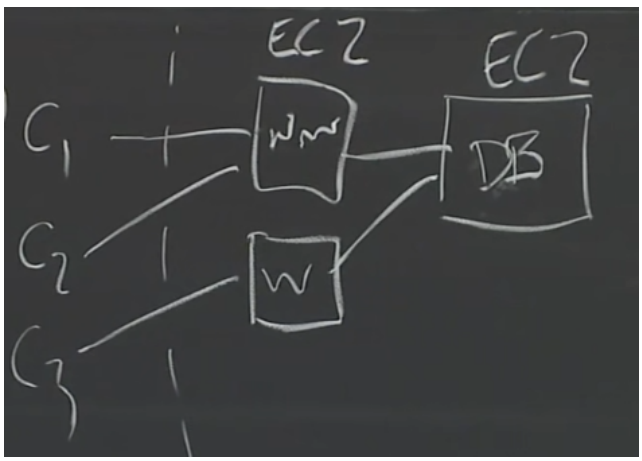
LECTURE 9 AURORA

1. 背景知识

- 亚马逊信息基础设施
  - EC2（Elastic Compute Cloud）弹性计算云：在一台物理机器上运行多个虚拟机。这些虚拟机出租给用户。这台物理机器使用的存储为连接该物理机器的磁盘。每个用户被划分了一定大小的磁盘空间。
    - EC2用来构成数据库会存在问题，考虑如下场景：若干台web服务器使用了EC2实例，用于应答用户的请求，而这些web服务器本身并不存储状态和数据，额外存在一个EC2实例用于运行数据库。但当运行数据库的EC2实例崩溃时，EC2对应的物理资源也不再有权限访问。而亚马逊的S3服务允许拍摄快照，可以一定程度上解决上面的问题，但是两次备份之间的数据会丢失。



  - EBS（Elastic Block Store）弹性块存储：EBS服务由两台台EBS服务器及该服务器对应的物理磁盘组成。两台EBS服务器的硬盘使用链式复制的方式保持一致。EBS服务对于EC2实例而言，就像EC2本地的磁盘一样。EBS只能为一台机器上的一个EC2实例使用而不能共享。为了降低CR（Chain Replication）的成本，因此同一EBS服务下的两台EBS服务器必须在一个可用区（AZ）内
    - 在EC2构建服务器的情况中，当EC2实例崩溃时，可以重开一个EC2实例，将EBS挂载到新的EC2实例，重新配置数据库，就能得到数据
- 背景噪声：在分布式系统中，持续出现的各种硬件故障和软件故障，它们可能会干扰系统的正常运行
- 数据库实例：一个数据库实例（Database Instance）指的是一个正在运行的数据库系统的单个副本。每个数据库实例都是一个独立的、在内存中加载的数据库运行环境，它可以处理数据库中的请求、执行查询、管理事务等。一个数据库系统可以同时运行多个数据库实例，每

个实例可以访问不同的数据库。每个数据库实例都有自己的内存、进程和资源管理，使得它们可以独立地运行和处理用户请求。数据库实例的数量和配置取决于系统的需求和硬件资源。数据库实例通常包括以下组件和功能：内存结构：数据库实例会分配一部分内存用于缓存数据、索引和查询计划，以加快数据的访问速度。

  - 进程管理：数据库实例会管理多个进程，包括查询处理器、事务管理、锁管理、缓冲区管理等。
  - 存储管理：数据库实例负责将数据从磁盘加载到内存中的缓冲区，并将修改后的数据写回磁盘。
  - 事务管理：数据库实例确保事务的ACID特性，以保证数据的完整性和可靠性。
  - 查询处理：数据库实例会处理用户提交的查询，执行查询优化和执行计划，返回结果给用户。
  - 锁管理：数据库实例负责管理并发访问，以防止多个事务之间的数据冲突。

  注意：数据库是所有元组的集合！而数据库实例则可以认为是一个软件。MySQL中一个数据库实例可以操纵多个数据库，Oracle中一个数据库对应一个数据库实例。

- Redo 日志：数据库管理系统中的一种重要机制，用于记录数据库中发生的所有修改操作，以确保数据库的持久性和恢复能力。它是数据库事务处理和故障恢复的关键组成部分（可以说是事务的持久性的保证！）。当数据库执行写操作（如插入、更新、删除）时，会将数据的变化记录到Redo日志中。Redo日志具体会记录：
  - **物理页**发生变化的具体内容（修改前后的值）
  - 操作的时间戳
  - 相关的事务标识

    在数据库中，当一笔数据修改操作发生时，首先会将这个操作记录到Redo日志中，然后再将实际的数据修改应用到内存中的数据页中，最后再将这个操作标记为已完成。这样，即使在数据被写入磁盘之前，如果数据库发生崩溃，通过Redo日志中的信息，系统可以重新执行日志中的操作，以确保数据的一致性。在故障恢复中，数据库系统会利用Redo日志来重新执行数据库中未完成的操作，从而将数据恢复到最近的一致状态。对于事务，从事务开始就逐渐写入Redo Log执行，而非写完Redo Log再进行执行。当内存中的物理脏页被写回到存储器中时，Redo Log相关Entry就可以被覆盖

- 二进制日志：与Redo日志不同，二进制日志记录的是逻辑操作的变化，可以认为存储的就是sql语句。但是二进制日志是为了恢复数据库的，而Redo日志是为了保证事务的持久性的。同时，二进制日志仅在**事务提交之后**一次性被写入。

- 松耦合：软件工程中的一个概念，用于描述系统中各个组件之间的关系程度。它指的是组件之间的依赖关系相对较弱，即一个组件的修改不会显著地影响其他组件。这有助于实现系统的模块化、可维护性和可扩展性。
- 可用区（Availability Zone）：一个云服务提供商（如Amazon Web Services）的数据中心通常会划分为多个可用区。每个可用区都包含了计算资源、存储设备、网络设备等，这些资源相对独立地运行和管理。这种架构可以实现高可用性和容错性，确保系统在面临故障或其他问题时能够继续提供服务。
- Double Write：传送门

2. 概述

- Amazon Aurora是一个用于联机事务处理（OLTP）的关系型数据库（rational database）
- Aurora的总体架构为单点写，多点读（此处结点指数据库引擎）。主备数目（数据库引擎）可以达到15个。存储层结点的replica数目为6个。只有一个primary可以写，其他的只能读。存储层结点均为一个一个为Aurora特制的服务器，而不是通用的EBS（EBS只能处理和块相关的操作），也许存储节点内部可能用到了EBS进行构造
- 作者认为对于数据吞吐量的限制已经由存储和计算转变为了网路
- 数据库，通过使用多租集群（multi-tenant fleet）可以将IO负载均摊到集群上的各个结点，因此存储系统中的热点现象不再成为数据库性能的主要障碍。数据库的瓶颈变成了数据库层发出IO和存储系统层处理IO的网络，具体包括：
    - PPS （packets per second）
    - 带宽
    - 大流量 ， which 因为数据库系统大量并行发送写请求到存储集群而导致的
- 数据库大多数操作可以并发进行，但是有些情况下仍然需要进行同步：
    - cache缺失。（此处的cache似乎是数据库在内存中维护的一个buffer，而非硬件cache）当数据在数据库的cache中缺失时，需要写回脏数据，然后从磁盘中读取数据。这种情况下，读取线程需要等待读取数据完成，期间一直阻塞。虽然数据库可以通过检查点和脏页写回来减少上述情况的发生，但是这些操作本身也需要阻塞，并且需要上下文切换。
    - 事务的提交。事务的提交需要阻塞其他的操作。这一过程需要用到多阶段提交协议（例如2PC），分布式数据库系统会持续性地产生背景噪声，而这些协议在容错方面表现却并不好。同时分布式数据库的各个结点可能在地理上相隔非常远，因此多段提交协议的延迟比较大

- 本论文充分利用了Redo Log，使用了一套新颖的面向服务架构。这套架构包含一个**多租横向存储服务（multi-tenant scale-out storage sevice）**。此存储服务抽象了一个虚拟化的分段redo log，并且和分布式数据库集群的**实例（instance）** 是松耦合的。每个数据库实例都保留了查询处理器（见前文数据库实例的介绍）、事务管理、缓冲区管理和锁管理，但是把redo日志记录、持久化存储、崩溃恢复、备份等操作移植到了上述多租横向存储服务中。
- Aurora的优势
    - 通过构建一个具有独立容错和恢复的存储服务，能够保护数据库免遭性能波动以及短期或长期的故障
    - 通过将Redo 日志仅写入存储，将**IOPS（Input/Output Operations per second）** 降低一个数量级。从而解决网络瓶颈，提高数据库吞吐量。
    - 将备份和redo recovery从 一次性的、昂贵的操作之中转移到 连续的、异步的 操作中。这些操作可以在大型分布式集群中进行分摊（amortize）。从而：
        - 提供了不需要检查点的即时恢复
        - 不影响前台（foreground）进程的备份操作

3. Durability at Scale （规模化的持久性）

3.1. 复制和相关的故障

- 数据库实例的生命周期与存储系统的生命周期并不一致（例如，数据库实例故障，但是存储系统中存储的数据往往不会被清空或重写）。因此可以做到将计算层和存储层进行解耦。
- 多数投票：用于给存储系统容错。如果一个数据项有V份replica，则读取操作和写操作分别需要获得V_r和V_w的票数。并且需要保证：
    - 对于读操作：$V_r + V_w > V$

      这一条是为了保证当前的读集和上一次写集存在交集，因此读集之中必定至少有一个元素有最新写入的数据。但注意未必会读到最新写入的数据（也就是可能会读到stale数据），所以在每一个**存储结点**中会为数据维护版本号！
    - 对于写操作：$V_w > V/2$

      这一条是为了写操作能够发现在自己之前最新写入的数据，避免数据冲突
- 亚马逊将一个数据项的多份replica分布在不同的AZ上（Availability Zone），用于加强容错。但亚马逊考虑了如下的复杂情景：假设有replica A，B，C分别分布在AZ A，AZ B和AZ C上，

此时AZ B中有一个replica存在错误，正在恢复，而此时整个AZ C由于不可抗力等原因全部故障，则剩下一个AZ A无法获得多数投票。

- 亚马逊考虑整个AZ故障之后提出的投票模型：
  - 一共6份replica，存在三个AZ上，每个AZ两份replica
  - 读操作需要获得3个投票
  - 写操作需要获得4个投票
- 根据上述的投票模型，系统的容灾容错能力提升为：
  - 允许整个AZ故障，以及该AZ之外的另一个replica故障，仍然保证可读。（此时仍然可以得到3个投票，但不可写）
  - 允许整个AZ故障，仍然保证可写。（此时可获得4个投票）

3.2. 分段存储

- 原因：我们不希望在系统发生故障时，第二次发生故障。因此需要升高MTTF（Mean Time To Failure）或者降低MTTR（Mean Time To Repair）。但是MTTF通常与硬件的设计等因素相关，所以很难提高。Aurora转向降低MTTR。
- 具体实现：恢复时，需要恢复的数据越小，MTTR就会越小。因此Aurora将DB Volume（暂不明具体含义，姑且理解为数据库存放的数据）拆分成为若干个10GB大小的段。每个段有6个replica（算上自己），此6个replica成为一个PG（Protection Group）。6个replica的分布方式如前文所述分布在3个AZ的6个结点上。DB Volume是由若干个PG串联组成的。组成DB Volume的PG数目会随着DB Volume的增大而增多，最大支持64TB DB、 Volume。
- 一个存储结点可能会有若干个属于不同PG的段。某一个**存储结点**（注意是存储层结点而不是主备数据库）发生故障时，所有属于不同PG的段都会进行恢复。此时只需要从每个PG中选出一台，将段信息传送到新的机器上即可，如果每个PG选出的结点能尽量不相同，则这些结点可以并行地读取，提高吞吐率和恢复速度。

3.3. 因为系统弹性而带来的其他优势

首先，Aurora在出错恢复方面的能力很强。如果一个系统能够处理非常麻烦的故障情况，自然也能处理非常简单迅速的故障情况。因此这一特性可以被充分利用：

- 热点问题：当某些存储节点过热时，可以将在该存储节点上的段标记为故障，此时并不会影响多数投票。而后将一些段移植到别的存储节点上之后再回复即可
- 系统安全补丁、软件升级：一次处理一个AZ，不允许机器自动升级和打补丁。

4. Redo Log

4.1. 分段方式对传统数据库的负担

- MySQL的写操作需要很多不同的IO操作。而段的复制（图中序号为3）又会进一步导致IO操作增加，从而导致PPS（packet per second）负担加重。同时，一些IO操作需要进行同步，因此会阻塞流水线，扩大延迟。（CR/CRAQ似乎可以解决网络IO频繁的问题，但是仍然不能解决阻塞流水线的问题）
- MySQL写入流程：将数据写入堆文件或B+树中，同时将数据修改写入Redo Log之中（实际情况更为复杂）。可以将修改操作应用于更新前的数据来得到最新的数据。
- 下图展示了完整的数据流：

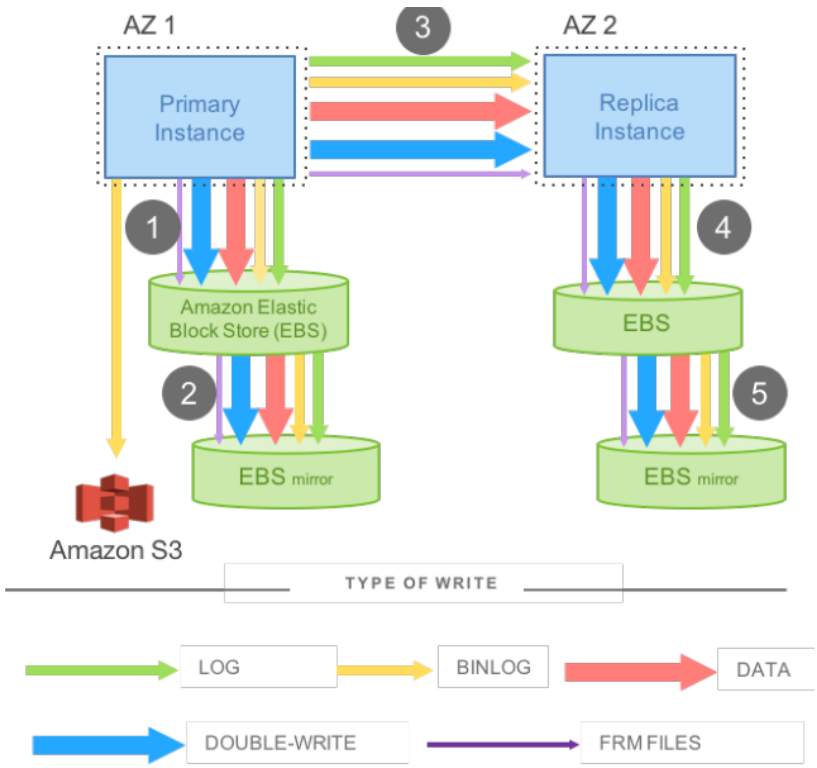


Figure 2: Network IO in mirrored MySQL

其中frm files为表的元数据。DB 引擎额外发送了一份binlog给Amazon S3，用于提供按时间点恢复的功能。（Amazon Simple Storage Service是亚马逊提供的一种云存储服务，是一种高度可扩展的对象存储服务，不仅可以存储文本和二进制数据，还可以存储图像、音频、视频、备份文件、日志数据等各种类型的数据。）

数字指明了数据流传递顺序。上图的缺陷如下：

- 其中1、3、5是顺序（sequential）且同步的（synchronous），因此1、3、5的延迟会叠加。
- 由于需要等待最慢的操作完成，抖动也会异常情况被放大。

- 从分布式系统的角度来看，是一种4台机器，写入需要4票的多数投票方法，显然不能够应付某结点故障的情况
- 很多写入操作都写的是一样的数据，尽管方式可能不同。

4.2. 存储层分离

通常情况下，当一个事务开始时，Redo Log就开始工作，当事务中的某一个操作被Redo Log彻底持久化之后，事务才被应用到MySQL缓冲池中，此时即视为该操作完成。在事务提交时，必须保证Redo Log已经被完整地持久化了。

Aurora将计算层和存储层分离，日志应用程序（Log Applicator）也被完全放到了存储层中。网络传输的内容只包含Redo log的内容。从引擎的角度来看，Redo log就相当于数据库（the log is the database）。在存储层，日志应用程序会根据日志的内容来生成数据库的页，这一过程根据需求产生或是后台自动进行。当日志内容较长时，日志应用程序可能会缓存（materialize）生成的页（这一步不是必须的）。相比于检查点机制定期地记录检查点而言，这些页不是定期缓存的，只在日志较长时需要缓存。

具体例子如下：

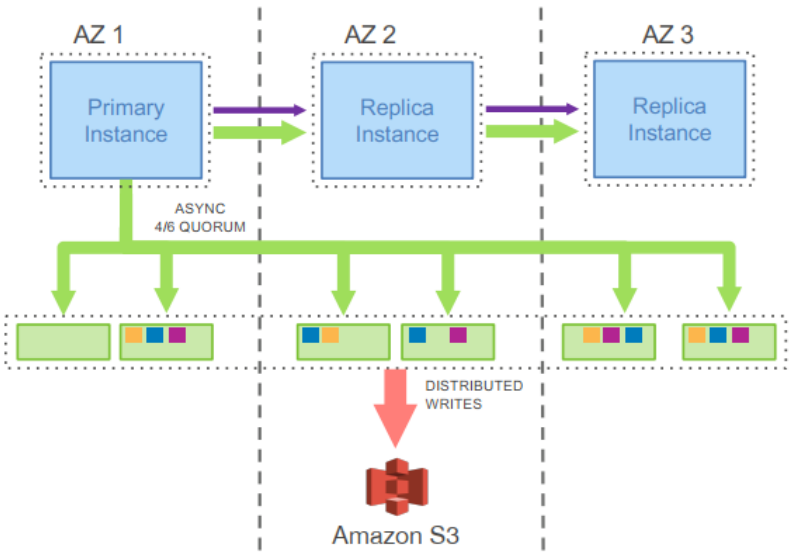


Figure 3: Network IO in Amazon Aurora

其中AZ 1 首先把Redo log 记录写给存储服务；并且IO流会批处理log记录，把这些log记录和元数据更新一起发送给replica。**此后，每个存储结点如何操作详细请看关于存储结点的部分！**
primary会等待replica的回复，用来进行多数投票表明写入成功。每个数据库引擎内部都会有page buffer，包括replica内部。因此 是为了保证replica中page buffer的一致性，才需要primary向replica传递log record 和元数据。事实上replica面对读取请求时，是直接从自己的buffer cache（注意在数据库引擎层提到的buffer = cache = page buffer = buffer cache，都是一样的概念）和存储结点中读取信息的（发生页缺失时）

Aurora大大提升了网络IO性能！与镜像MySQL相比，Aurora提高了单位时间处理的事务数目（即吞吐量增加），降低了每个节点上处理事务的IO次数。

将存储层分离，还进一步带来了如下优势：

- 最小化故障恢复时间
- 消除系统抖动（由检查点、后台页面回写、备份等后台进程导致）

对于容错恢复而言，Aurora与传统的数据库有非常大的不同。传统的数据库需要从最近的检查点开始，重做 redo log（已持久化的redo log，redo log本身也有buffer）中的内容来保证事务一致性。但在Aurora中，redo log的重做发生在存储系统层。并且重做是持续性、异步地发生的。任何读取操作到来时，如果当前页尚且不存在，则需要应用redo log。因此，对于Aurora来说，将容错恢复分布在如上所述的常规前台操作之中。

4.3. 存储服务的设计

- 根本宗旨：降低写操作在前台的延迟。因此Aurora把写操作大部分的处理挪到了后台
- 由于前台的请求是波动的，因此Aurora有足够的时间在后台处理相关的写操作
- 当磁盘没有快要满时，不使用垃圾回收机制（回收旧的页），来用CPU时间换磁盘负载
- 在Aurora中，后台操作和前台操作是呈现负相关的。（为什么？）而传统数据库后台操作和前台操作是呈现正相关的。当后台积压了太多未完成的处理时，Aurora就会限制前台写请求的数目。由于Aurora对于写操作有6台replica拿到4票即能通过的操作，因此当某个结点工作量过大时，可以将其视为慢结点，而不影响整个多数投票算法。
- 存储结点上的对log record的处理：

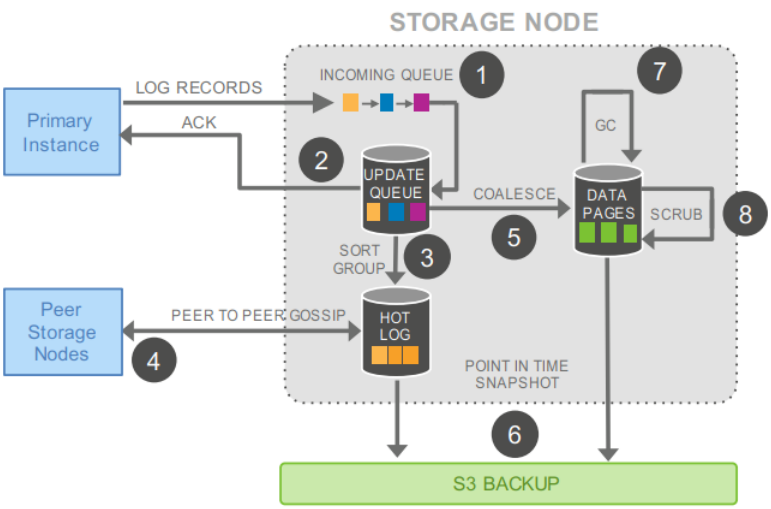


Figure 4: IO Traffic in Aurora Storage Nodes

1. 接受来自上层发送的日志，将其插入到自己的**更新队列**
2. 持久化日志条目，并且向上层发送ACK
3. 组织这些log entry，检查中间是否有空缺。（因为网络IO会以批处理的方式发送数据，有一些batch可能会丢失）
4. 若log entry存在丢失，向别的结点索要对应log entry
5. 将多条log entry合并为新数据页
6. 定期将数据页写入Amazon S3
7. 定期进行垃圾回收，回收条件：某个页的$LSN < VDL$
8. 定期对页上的**CRC(循环冗余校验码)** 进行校验

上述所有操作都是异步的，且只有1和2是在前台进行的。由1和4操作可以得出VCL，也就是连续的、一致的最大的LSN

5. Log如何产生?

5.1. 异步Log处理

- 每一条Log Entry都含有一个LSN（Log Sequence Number），LSN是单调递增的，由DB引擎分配
- 关于一致性和持久性，Aurora维护了一个节点，当收到来自其他replica关于未完成请求的ACK时，就会向前推进这个节点。
- 这些运行时状态使得在进行读取操作时不需要再进行多数投票；但当在故障恢复时，这些状态会丢失，因此仍然需要多数投票
- 存储系统的恢复与事务恢复的不同：
  - 对于事务而言，若数据库崩溃，则需要DB引擎中的跟踪机制来分别决定每一个事务执行是否需要回滚。
  - 对于存储系统，存储系统会在整个数据库系统启动时进行自检和恢复操作。这一过程不是从事务的角度来完成的，而是保证分布式数据库各个结点能看到同样的存储内容和结构。
- 存储系统的几个概念：
  - LSN：Log Sequence Number，见前文

- VCL：Volume Complete LSN，在VCL之前的所有LSN对应的log entry都必须是可用的。而VCL必须是满足上述条件的最大的LSN。一般来说，在故障恢复时VCL之后的所有log entry都必须被截断（truncate），并恢复。VCL即各个存储结点中连续一致的最大LSN
- CPL：Consistency Point LSN，它们是每个min-transaction的最后一条log record
- VDL：Volume Durable LSN，小于等于VCL的最大CPL。若存在VDL，则从VDL之后截断。VDL是由db引擎发送给存储层的。VDL之前的日志都可以被垃圾回收。

5.2. 数据库的常规操作

5.2.1. 写

- 写限制：数据库中可能会有多个并发的事务在执行。这些事务都需要生成各自的日志记录。数据库会给每个事务的日志记录分配一个LSN。这个LSN满足：$LSN \leq VDL + LAL(LSN\ Allocation\ Limit)$。LAL是对当前执行事务的写操作限制，避免前台接受了过多的写操作请求，而存储层来不及处理。
- 每个PG中包含6条一样的段，这个段只能看到一部分的log record。这些log record必须是与该段相关的。
- backlink 反向链接：每个log record都有一个指针指向同一个段中的前一条log record。由此可以构建出整个段上的log record
- SCL：Segment Completeness LSN。比SCL小的log record （当前段的log record）都已经收到了，并且SCL是满足该条件的最大的LSN。是根据backlink 而得到的。当有些结点丢掉了批处理的网络包之后，可以通过SCL向PG内的其他结点索要数据（即Gossip协议！）

5.2.2. 提交

- Aurora中事务提交是异步完成的。
- 数据库中有个专门处理事务提交的线程。当事务到达时，它会将事务的**commit LSN**记录在一个列表中。这个列表保存了所有待提交的事务。
- 当且仅当 最新的 VDL 大于等于事务的commit LSN时，事务被提交。
- 当有满足上述条件的事务时，会有专门的线程给等待的client发送提交确认消息

5.2.3. 读取

- Aurora也有缓冲区（buffer cache），读写操作先针对缓冲区进行。仅当缓冲区发生了页缺失，才会发起磁盘IO的请求

- 传统数据库在进行页面替换时，会写回脏页。而Aurora不写回。Aurora通过始终保证buffer cache中的数据是最新的来实现同样的效果：当**page LSN（也就是修改了这个页的，最新的log record的LSN）** 大于等于VDL时写回（有争议！！！！！！！）。这一替代的实现保证了：
  - 页中的所有修改都在record log之中
  - 缺页时，只需要请求到当前VDL版本的页即可获得最新的持久版本
- Aurora在集群工作正常的情况下，读取操作不需要多数投票。当需要读某个页时，由于该页存在Segment Completeness LSN，因此只需要寻找那些满足$SCL \geq ReadPoint$的存储节点读取数据即可，其中ReadPoint是**发出**读取请求时的VDL。
