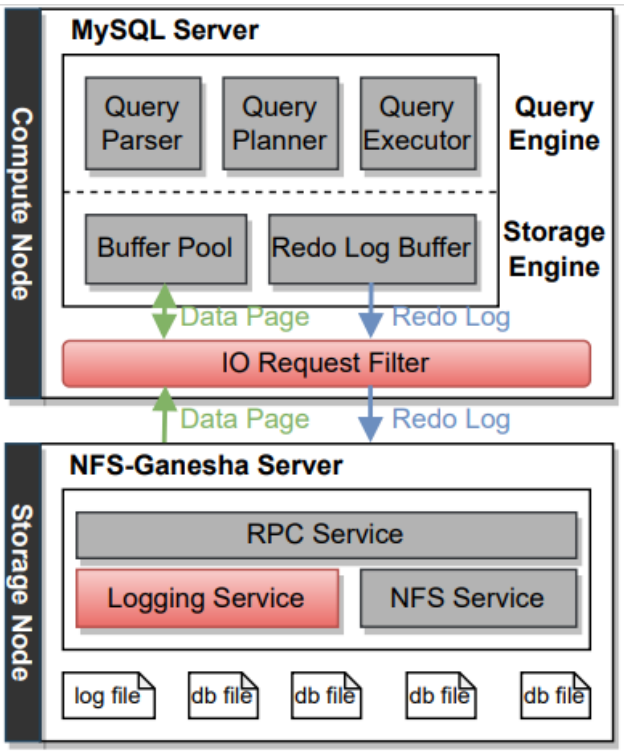

图3. logpushdown整体结构

MySQL作为一种常用的关系型数据库管理系统，在云数据库系统中扮演着重要的角色。然而，存算分离架构在提高可扩展性和可用性的同时，却引入了网络瓶颈的问题。此外，Buffer Pool和WAL机制导致了严重的写放大现象，对整个系统性能产生不可忽视的影响。为了应对这些挑战，我们提出了一种非侵入式的改进方案，旨在减少网络IO并降低写放大程度。

传统的MySQL日志系统要求将所有的数据修改操作持久化到Redo Log中，并将其下刷到磁盘。我们注意到，只需确保Redo Log的持久化，即可保证数据库系统的一致性。基于此观察，我们提出了一种非侵入式方案，通过绕过Buffer Pool中Data Page的下刷操作来减少网络IO，并利用存储节点的CPU资源实现日志重放，从而降低写放大的影响。

我们的非侵入式方案具有以下优势：

- 避免对MySQL源代码的修改：我们的方案无需修改MySQL的源代码，这意味着不会引入额外的风险和复杂性。它可以与现有的MySQL部署相兼容，并且易于实施和维护。
- 减少网络IO：通过仅下刷Redo Log并绕过Data Page的下刷操作，我们能够大大减少网络IO量。这在存算分离架构中尤为重要，可以显著降低网络传输带宽的消耗，提高整个系统的性能和响应速度。
- 充分利用存储节点的CPU资源：我们的方案依赖于存储节点的CPU资源进行日志重放，而不是依赖网络传输的速度。这允许日志重放工作快速完成，避免了读取等待的延迟，进一步提高系统的性能和可用性。

通过这个非侵入式的改进方案，我们能够在保持数据一致性的前提下，解决MySQL日志系统的网络瓶颈和写放大问题。我们的方法为云数据库系统带来了更好的性能和可扩展性，为用户提供更高效、稳定的数据管理服务。

我们基于MySQL 5.7.30和NFS-Ganesha实现了一个原型系统，称为LogPushDown，图3展示了LogPushDown的整体结构。

NFS-Ganesha是一个广泛使用的运行在用户态的NFS服务，它实现了完整的NFS协议，对外可以提供文件系统接口。相比于内核的NFS服务，它更加灵活，更利于原型系统的开发。

虽然我们实现的原型系统特定于MySQL数据库，但考虑到Redo Log是大多数面向磁盘的数据库系统都会采取的方案，因此我们相信该方法具有普适性，可以移植到其它数据库系统中。

IO Request Filter.为了绕过Buffer Pool中数据页面的下刷操作而不对MySQL应用造成任何影响，我们实现了一个可插拔的IO请求过滤器，MySQL对该组件是无感知的，该组件会将MySQL发出的Data Page写入请求过滤掉，只保留Redo Log的写入请求。

Logging Service.当Buffer Pool缓存未命中时，需要从存储节点读取相应的Data Page，Logging Service会对Redo Log进行管理，定期重放日志，保证Data Page Read请求读取到页面的最新版本。

2.1 数据页面读取路径

下图4展示了前台事务请求数据页面时的路径。可以将其分为Buffer Search，Waiting、Prepare Newest Page三个阶段。

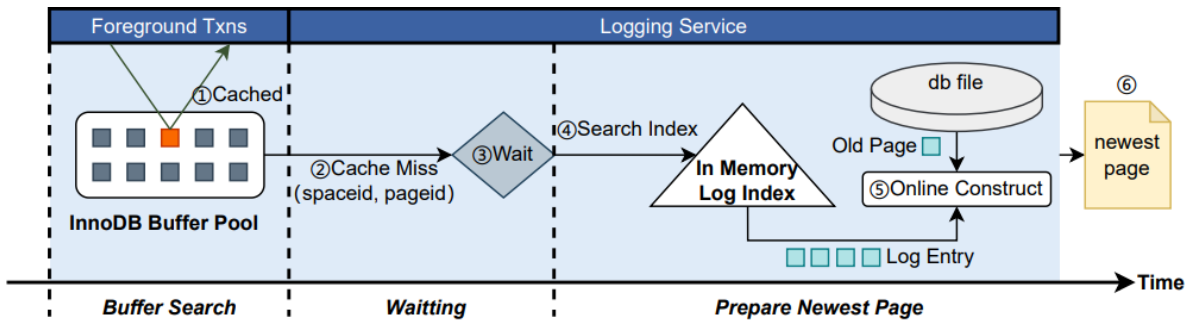

图4. 数据页面读取路径

Buffer Search.如果Buffer Pool中缓存有目标Page，会将其立即返回给前台事务（①），否则会产生Cache Miss，Buffer Pool会向存储服务器请求相应的页面（②）。

Waiting.Data Page的读取请求会被存储节点的Logging Service处理。Logging Servcice维护了一个内存日志索引，同时后台会有Log Parser线程不断地解析日志文件，更新索引。日志文件的解析可能会落后于日志文件生成的速度，当Data Page读取请求到来时，Log Parser可能还未来得及将最新产生的日志文件解析，更新到索引结构中，未解析的这部分日志可能会包含与该Page相关的日志。我们需要记录下请求到达这一时刻最新产生的日志的LSN，然后自旋等待log parser将最新产生的日志解析（③）。

Prepare Newest Page.Logging Service后台会有相应的Apply Worker线程定期进行日志重放操作，Apply Worker线程会从日志索引中选取部分日志，将其重放到Data Page中，被重放的日志会从日志索引中删除。当Data Page读取请求到来时，会从日志索引中检索与该Page相关的日志（④），如果检索不到，证明该Page已经被Apply Worker线程更新到最新的版本，此时无需进行其它任何操作，将最新的页面返回即可（⑥）。否则，我们需要进行在线的日志重放，以构建出最新版本的Page（⑤）。

3. 实现细节

3.1 非侵入式的IO过滤器

MySQL的所有IO请求均使用POSIX标准的文件系统调用接口，如open、pread、pwrite等，这些系统调用通常在libc中实现，LD_PRELOAD的机制允许指定的共享库优先于系统默认的库加载。因此，当一个程序中的函数被调用时，动态链接器首先会在预加载的共享库中寻找对应的函数实现，如果找到则会使用预加载的函数实现。因此，我们将IO过滤器实现为一个共享库libcatcher，libcatcher中实现了同名的pread、pwrite等系统调用，替换libc默认的系统调用行为。

具体来说，libcatcher拦截了下列系统调用：open、open64、pread、pread64、pwrite、pwrite64、close。

当然，在某些情况下我们仍然希望使用libc的系统调用，dlsym函数可以帮助完成这一任务，在libcatcher库被加载时使用dlsym在libc库中查找指定名称系统调用的地址，将其注册，后续即可使用。

MySQL中Redo Log文件有着特定的文件名，而数据库文件的格式为*.ibd，因此，在open系统调用中可以根据文件名区分不同的文件，如果是数据库文件（ibd），我们将其文件描述符加入过滤列表，这样，在pwrite系统调用中可以查找文件描述符过滤表，将数据库文件的写操作直接过滤。

这种方式无需对MySQL做出任何修改，只需要在MySQL启动时，通过LD_PRELOAD系统变量指定libcatcher所在的位置，如：

```
LD_PRELOAD=./libcatcher.so ./mysqld
```

即可无缝集成过滤器组件。

3.2 Logging Service

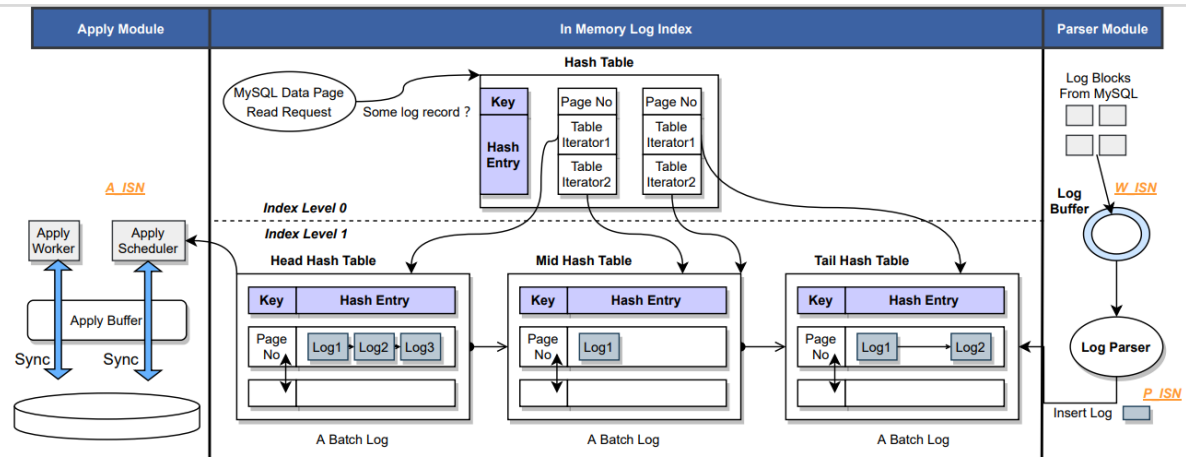

图5. logging service

3.2.1 Parser Module

当MySQL向存储节点写入日志时，不仅会将其写入到磁盘上的日志文件，同时会将日志在内存中的Log Buffer中缓存一份。Log Buffer是一块环形缓冲区，只会缓存最新的未被回放的日志，其大小与Batch Size有关。通过实验发现，在存算分离的场景下，由于受到网络带宽，锁争用等影响，MySQL事务系统处理事务生成Redo Log的速度始终要慢于日志回放的速度，在我们的设计中，由于后台线程会按批次（Batch Size）回放Redo Log，因此在最坏的情况下，会有一批未满Batch Size正在接收写请求的日志，和一批已满Batch Size正在进行回放的日志，因此，我们将Ring Buffer的大小设置为2倍的Batch Size是合理的。未来，如果计算节点使用了更强的CPU，同时使用了更高带宽的网络技术，如RDMA，有可能会出现Redo Log生成速度快于日志回放的速度，随着系统的运行，未被回放的Redo Log会逐渐堆积，一方面Log Buffer可能需要动态扩容，Redo Log会无限堆积，需要很大的内存空间。另一方面，存储节点中堆积的未被回放的Redo Log越多，MySQL读取数据页需要Online Construct的时间也会越久，因此，用户需要根据实际系统环境给Log Buffer设置一个合理的上限，当Redo Log的堆积达到阈值，必须暂停MySQL写入Redo Log。

需要注意的是，为了避免Redo Log的部分写（Partial Write）问题，MySQL以Block（512字节）为单位组织Redo Log，而Log Entry的大小是任意的，如图2中Redo Log文件结构所示，除了Log Record之外，Log Block还会在头部和尾部额外存储一些元数据，Ring Buffer中只会存储Log Record，并不会存储这些元数据。同时，如果Log Record不足以填充满整个Log Block，剩余部分会被0字节填充。MySQL的相邻两次Redo Log写入请求中，可能会存在两个Log Block被重复写入的情况，因此我们需要解决两个问题，1）对Log Block去除头尾部分的元数据，2）同时准确的提取出Log Record的增量部分。

我们使用三个指针来管理Log Buffer的空间：W_ISN、P_ISN、A_ISN，ISN（internal sequence number）是按照字节数全局单调递增的一个整数。

W_ISN。我们维护了一个W_ISN来指示下一次写入请求应该写入的位置。当有新的N个字节大小的Redo Log Record写入时，W_ISN会相应的增长N个字节。

P_ISN。从图5能够看到。Log Parser会不断地解析日志，每解析N个字节的日志，P_ISN便会相应增长N个字节。当P_ISN = W_ISN时Log Parser会暂停解析，等待Log Buffer中写入新的日志。

A_ISN。从图5能够看到。Apply Worker会不断地回放日志，一批日志全部回放完毕，A_ISN便会前进Batch Size个字节。为了防止未回放的日志被新写入的日志所覆盖只有当W_ISN - A_ISN < Log Buffer Size时才允许日志写入。

MySQL有两个Redo Log文件，它们组成一个日志文件组，循环使用，Log Buffer要比日志文件组小得多，因此还需要解决第三个问题，3）如何将Log Blocks中提取出的Log Record映射到范围更小的Log Buffer中。代码片段展示了将Redo Log blocks写入到Log Buffer的过程。

3.2.2 In Memory Log Index

如图5所示，我们基于哈希表为Redo Log维护了一个两层的内存索引，来加速Redo Log的查找。我们将第1层索引组织成一个哈希表队列。哈希表以Page No为键，值是一个Log Record的链表。

第一层索引中每个哈希表有三种状态：Open、Close、Applying。

队尾的哈希表处于Open状态，Log Parser单线程的顺序解析Log Buffer中的日志，根据Log Record中的Page No字段，将属于某个Page的日志添加到队尾的哈希表中。为什么将Log Parser设计成单线程而不是使用多线程进行加速，主要有两个原因：首先，MySQL的Redo Log中并没有字段表明该Log Record的长度，因此对于一条Log Record来说，必须顺序解析完所有的字段，才能知道下一条Log Record所属的位置；其次，为了保证日志回放的正确性，LSN小的日志需要先被回放，所以哈希表中的Log Record链表必须按照LSN的顺序排列，多个线程解析出的Log Record无法保证LSN单调递增的顺序。基于这两个原因，Log Parser必须被设计成单线程。Open状态的哈希表中存储的日志总字节数达到一定阈值（Batch Size）之后，会转换成Close状态，不再接收Log Parser的写入请求。Log Parser会重新初始化一个空的哈希表，将其加入到哈希队列的队尾，接收新的写入请求。队首的哈希表处于Applying状态，其中的日志会被Apply Module消费掉。

在3.2.1一节中我们提到，当日志产生速率大于日志回放速率时，日志会堆积，这就不可避免地造成第一层索引的哈希队列过长的问题，当MySQL下发一个页面读取请求，Logging Service需要进行Online Construct时，需要扫描哈希队列中所有的哈希表，最坏的情况是所有的哈希表中都没有相关的日志，那么这就是一次无效的扫描。为了加速这一过程，我们额外构建了一个哈希表将其作为第0层索引，该哈希表以Page No为键，值是一个哈希表指针数组，指向第1层索引中存储了该Page No相关的日志的哈希表。数据页面读取请求只需要查询该哈希表，找到相应日志所在第1层索引中的位置。大大提高了查询的效率。

3.2.3 Apply Module

Apply Module提供"尽力而为"的服务，它在后台不断的将索引中的日志重放，以期构建页面的最新版本。MySQL中有多达61种类型的日志，按照日志作用于目标页面的类型，可以将其分为6个种类，如下表1所示。其中，作用于R树页面和作用于压缩页面的日志是可选的，R树是MySQL中用来表示空间数据的数据类型，只有当启用了空间索引时才会生成该种类型的日志；同样的，只有当启用了页面压缩特性时，才会产生作用于压缩页面的日志。MySQL使用单独的Undo类型的页面来存储Undo日志，同样需要记录相应类型的日志来保证Undo页面的持久化。MySQL对应有Compact和Redundant两种行格式来格式化用户表中的行记录，不同的行格式在崩溃恢复进行回放时所需要记录的信息也是不同的，因此需要不同类型的日志。从MySQL 5.0开始，默认使用Compact行格式。当创建文件或修改文件名时，需要修改文件系统的元信息（inode），为了防止掉电丢失元数据，也必须记录相应的日志。

作用于文件信息	作用于 B+ 树页面		作用于 R 树页面	作用于 Undo 页面	作用于压缩页面	其它
FILE_NAME、	Compact 行格式	Redundant 行格式	PAGE_CREATE_RTREE、	UNDO_INSERT、	ZIP_PAGE_COMPRESS、	TRUNCATE、
FILE_RENAME、	COMP_REC_INSERT、	REC_INSERT、	PAGE_COMP_CREATE_RTREE	UNDO_HDR_REUSE、	ZIP_PAGE_WRITE_HDR、	INDEX_LOAD、
FILE_CREATE	COMP_PAGE_REORGANIZE	LIST_END_DELETE		UNDO_HDR_CREATE	ZIP_PAGE_REORGANIZE	WRITE_STRING

表1. Redo Log类型

每种类型的日志都有不同的回放逻辑，完全支持所有的日志类型的回放的工作量很大，通过分析TPC-C、Sysbench等工作负载产生的Redo Log，我们发现，只会产生作用于文件信息，作用于B+树页面（Compact行格式），作用于Undo页面和其它一些日志，在这些日志中，作用于B+树页面类型的日志对应的比例达到了95%，因此，我们只实现了作用于B+树页面类型的日志和其它一些必要的类型的日志，总共17种类型。对于Undo页面等不支持的日志类型，我们选择遵守MySQL原先的逻辑，在IO Filter组件中放行相应页面的下刷操作。

如下图6我们以向B+树插入一条记录对应的COMP_REC_INSERT类型的日志为例，讲解如何进行日志回放操作。

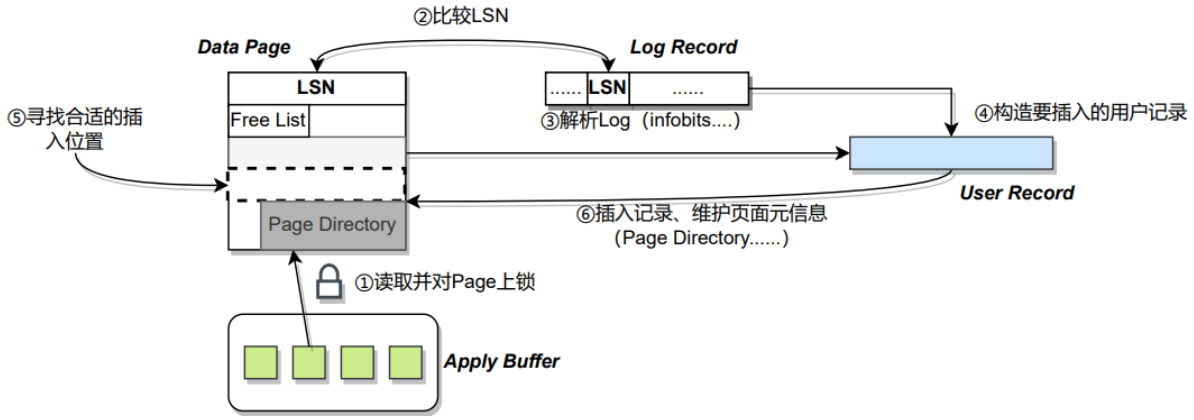

图6. COMP_REC_INSERT日志回放过程

从Apply Buffer中读取页面并上锁（①）。日志回放要求页面必须处于正确状态，因此一次日志回放操作必须被原子化，任何线程想要回放页面时必须先获取页面的锁，才能进行后续工作。比较Redo Log的LSN和Data Page的LSN（②），如果Redo Log LSN小于Data Page LSN，证明数据页面已经被其它线程回放到了一个较新的版本，此次日志回放可以直接跳过。

如下图7，想象这样一种场景，后台回放线程从Level 1 In Memory Log Index的Head Hash Table中提取出了LSN为100和110的两条日志，前台进程从整个In Memory Log Index中提取出了LSN为100、110、120、200的四条日志，他们都想要对Page 10进行日志回放，它们需要争抢页面的锁，假设后台线程先争抢到了锁，它回放日志100、110之后释放锁，此后前台线程才能获得页面的锁，通过比较LSN发现，日志100、110已经被回放，它会跳过这两条日志，而只回放日志120和200。

为了尽量减小Log Record的大小，COMP_REC_INSERT类型的日志只会记录和前一条用户记录中不一致的部分，需要解析Log Record（③），获取前一条用户记录在Data Page中的位置，合并Log和前一条用户记录的数据，构造出完整用户记录（④），然后需要从Page中申请一块空间（⑤）（可能会重用Free List空间），将新构造的用户记录拷贝到指定的空间中（⑥），同时，维护页面中的元信息，其中最重要的就是要更新页面的LSN。

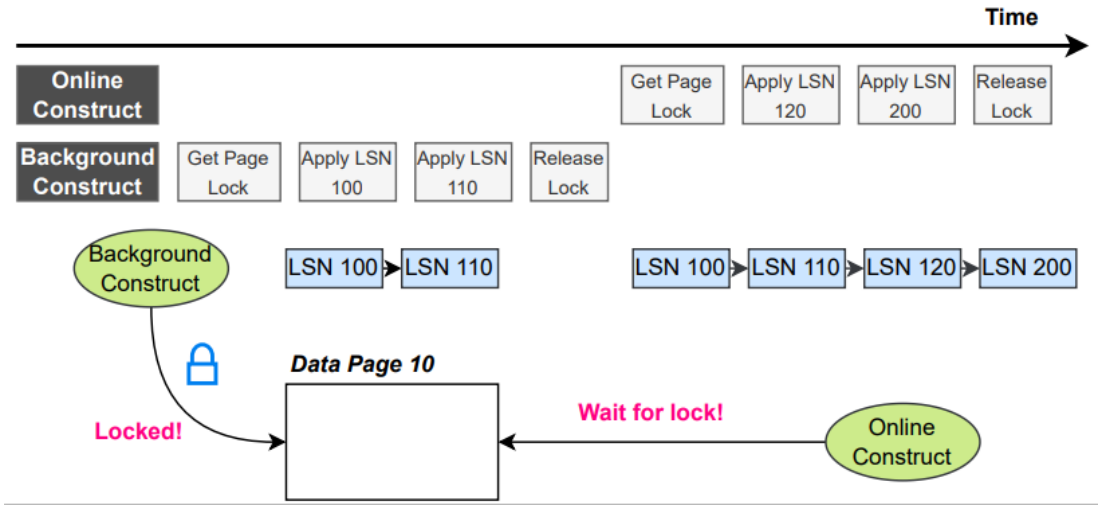

图7. Online Construct和Background Construct的冲突

只有加速后台日志回放的速率，在MySQL的Buffer Pool产生Cache Miss读取数据页面之前，尽可能早的将数据页面相关的日志重放，减少Online Construct过程中需要重放的日志条数，才能避免页面读取请求停顿过长的时间，从而影响事务的吞吐量和延迟。因此，Apply Module被设计为一个Scheduler和多个Worker的多线程的模型。当Head Hash Table转变成Close状态时，会立即启动一轮后台日志重放工作，Scheduler以桶为单位将哈希表中的日志平均分配给若干个Worker，同一个哈希桶内的日志只会被分配给一个Worker，以减少线程间的争用。Worker完成一个页面的日志回放之后都需要将页面立即写回磁盘。Scheduler在分配完任务之后，也会变成Worker进行日志回放工作，等待所有的worker完成工作变成空闲状态后，Scheduler会前进A_ISN。

4. 环境搭建

4.1 MySQL端

本文基于MySQL5.7.30进行构建，因此首先需要搭建MySQL的环境。

1. 首先安装依赖：

```
sudo apt install pkg-config libssl-dev bison -y
```

2. 配置生成Makefile：

```
# 进入项目根目录
mkdir cmake-build-release
cmake -S . \
-B cmake-build-release \
-G "Unix Makefiles" \
-DCMAKE_BUILD_TYPE=Release \
-DCMAKE_INSTALL_PREFIX=path/to/mysql \
-DMYSQL_DATADIR=path/to/mysql/data \
-DMYSQL_UNIX_ADDR=path/to/mysql/data/mysql.sock \
-DSYSCONFDIR=path/to/mysql/data \
-DWITH_DEBUG=ON \
-DWITH_BOOST=path/to/mysql-code/boost \
-DMYSQL_MAINTAINER_MODE=OFF
```

3. 编译生成可执行文件：

```
# -- -j 16 指定编译项目时使用的线程数量。
cmake --build cmake-build-release --target all -- -j 16
```

4. 在启动 MySQL 之前需要初始化数据目录，设置 root 账户的权限和密码：

```
# 初始化数据目录前创建 MySQL 用户组和 MySQL 用户
sudo groupadd mysql
sudo useradd -r -g mysql -s /bin/false mysql
```

5. 初始化mysqld的数据目录，进入项目根目录下，执行下列操作：

```
cd ./cmake-build-release/sql
 .mysqld --basedir=path/to/mysql --datadir=path/to/mysql/data --
lower_case_table_names=0 --user=mysql --innodb-flush-method=O_DIRECT --
innodb_flush_log_at_trx_commit=1 --innodb_log_file_size=2G --
innodb_change_buffering=none --default-storage-engine=InnoDB --default-tmp-
storage-engine=InnoDB --disabled_storage_engines=MyISAM --innodb-checksum-
algorithm=none --innodb_log_checksums=OFF --innodb_doublewrite=0 --initialize-
insecure
```

6. 解决启动 `Error` ：

`MySQL` 打印错误日志时依赖 `errmsg.sys` 文件，在编译 `MySQL` 后，会在 `cmake-build-debug/sql/share` 文件夹下生成各种语言版本的 `errmsg.sys` 文件，将该文件拷贝至 `/home/lemon/mysql/share` 文件夹即可。

```
mkdir path/to/mysql/share/
cp /path/to/mysql-code/cmake-build-release/sql/share/english/errmsg.sys path/to/
mysql/share/
```

7. 给数据目录添加读写执行权限：

```
chmod -R 777 path/to/mysql/data
```

8. 启动Server端mysqld：

```
 .mysqld --basedir=path/to/mysql --datadir=path/to/mysql/data --
socket=path/to/mysql/data/mysql.sock --lower_case_table_names=0 --user=mysql --
innodb-flush-method=O_DIRECT --innodb_flush_log_at_trx_commit=1 --
innodb_log_file_size=1G --innodb_change_buffering=none --default-storage-
engine=InnoDB --default-tmp-storage-engine=InnoDB --
disabled_storage_engines=MyISAM --innodb-checksum-algorithm=none --
innodb_log_checksums=OFF --innodb_doublewrite=0
```

9. 启动Client端mysql修改root用户密码，进入项目根部录下，执行下列操作：

```
/path/to/mysql-code/cmake-build-release/client/mysql -uroot -h127.0.0.1 -P3306 -p


```

10. 不用输入密码，直接回车，启动 MySQL 客户端登录 MySQL 之后，在客户端修改 `root` 用户密码，然后关闭Server端和Client端：

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY "root";
FLUSH privileges;
shutdown;
exit;
```

4.2 准备测试数据

以sysbench基准测试工具为例介绍如何生成测试数据。

重新启动MySQL，使用客户端工具进行连接，创建测试使用的数据库：

```
CREATE DATABASE sbtest;
```

然后使用sysbench初始化测试数据：

```
sysbench --threads=20 \
         --mysql-host=127.0.0.1 \
         --mysql-port=3306 \
         --mysql-user=root \
         --mysql-password=root \
         /usr/share/sysbench/oltp_common.lua \
         --tables=40 \
         --table_size=200000 \
         prepare
```

4.3 NFS-Server端

1. 首先安装依赖

```
sudo apt install g++ libboost-dev cmake make git doxygen
sudo apt install build-essential libglu1-mesa-dev libc6-dev
sudo apt install libkrb5-dev libsasl2-modules-gssapi-mit libkrb5-dev
sudo apt install liburcu-dev
sudo apt install libcap-dev libtirpc-dev rpcbind
sudo apt-get install uuid-dev libacl1-dev liblzo2-dev
```

注意：启动 `nfs-ganesha` 之前要先安装 `nfs-kernel-server`，不然当你启动 `nfs-ganesha` 时会得到下列报错，猜测可能是少了什么依赖？查找到应该是少了 `rpcbind`。

```
Cannot register NFS V3 on TCP
```

2. 安装 `nfs-kernel-server`

```
sudo apt install nfs-kernel-server
sudo /etc/init.d/nfs-kernel-server stop # 关闭nfs-kernel-server，因为我们不需要使用它
```

3. 修改NFS-Server配置文件：

找到nfs-ganesha-sql/src/include/applier/config.h文件，修改 LOG_PATH_PREFIX、DATA_FILE_PREFIX、DATA_FILES、PER_FILE_LOG_FILE_SIZE四个变量，其中DATA_FILES变量是你所生成的sysbench的所有表数据文件，LOG_PATH_PREFIX是Redo Log所在的文件路径，DATA_FILE_PREFIX是MySQL数据库表数据文件的主目录，PER_FILE_LOG_FILE_SIZE是每个Redo Log日志文件的大小。

4. 编译源码：

```
# 进入项目根目录
cd nfs-ganesha-sql
mkdir src/build
cmake -S src -B src/build -DCMAKE_BUILD_TYPE=Release -DUSE_FSAL_VFS=ON -
DUSE_9P=OFF -DUSE_FSAL_LUSTRE=OFF -DUSE_FSAL_GPFS=OFF -DUSE_NLM=OFF -
DUSE_FSAL_CEPH:STRING=OFF -DUSE_FSAL_GLUSTER:STRING=OFF -
DUSE_FSAL_KVSFS:STRING=OFF -DUSE_FSAL_LIZARDFS:STRING=OFF -
DUSE_FSAL_NULL:STRING=OFF -DUSE_FSAL_MEM:STRING=OFF -
DUSE_FSAL_PROXY_V3:STRING=OFF -DUSE_FSAL_PROXY_V4:STRING=OFF -
DUSE_FSAL_RGW:STRING=OFF -DUSE_FSAL_XFS:STRING=OFF -DUSE_GSS=OFF
cmake --build src/build --parallel 20
```

5. 启动nfs-server：

先创建配置文件

```
vim ~/ganesha.conf
```

配置文件内容如下，你需要修改Plugins_Dir参数，使其指向nfs-server编译生成的库目录，Path参数代表你需要暴露给客户端使用的文件目录。

```
NFS_CORE_PARAM {
   Plugins_Dir = path/to/nfs-ganesha-sql/src/build/lib;
}
EXPORT
{
    Export_Id = 77;
    Path = path/to/mysql-Dir;
   Pseudo = /;

    FSAL {
        Name = VFS;
    }
    Access_Type = RW;
    Disable_ACL = true;
    Squash = No_Root_Squash;
    Protocols = 3,4;
}
EXPORT_DEFAULTS{
    Transports = UDP, TCP;
    SecType = sys;
}
```

no_root_squash： 表示当客户机以root身份访问时赋予本地root权限（默认是root_squash）。

root_squash：表示客户机用root用户访问该共享目录时，将root用户映射成匿名用户。

建立一个目录存放 `ganesha` 的 pid 信息：

```
sudo mkdir /var/run/ganesha
```

最后提权启动nfs服务器：

```
sudo path/to/nfs-ganesha-sql/build/bin/ganesha.nfsd -F -L /dev/stdout -f path/to/
ganesha.conf -N NIV_EVENT
```

-F(foreground)，将ganesha.nfsd服务运行在前台，-L 指定将日志输出的位置，-f 指定配置文件所在的路径，-N 指定输出日志的级别。

4.4 启动并测试

1. 客户端MySQL一侧需要挂载NFS服务，将服务器的mysql目录暴露给mysql一侧：

```
sudo mount -t nfs4 11.11.11.12:/ path/to/mysql-Dir(本地挂载点)
```

2. 修改io_filter组件，并编译，打开libcacher项目，再 catcher_filter.cpp 文件中修改 data_file_set 变量，指向所有的sysbench生成的数据库表文件，然后将其编译，编译出来的.so动态库叫 libcatcher_filter.so。
3. 启动MySQL，其中LD_PRELOAD指定libcatcher_filter.so动态库的位置，--basedir和datadir应该是你挂载的远程目录。

```
LD_PRELOAD=path/to/libcatcher/libcatcher_filter.so ./mysqld --basedir=path/to/
mysql --datadir=path/to/mysql/data --
socket=/tmp/mysql.sock --lower_case_table_names=0 --user=mysql --innodb-flush-
method=O_DIRECT --innodb_flush_log_at_trx_commit=1 --innodb_change_buffering=none
--default-storage-engine=InnoDB --default-tmp-storage-engine=InnoDB --
disabled_storage_engines=MyISAM --innodb-checksum-algorithm=none --
innodb_log_file_size=2G --innodb_buffer_pool_size=128M --innodb_doublewrite=0 --
innodb_log_checksums=OFF
```

4. 启动sysbench进行测试：

```
sysbench --threads=8 \
         --time=180 \
         --report-interval=10 \
         --mysql-host=127.0.0.1 \
         --mysql-port=3306 \
         --mysql-user=root \
         --mysql-password=root \
         /usr/share/sysbench/oltp_read_write.lua \
         --tables=40 \
         --table_size=2000000 \
         --mysql-ignore-errors=2013 \
         --mysql-socket=/tmp/mysql.sock \
         run
```

5. 实验结果展示

我们搭建了一个存算分离系统进行实验，检测LogPushDown的性能。存储节点的配置和计算节点相同，额外挂载了一块PCIe3.0的三星980固态硬盘，用来存储数据库文件。节点之间使用RDMA网卡（仍然使用普通的TCP/IP协议）进行网络通信，计算节点运行MySQL-Server和TPC-C工作负载，存储节点运行NFS-Ganesha。

5.1 吞吐量测试

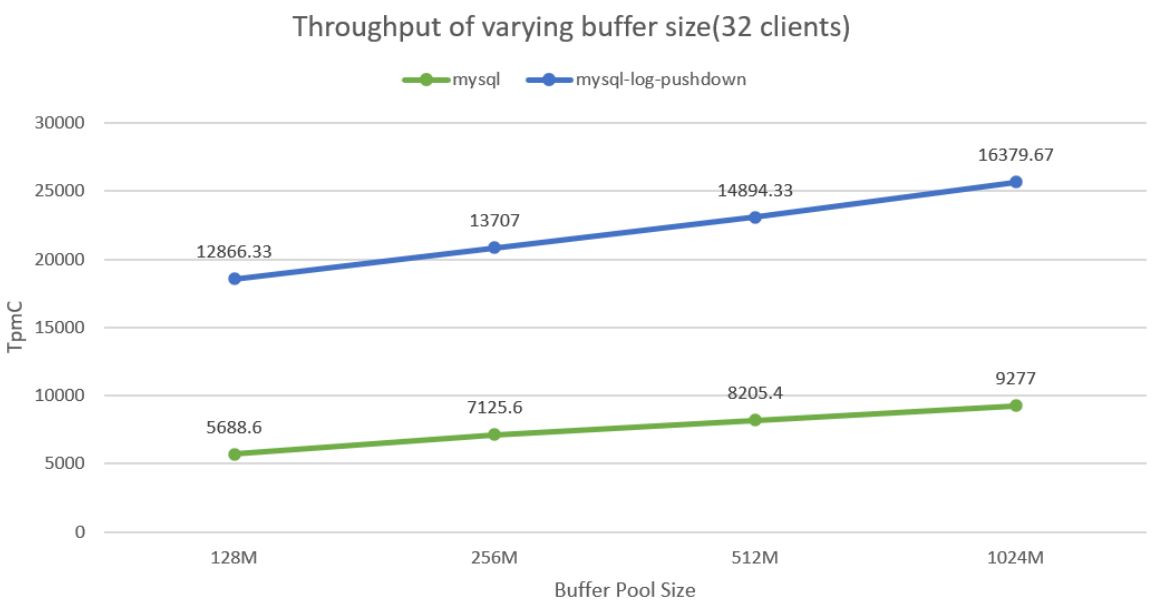

图8. TPC-c吞吐量测试

如上图8所示，在固定MySQL连接数量为32个线程的情况下，我们将MySQL的Buffer Pool从128M逐渐增大到1024M，上图展示了TPC-C的吞吐量对比，绿色线条mysql代表原始场景下的吞吐量，蓝色线条mysql-log-pushdown代表本文所提出方案的吞吐量。可以看到相较于原始场景，吞吐量最高提升了接近两倍。

5.2 网络IO测试

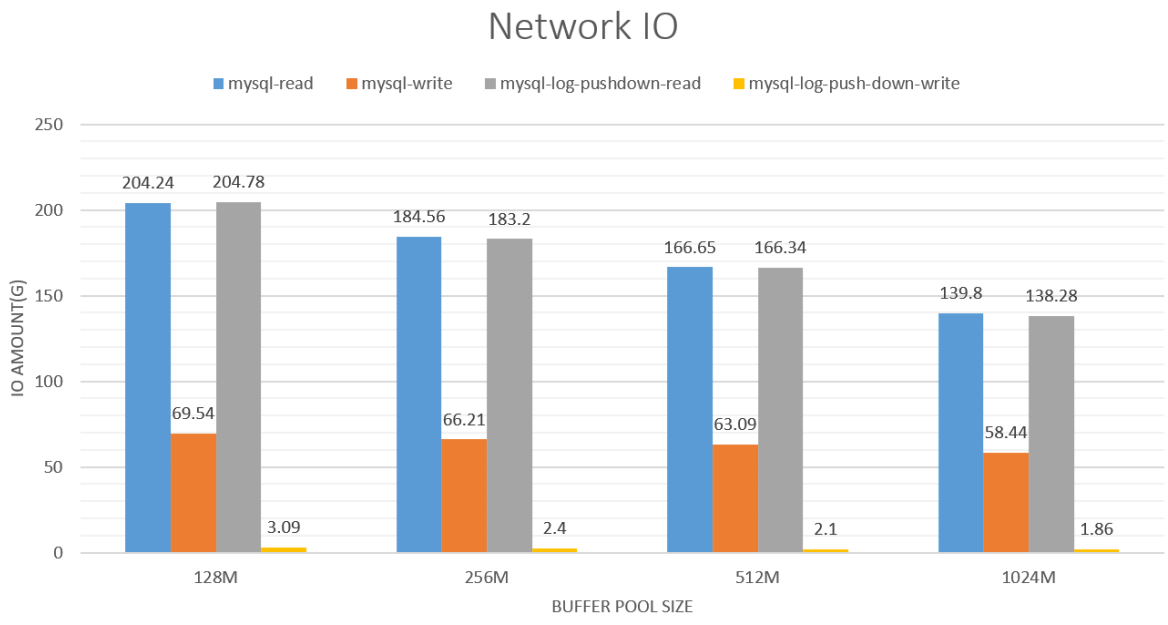

图9. 网络IO测试

如上图9所示，我们修改了TPC-C的源码，在MySQL连接数量为32个线程的情况下，使每个线程运行3000个事务，测试结束后我们测量了计算节点和存储节点之间的网络IO情况。如上图所示，与原始场景对比，读网络IO基本保持不变，但是由于我们的方案基本过滤掉了所有的数据页面写入操作，在运行过程中只有Redo Log、Undo Log和其他一些页面的写入，因此写IO非常小，相较于原始场景，在Buffer Pool Size为512M的情况下，写入IO最多下降了79.2倍。在运行同样数量的事务的情况下，越大的Buffer Pool能够缓存更多的页面，触发的IO次数也会越少，因此随着Buffer Pool大小的增加，相应的读写IO总量呈现下降的趋势。