云原生数据库架构的演变

Log is Database

目录

CONTENTS

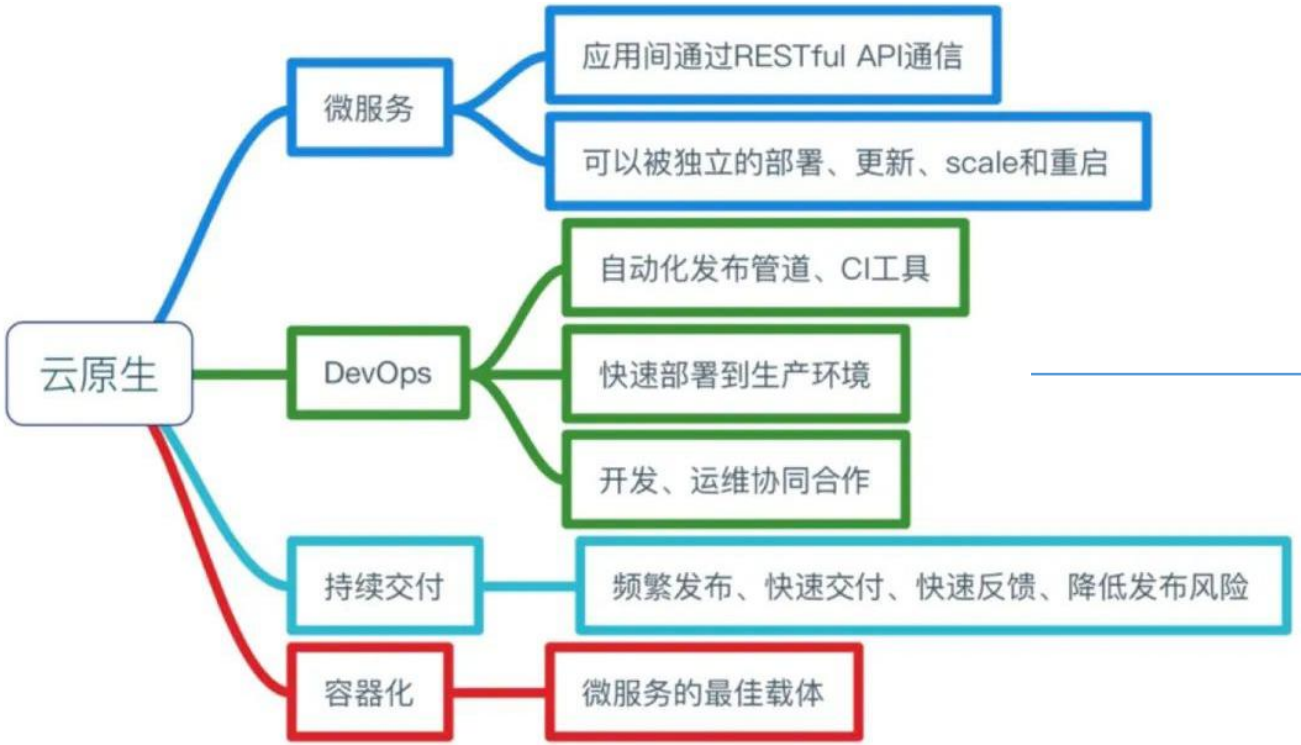
目录

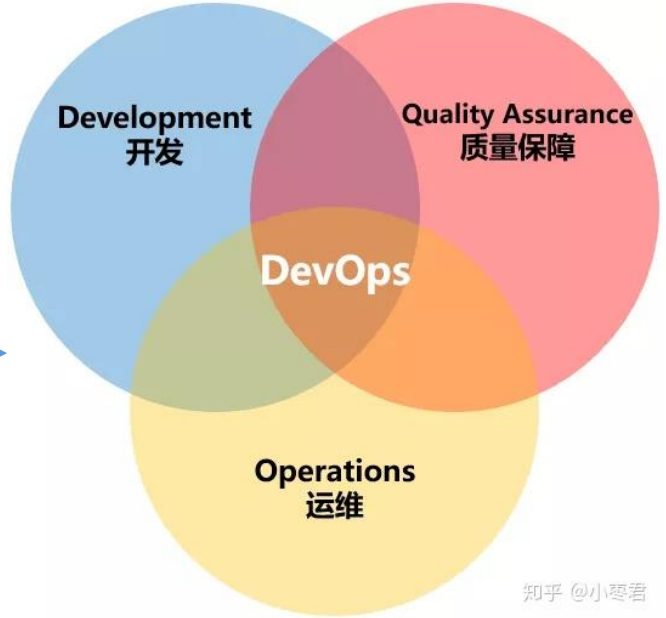
CONTENTS

01 云原生

云原生（Cloud+Native）

- Cloud 是适应范围为云平台，
- Native 表示应用程序从设计之初即考虑到云的环境，原生为云而设计，在云上以最佳姿势运行，充分利用和发挥云平台的弹性+分布式优势。





目录

CONTENTS

02 在云上使用传统MySQL遇到的问题

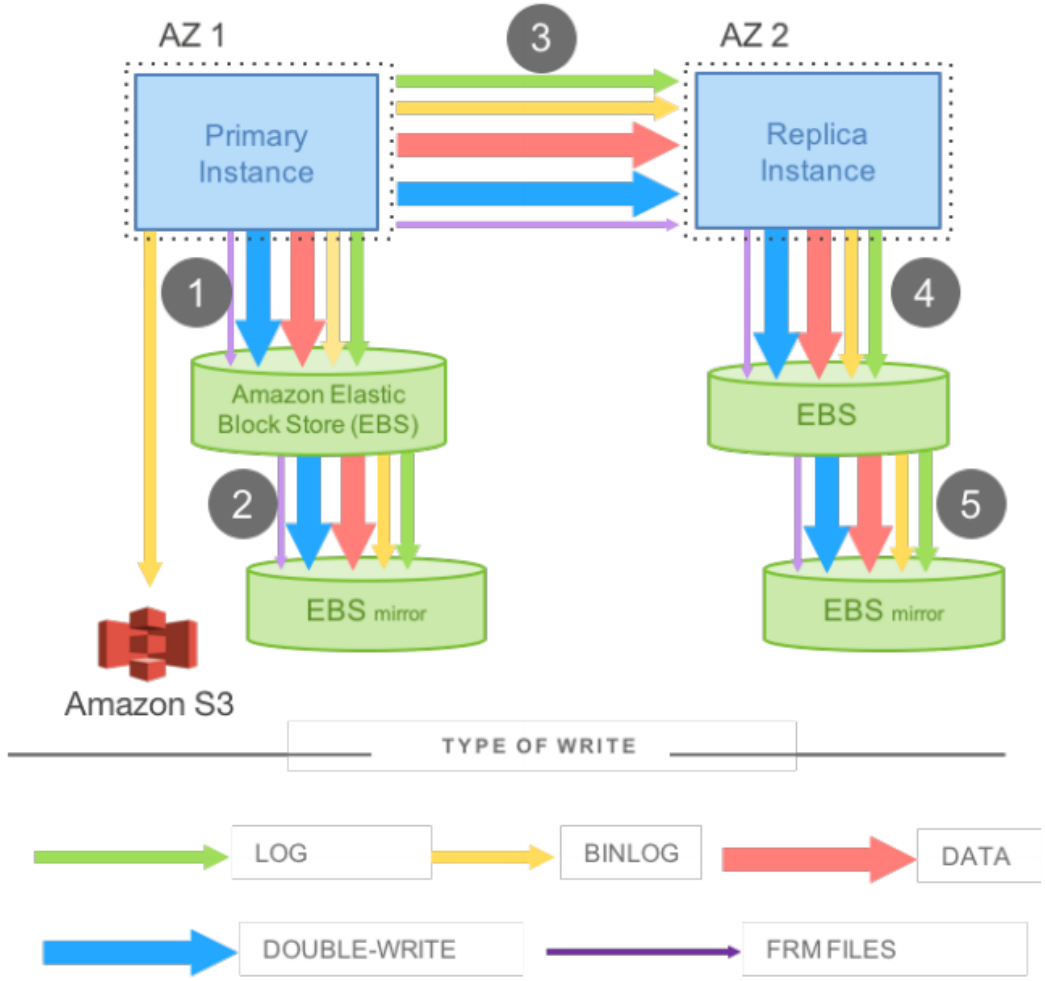


Figure 2: Network IO in mirrored MySQL

Amazon在日常开发和维护中发现，计算能力和存储性能已经不再是其工作的瓶颈了，取而代之的是网络的流量。其实对于Amazon来说，只要有钱，CPU能用最好的就能解决计算能力的问题，机械硬盘不够用固态硬盘，固态硬盘不够就上内存，存储性能也解决了，但是网络的延迟靠大带宽是很难解决的，必须要从业务逻辑和服务组件上找问题。所以他们发现了MySQL在分布式系统中消耗了大量的流量，增加了网络延迟

02 在云上使用传统MySQL遇到的问题

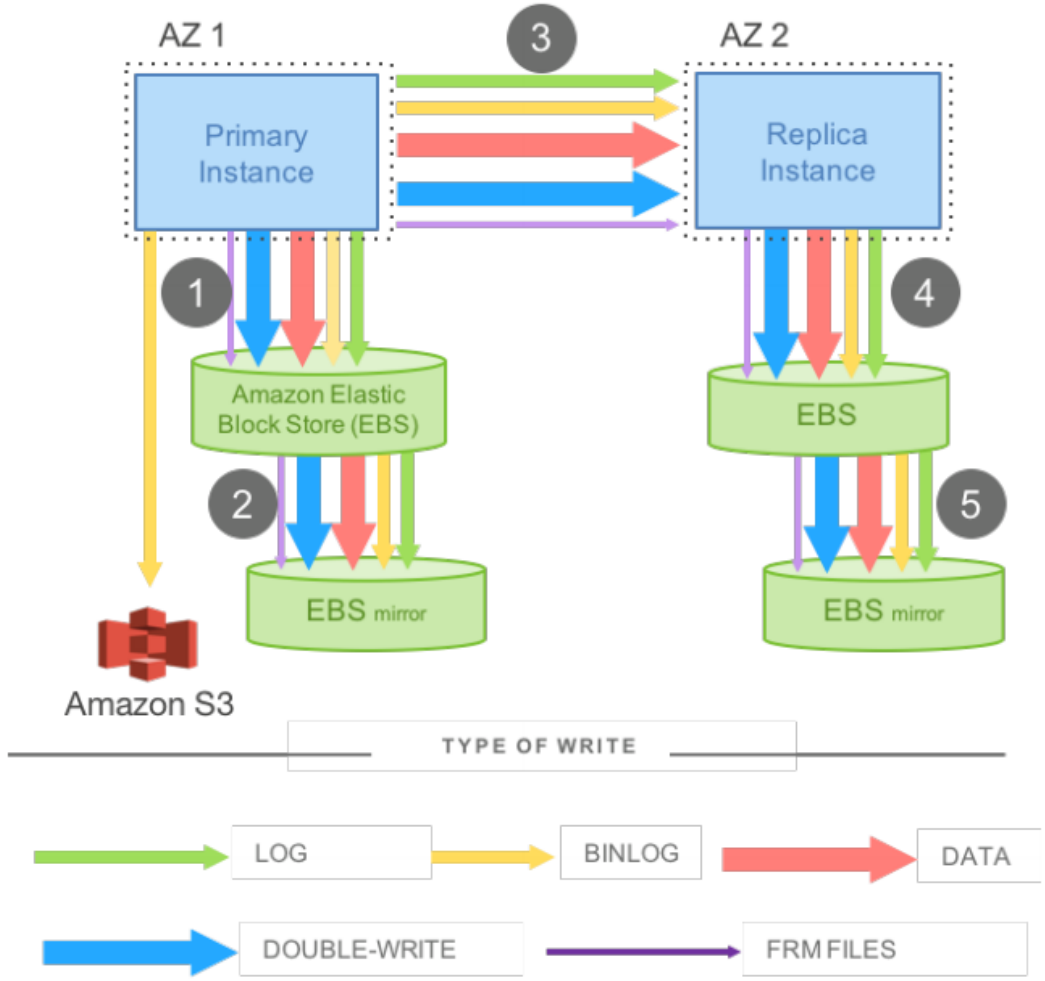


Figure 2: Network IO in mirrored MySQL

在一个主从MySQL架构下，要执行一次写操作必须经历以下路径：

① 主节点AZ1将数据写入EBS1

② EBS1将数据写入备份镜像EBS2

③ 主节点将数据发送到从节点AZ2

④ 从节点AZ2将数据写入EBS3

⑤ EBS3将数据写入备份镜像EBS4

每次数据写入都包含5个部分，消耗了太多的网络带宽，加上1、3、5步是顺序同步的，导致响应时延过高。

The binary (statement) log that is archived to Amazon Simple Storage Service (S3) in order to support point-in-time restores

02 THE LOG IS DATABASE

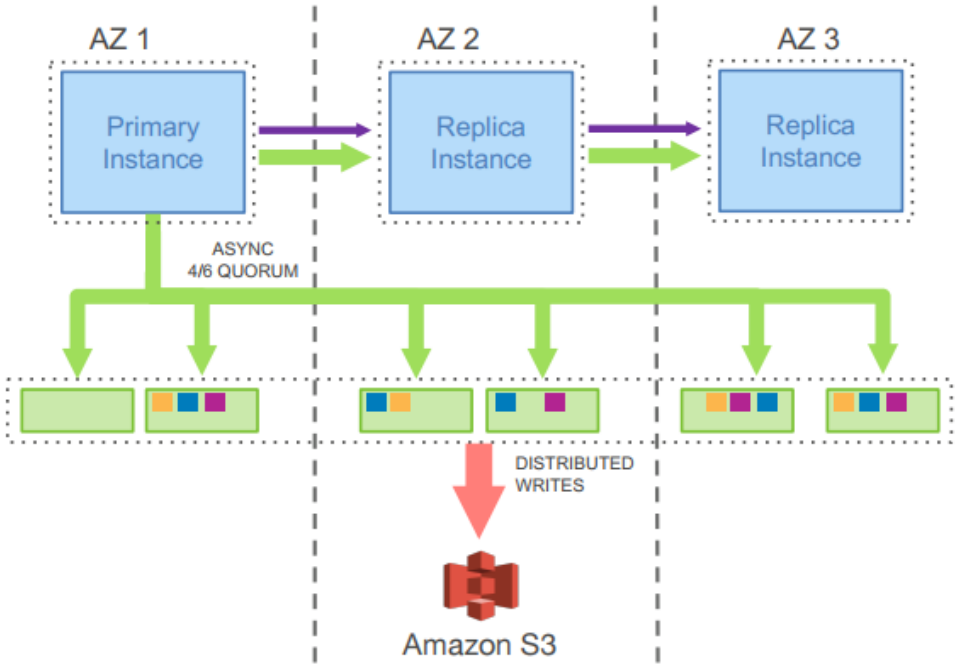


Figure 3: Network IO in Amazon Aurora

由于传统MYSQL在同步数据的过程中发送的信息过多，Amazon提出**LOG IS DATABASE**的思想，用**REDO LOG**来整合所有有用的信息并略去无用信息，减小了网络IO。另外，Amazon使用**链式复制结构**取代主从结构，简化了保证数据一致性的复杂度。

以三个副本为例，当AZ1主节点收到写请求后，它将LOG异步写入六个存储节点中，然后将 LOG 和 METADATA 通过链式复制结构传递给AZ2和AZ3。

另外，只有主节点负责将LOG写入存储节点，从节点只负责接收LOG并在本地重放，不需要负责写入存储节点。这样就减少了传统MYSQL架构中的第4、5步开销。

传统MYSQL中的两级存储节点也由一级Quorum代替。两级存储的响应时延是两次操作之和，而一级Quorum操作时延取决于Quorum中的最长时延。这样，Aurora也优化了整个系统的响应时延。

02 Storage Node

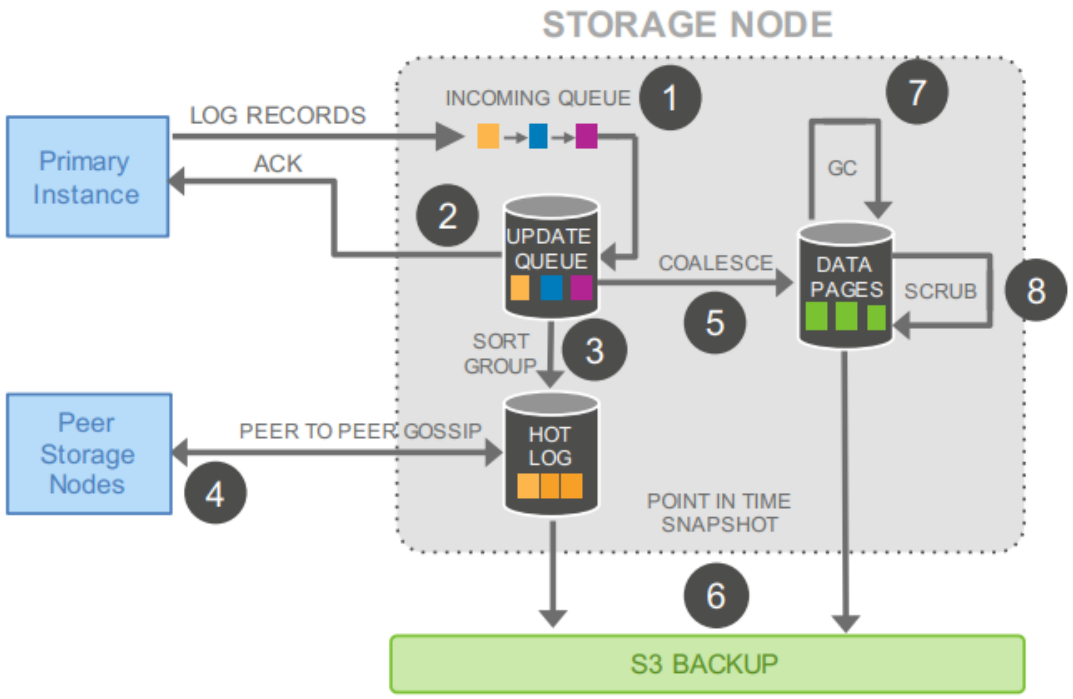


Figure 4: IO Traffic in Aurora Storage Nodes

① 接收到LOG RECORDS后将其写入一个内存队列；

② 将LOG RECORDS持久化后向主节点返回ACK；

③ 由于网络不可靠和Quorum机制，当前存储节点可能缺失了部分LOG。所以需要根据LSN对LOG进行排序，找出缺失的LOG；

④ 通过Gossip协议与其他存储节点交换信息，将缺失的LOG复制到本地；

⑤ 将LOG应用到Data Pages；

⑥ 周期性地将LOG和Data Pages备份到远端S3；

⑦ 周期性GC，收集旧版本Data Pages；

⑧ 周期性地对Data Pages进行CRC校验。

主节点的响应时延取决于（1）（2）

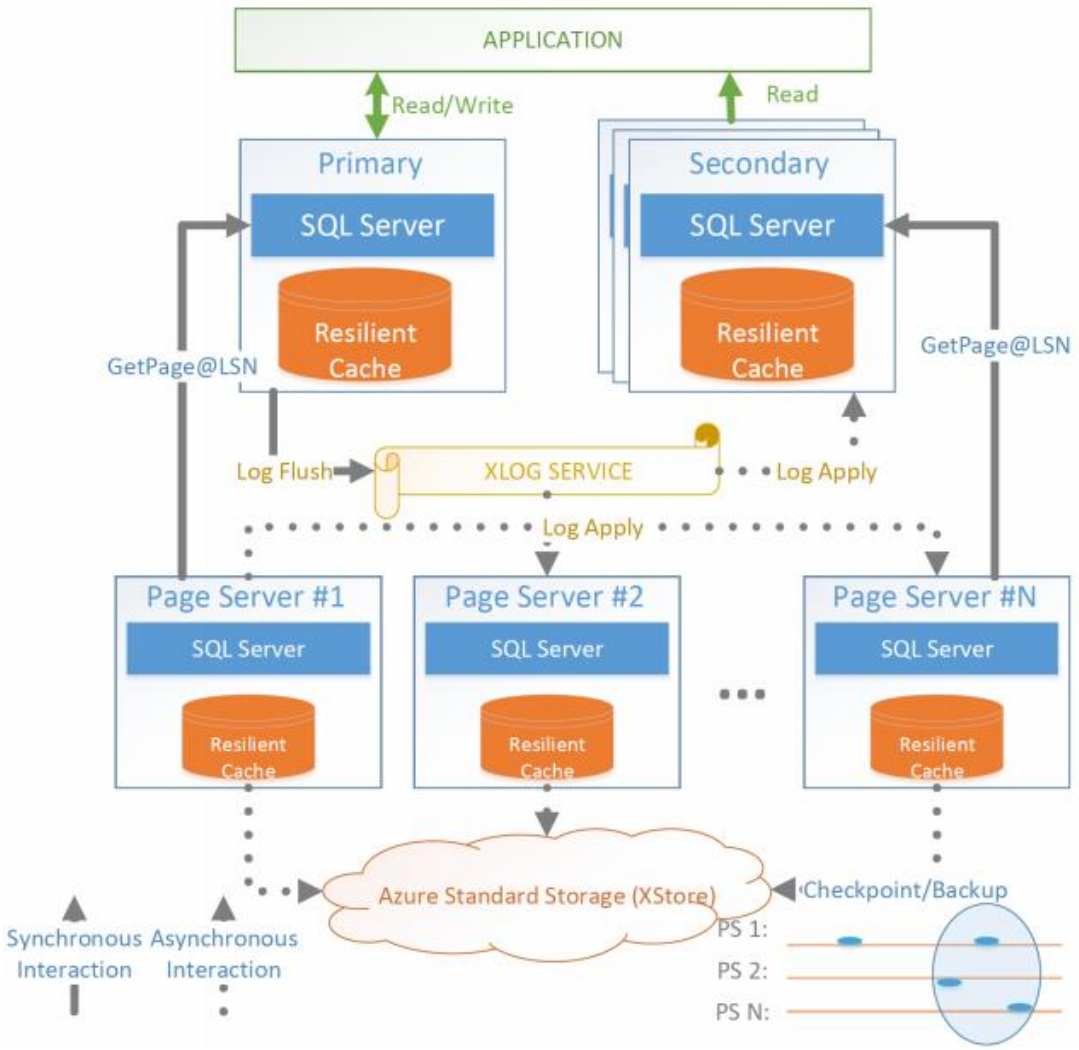
02 Amazon Aurora总结

最早（2017）提出云原生数据库的 LOG IS DATABASE 方案，
为后来工业界云原生数据库的发展提供了借鉴

目录

CONTENTS

03 Microsoft Scorates



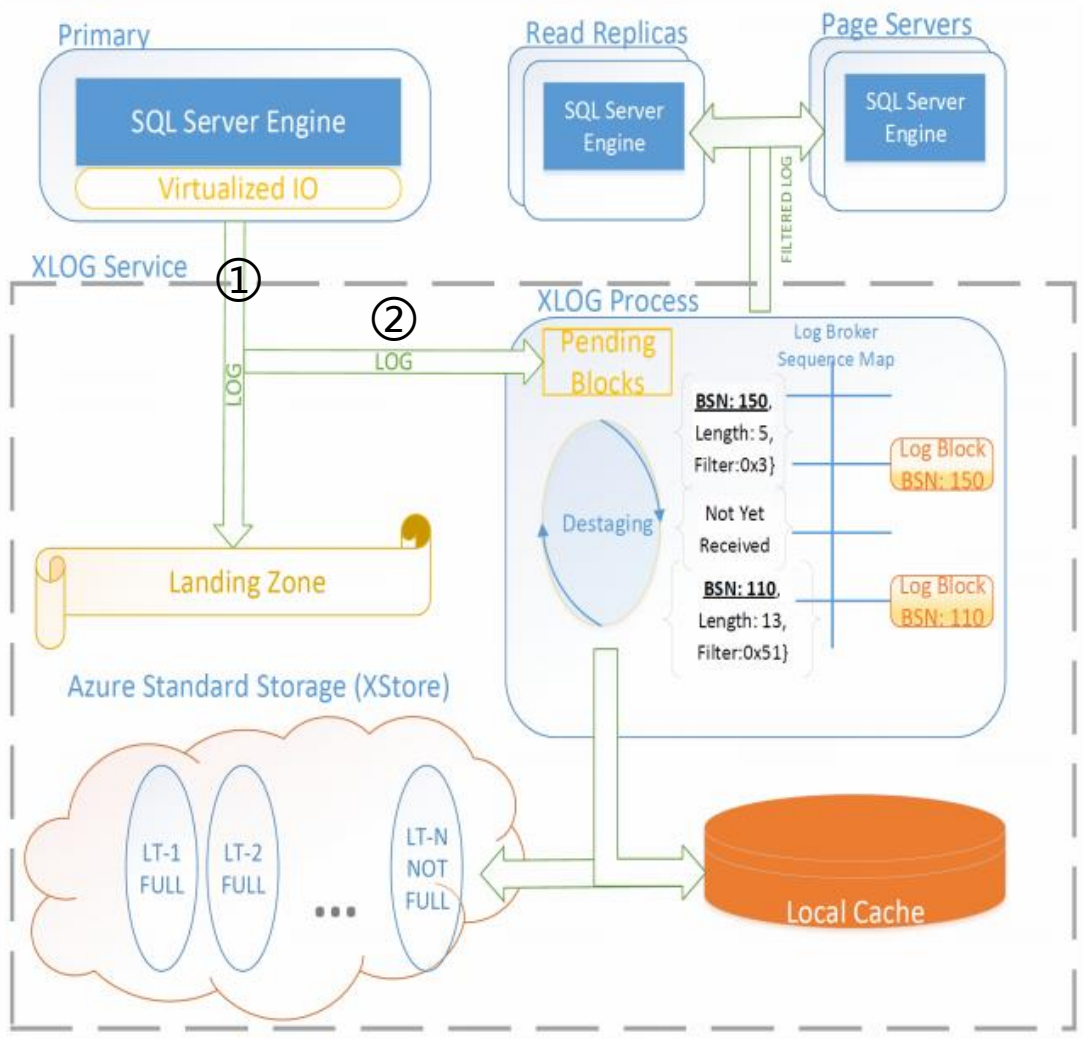
- 计算层

一个Primary节点处理所有的读写事务，任意数量的Secondary节点处理只读事务。如果Primary宕机，任选一个Secondary作为新的Primary，从而实现故障恢复。所有计算接节点都把data page缓存在本地主存和SSD共同组成的缓冲池Resillent cache

- XLOG SERVICE

实现了整个系统的“日志分离”。日志是任何OLTP数据库系统的潜在瓶颈。在提交事务之前，必须对每个更新进行日志落盘，并且必须将日志发送到数据库的所有副本以保持一致。如何在云上为数据库提供一种高性能的日志解决方案？Socrates将日志系统抽离出来，单独做成XLOG SERVICE：

03 Microsoft Scorates



Primary计算节点写日志的路径有两条：

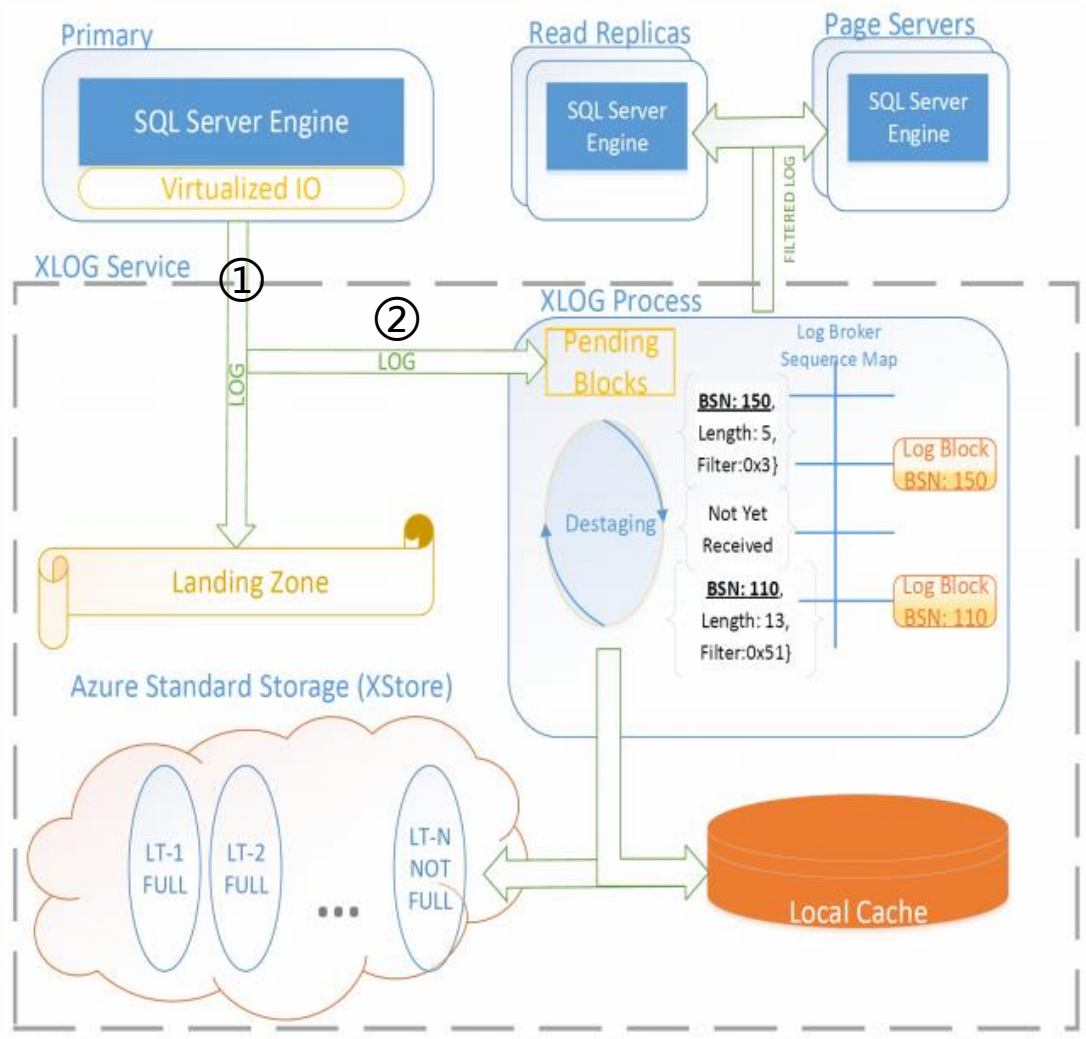
①直接将日志写入Landing Zone(LZ),这是一个Quorum系统，提供高速的持久化存储服务，提供数据完整性、可恢复性、一致性保证。LZ内部组织是一个循环缓冲区，日志格式采用传统Microsoft SQL Server格式。

②将日志写到XLOG Process，由它将日志发送到Secondary节点和Page Servers。

①是同步的，为了保证持久性；

②是异步的，为了保证可用性和日志归档备份。

03 Microsoft Scorates

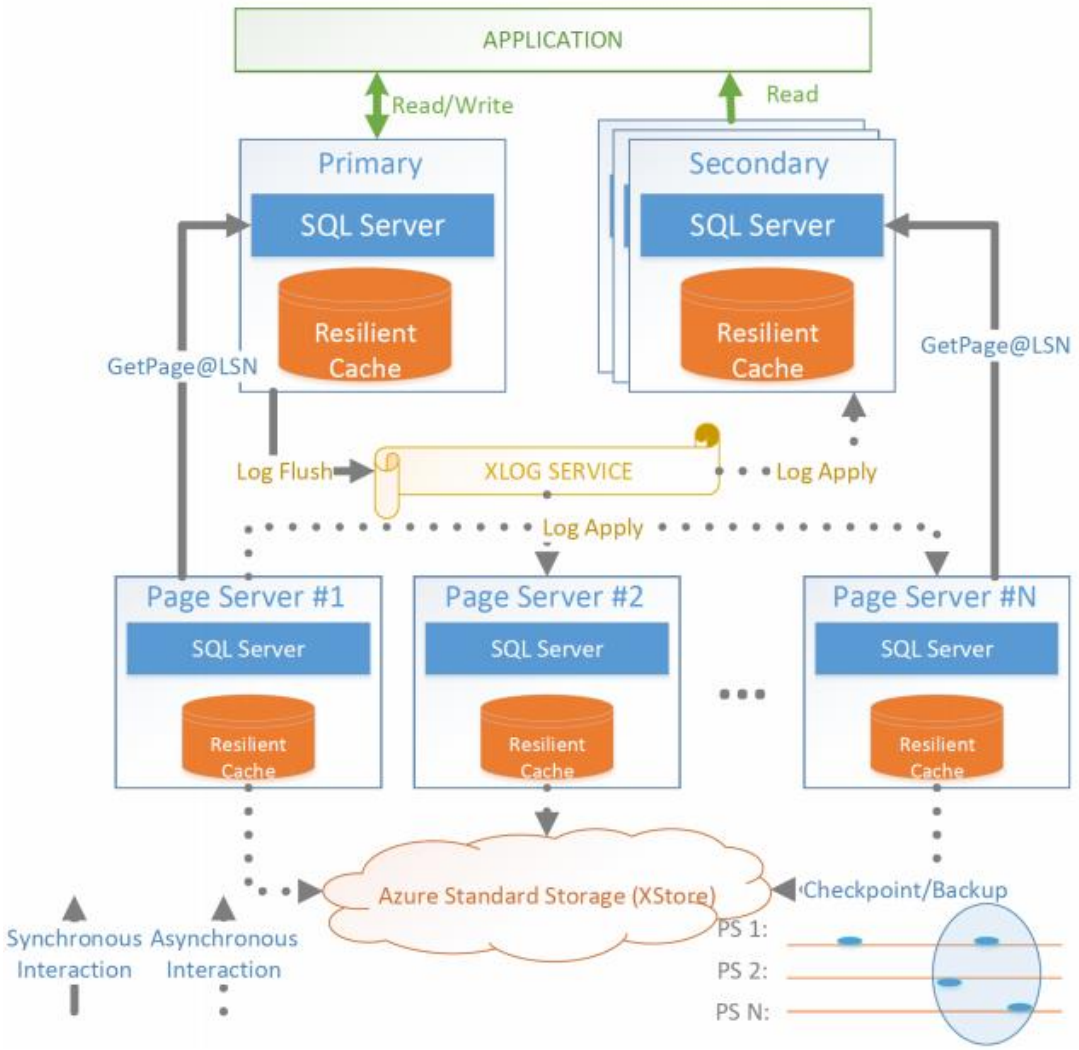


①②两条路径是并发的，所以容易造成数据不一致：

如果日志沿路径②到达Secondary节点并持久化后，路径①的日志尚未持久化到LZ。如果LZ宕机导致日志持久化失败，则可能导致Primary/Secondary数据不一致。为了解决这个问题，**XLOG只向外部传播成功持久化在LZ上的日志**(with write quorum)。

实现方法是：首先将日志写入XLOG Process的pendding area，一旦日志持久化到LZ，XLOG就将其从pending area移动到LogBroker准备发送出去。此外还会将这些日志写到本地SSD cache用于快速存取，以及远端Xstore用于归档备份。

03 Microsoft Scorates

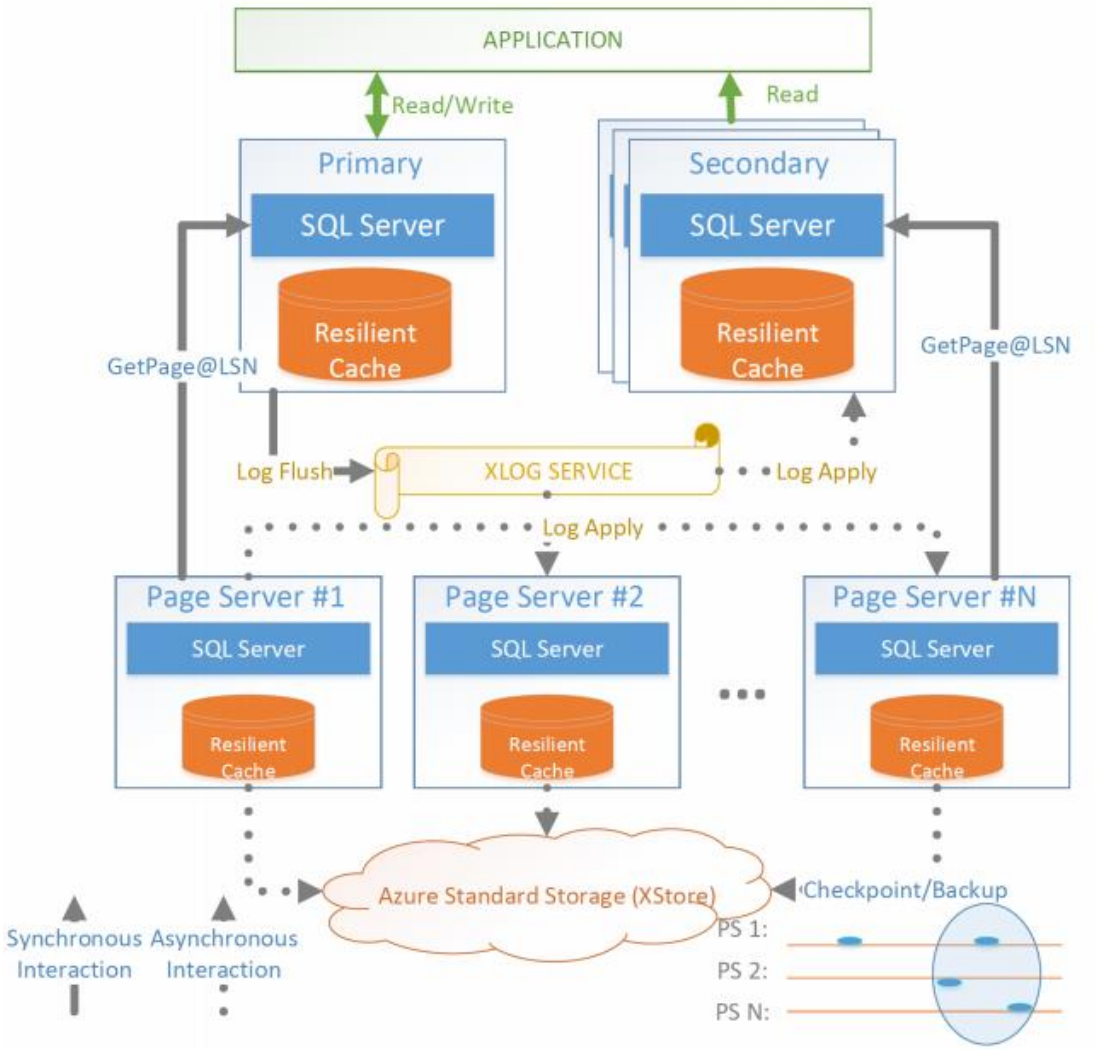


- 存储层Page Servers

每个Page Server都持有数据库的一个分区(partition)，从而实现横向拓展(scale-out)存储架构。Page Servers的两个重要作用：

- 为计算节点提供pages存取服务。每个计算节点都可以按照shared-disk架构从Pager Servers请求页面。（为什么使用shared-disk架构？Socrates的设计理念之一就是要支持海量存储，为每个节点都配置海量的本地存储是不现实的。Shared-disk架构还带来一个好处，可以把计算层的存储任务（backup, checkpoint, etc.）下推到存储层，缓解系统性能瓶颈）。
- 对page做checkpoint，并备份到下一层的Xstore。和计算节点一样，Page Servers把数据存储在主存和本地SSD，以便快速存取。

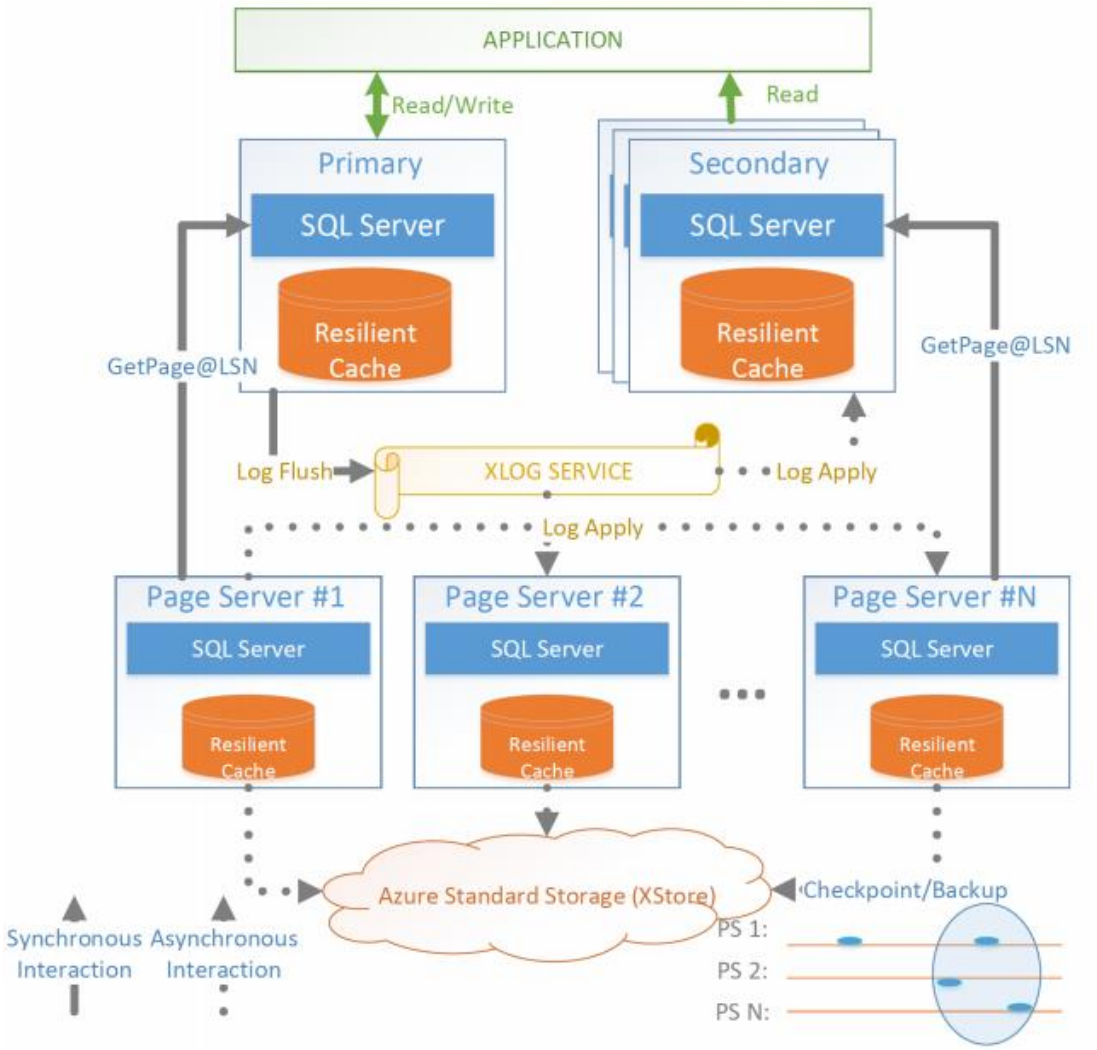
03 Microsoft Scorates



- Azure Standard Storage(Xstore)

使用廉价的大容量HDD。将其与Page Servers的本地SSD分离，体现了存储架构的设计思想：高速存储（SSD）是性能需要，廉价的大容量存储（HDD）是持久性和大规模扩展的需要。

03 Microsoft Scorates



计算节点和Page Servers是无状态的，可随时宕机随时恢复，不会导致数据丢失。因为数据库的“本体”位于XLog和XStore。

03 Microsoft Scorates总结

Amazon Aurora提出“log is database”的思想，Socrates进一步将其发扬光大，把日志从存储层分离出来，减轻了存储层的负担。

目录

CONTENTS

04 Alibaba PolarDB & PolarDB Serverless

三种典型云数据库架构：

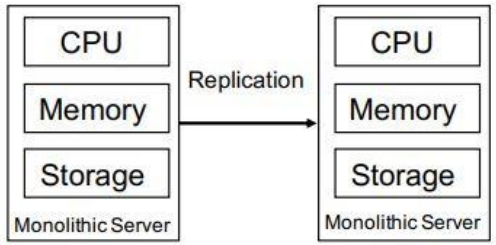


Figure 1: *monolithic machine*

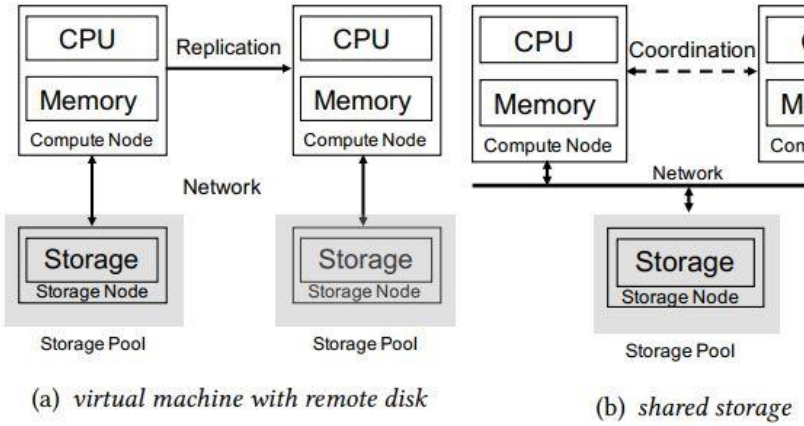


Figure 2: *separation of compute and storage*

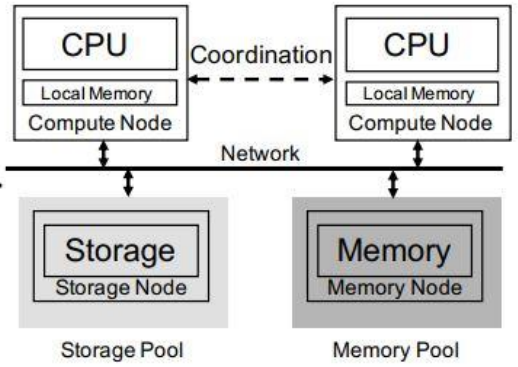


Figure 3: *disaggregation*

PolarDB Serverless 是一种新的disaggregation架构，比shared storage更进一步，将CPU和内存资源从单台机器上解耦出来。位于remote memory pool的内存资源可被多个数据库实例共享。添加新的read replica除了少量的本地内存外，不会消耗更多的内存资源。

04 Alibaba PolarDB & PolarDB Serverless

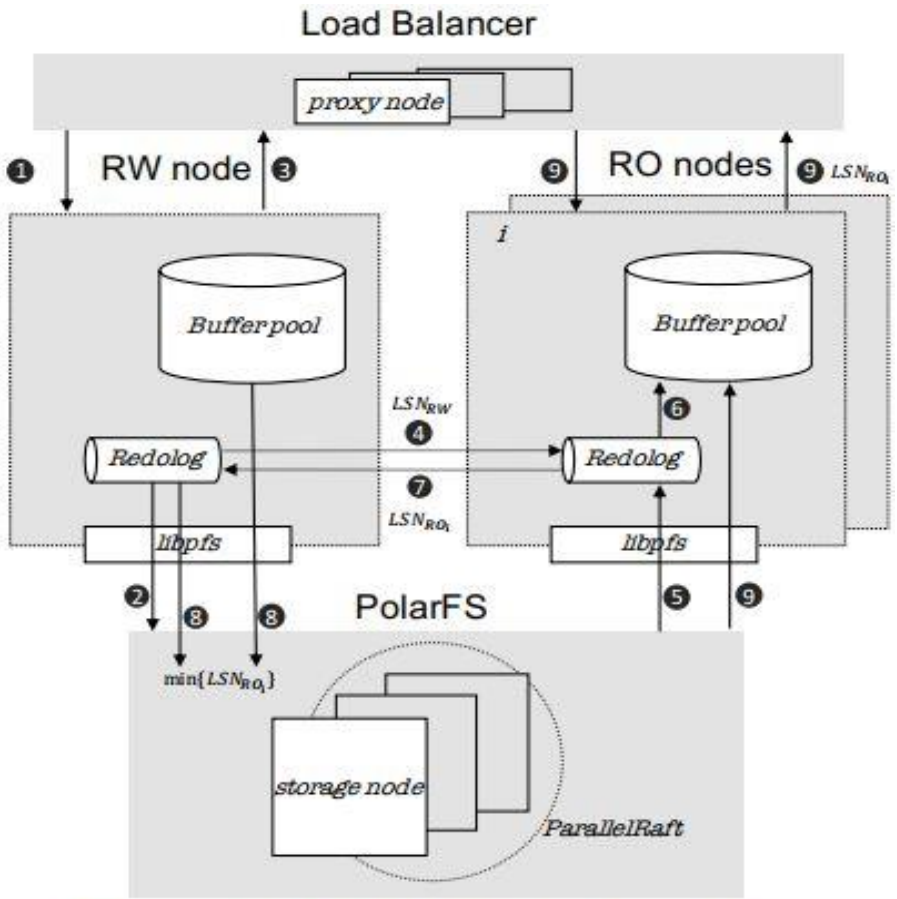


Figure 4: PolarDB Architecture

- PolarDB是一个采用shared storage架构的云原生数据库，衍生自MySQL，采用分布式文件系统PolarFS作为底层存储池。它在计算层包含一个主节点（RW node）和多个读副本（RO nodes）。和传统数据库内核一样，每一个RW和RO节点包含buffer pool、索引、SQL执行引擎、事务引擎。另外还包括用作负载均衡的proxy node。
- PolarFS是一个可横向扩展（scale-out）的分布式文件系统，以chunk为存储单位进行管理，每个chunk是都有三个副本，采用ParallelRaft协议。

04 Alibaba PolarDB & PolarDB Serverless

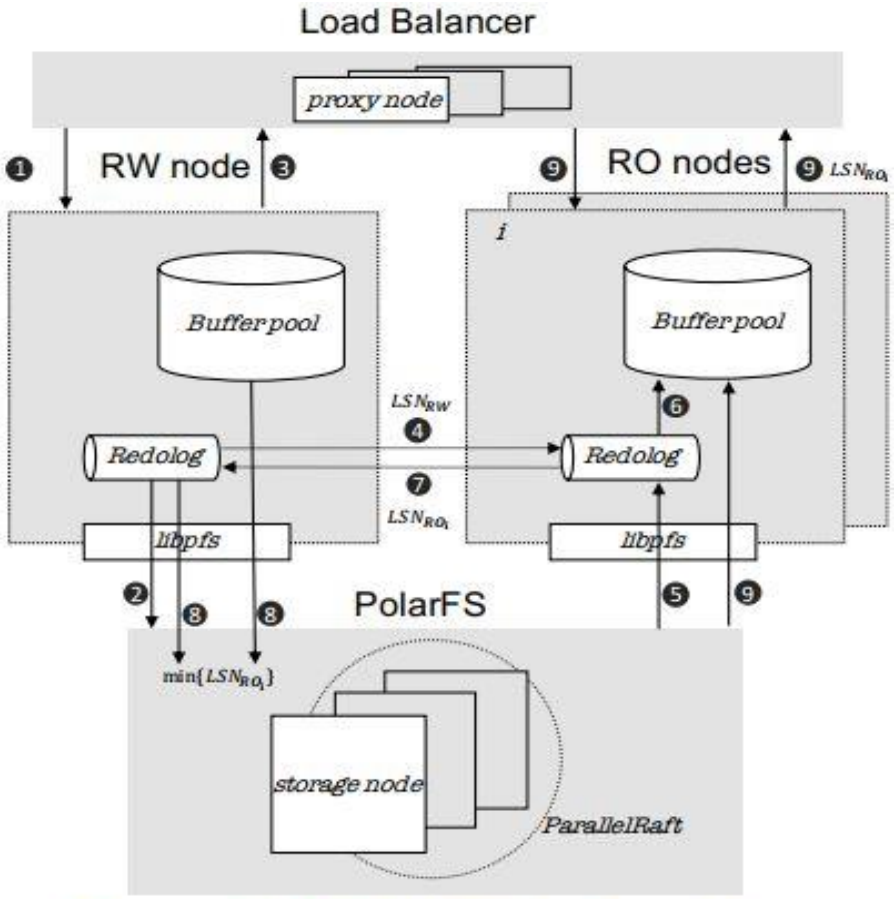


Figure 4: PolarDB Architecture

- RW和RO使用redo logs进行内部状态的同步。
- RW处理事务①时，把redo logs持久化到PolarFS后②，事务方能提交③。
- RW把redo logs和最新的LSN_{RW}异步地广播给RO④。当RO_i接收到RW的消息后，就会从PolarFS把redo logs拉取下来⑤，然后应用到本地buffer pool里的pages⑥。
- RO_i会回复RW，它自己把redo logs应用到了哪个位置（LSN_{ROi}）⑦。
- RW会把本地{min LSN_{ROi}}之前的redo logs删掉，再把page LSN < {min LSN_{ROi}}的脏页刷到PolarFS ⑧。
- RO_i在处理只读事务时使用快照隔离，读到的数据版本位于LSN_{ROi}之前⑨。
- 如果存在一个RO_k，它的LSN_RO_k远小于LSN_RW，就说明RO_k延迟过大，需要将其从集群中踢出，避免拖慢集群的性能。

04 Alibaba PolarDB & PolarDB Serverless

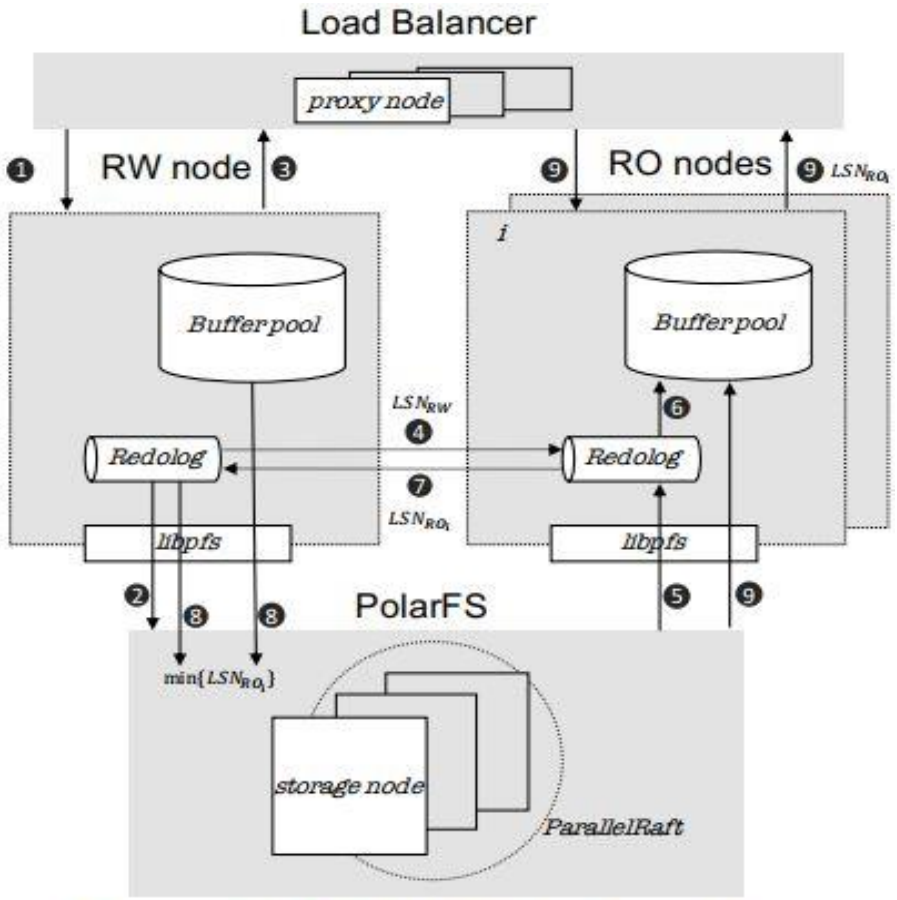


Figure 4: PolarDB Architecture

PolarDB借鉴了Amazon Aurora “log is database”的思想，网络中传输的数据只有redo log。

04 Serverless Database

Serverless是cloud-native的高度弹性化衍生。使用pay-as-you-go模型，实现资源的自动调配—动态扩展和缩减。可分为两种：

- auto-scaling 资源的自动扩展和缩减
- auto-pause 当以存储层任务为主的时候，释放计算层资源；反之亦然。

auto-scaling的缺陷：由于大多采用的是shared-storage架构，CPU和内存高度耦合，对不同场景的适应性欠佳。例如，对于OLTP，高并发的查询非常消耗CPU，但是所涉及的数据量不会很大，因此对内存的需求不会很大。对于OLAP，查询执行的频度很低，因此对CPU要求不高。但是单条查询通常需要做全表扫描，所以非常耗内存。**在disaggregation架构下，CPU和内存解耦，资源分配也更加灵活高效。**

auto-pause的缺陷：由于计算节点的CPU和内存资源高度耦合，所以必须统一释放、统一恢复—恢复时延被拉长。**在disaggregation架构下，CPU和内存解耦，使得计算节点释放资源时只会释放CPU，remote memory pool保持不变，因此恢复时更为快速。**

04 PolarDB Serverless

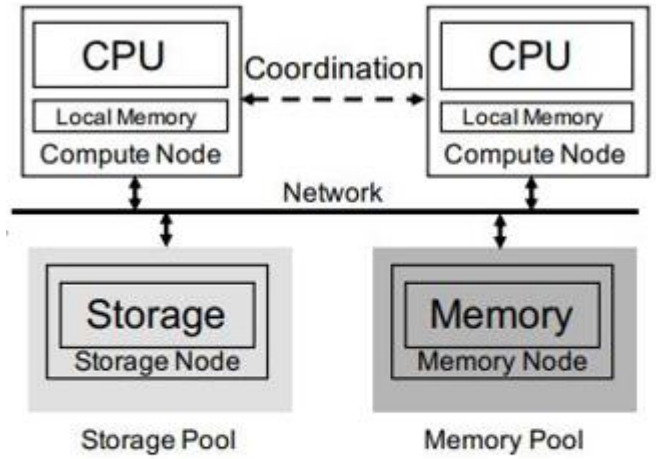


Figure 3: *disaggregation*

- PolarDB Serverless和PolarDB最大的不同在于使用了disaggregation架构，增加了remote memory pool。目前在remote memory pool里存储的是pages。由于内存被单独拎了出来，所以水平扩展能力更强，能够支持更大的内存容量，能将更多的数据缓存在内存里。remote memory的主要瓶颈在于：
  - 访问速度远低于local memory。解决方案：使用分层内存系统、预取(prefetching)机制、速度更快的RDMA。
  - 由于remote memory成为了RW和RO的共享资源，所以需要提供互斥访问机制。
  - remote memory pool存储的是pages，是否需要直接在网络中传输pages？不需要，依然是遵照**LOG IS DATABASE**思想，只传输redo log和必要的metadata。

04 PolarDB Serverless

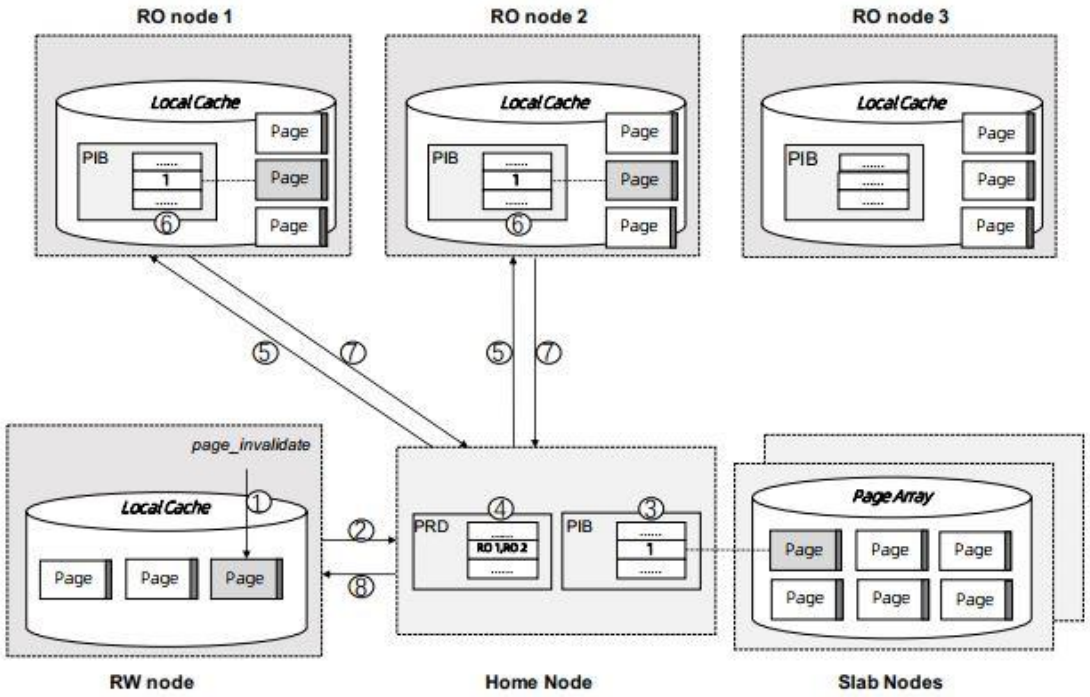


Figure 6: Cache Invalidation

local cache和remote memory的交互

由于网络延迟原因，不可能每次读写操作都直接操作remote memory，所以需要将一些pages缓存在本地。local cache的大小设置为 $\min\{1/8 * Size_{remote\_memory}, 128GB\}$，这是性能和开销的权衡。

- 如果被访问的pages未驻留在remote memory，计算节点（RW/RO）将从存储节点把pages读取过来，然后再将其写回remote memory。内存节点和存储节点之间不直接通信。
- 并非从存储节点取得的全部pages都需要写回remote memory，例如进行全表扫描时读取的pages在近期再被使用的可能性较低，如果将这些pages全部写入remote memory就容易造成缓存污染。
- 如果发生local cache miss，RW/RO就需要从remote memory或remote storage读取pages，速度肯定比local memory/storage慢。因此需要设计预取机制提高性能。
- 如果local cache满了，就按照LRU将pages替换出去，并写回remote memory。

04 PolarDB Serverless

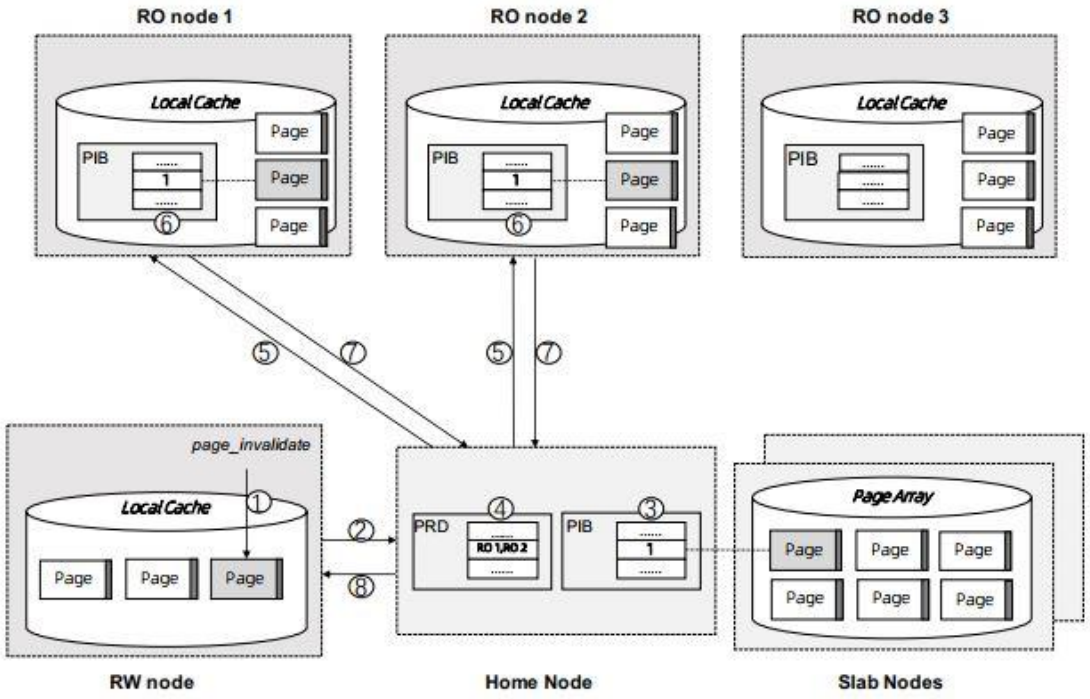


Figure 6: Cache Invalidation

Cache Coherency

- remote memory pool里的pages被RW和RO共享。对于一个写事务，如果RW能立刻把pages写回remote memory后，RO就不再需要重放redo logs了。但是出于降低响应时延的考虑，RW会把更新先写入local cache，而不会立即同步到remote memory。
- 另外，RO也会在local cache里持有pages副本。这就容易产生cache不一致问题。
- 解决方案：RW在local cache修改了pages之后，告诉remote memory和RO它修改了哪些pages，把那些pages标记为“已过期”。

04 PolarDB Serverless

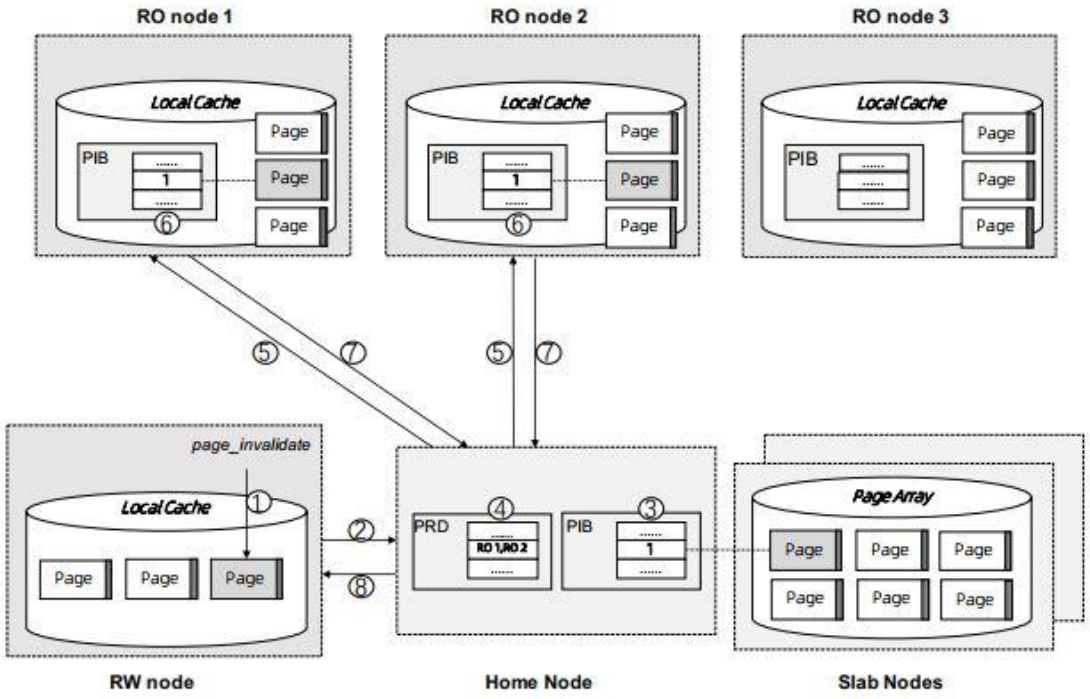


Figure 6: Cache Invalidation

Cache Coherency

- RW更新完它的local cache之后，调用*page_invalidate*函数①，
- 在Home Node的PIB（Page Invalidation Bitmap）里将相应的page标记为无效③，
- 查找PRD（Page Reference Directory）找到持有这个page副本的RO，再将这些RO本地的PIB里对应page的bit置1，表示page无效⑥。
- *page_invalidate*是同步阻塞操作，只有当所有分区上的PIB都设置完毕才能返回⑦⑧。如果每一个RO出故障导致操作超时，他就会被踢出。

04 PolarDB Serverless 计算层与存储层的交互

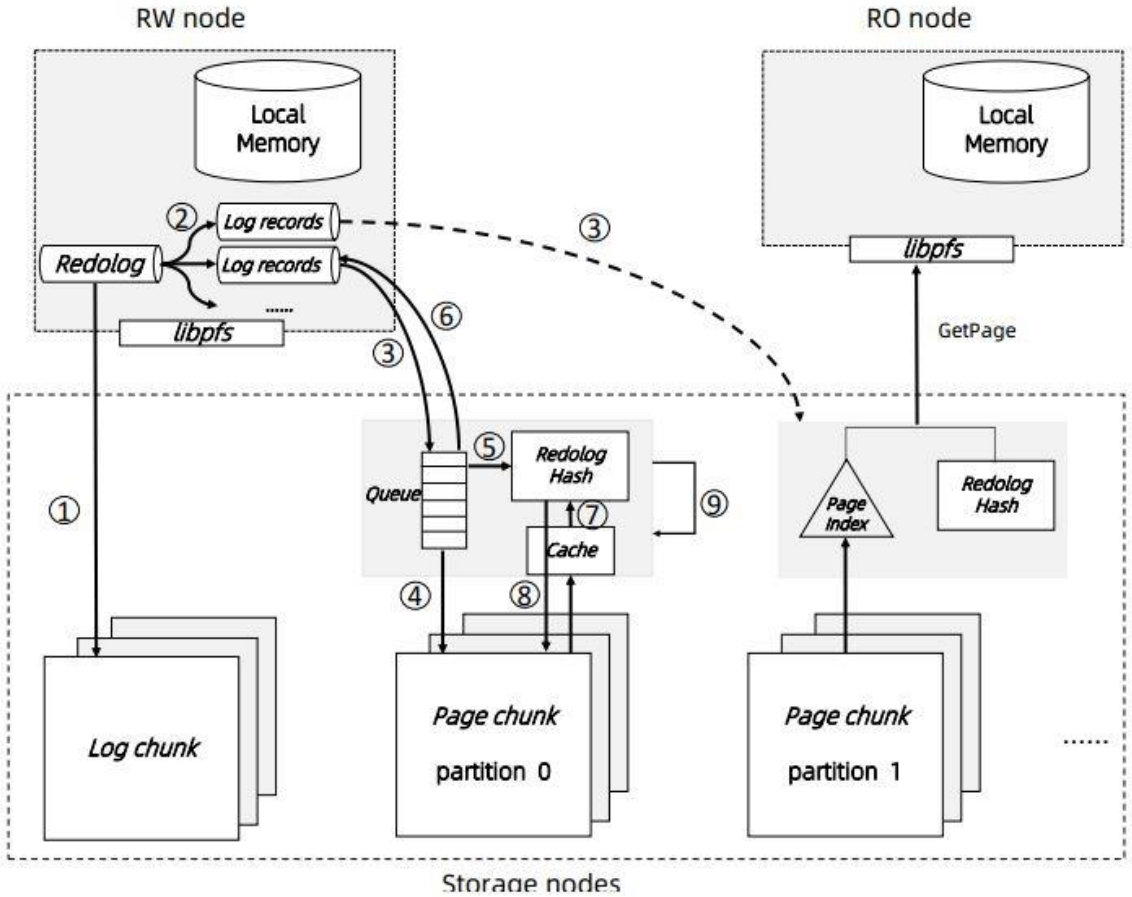


Figure 7: Page Materialization Offloading

- PolarDB Serverless类似于Socrates，log和page分开存储
- 在PolarFS中log和page分别存储在两种chunk里：log chunk和pages chunk，使用三副本，ParallelRaft
- redo log首先刷到log chunk，然后异步地发送到page chunk，从而将更新应用到pages

04 PolarDB Serverless 计算层与存储层的交互

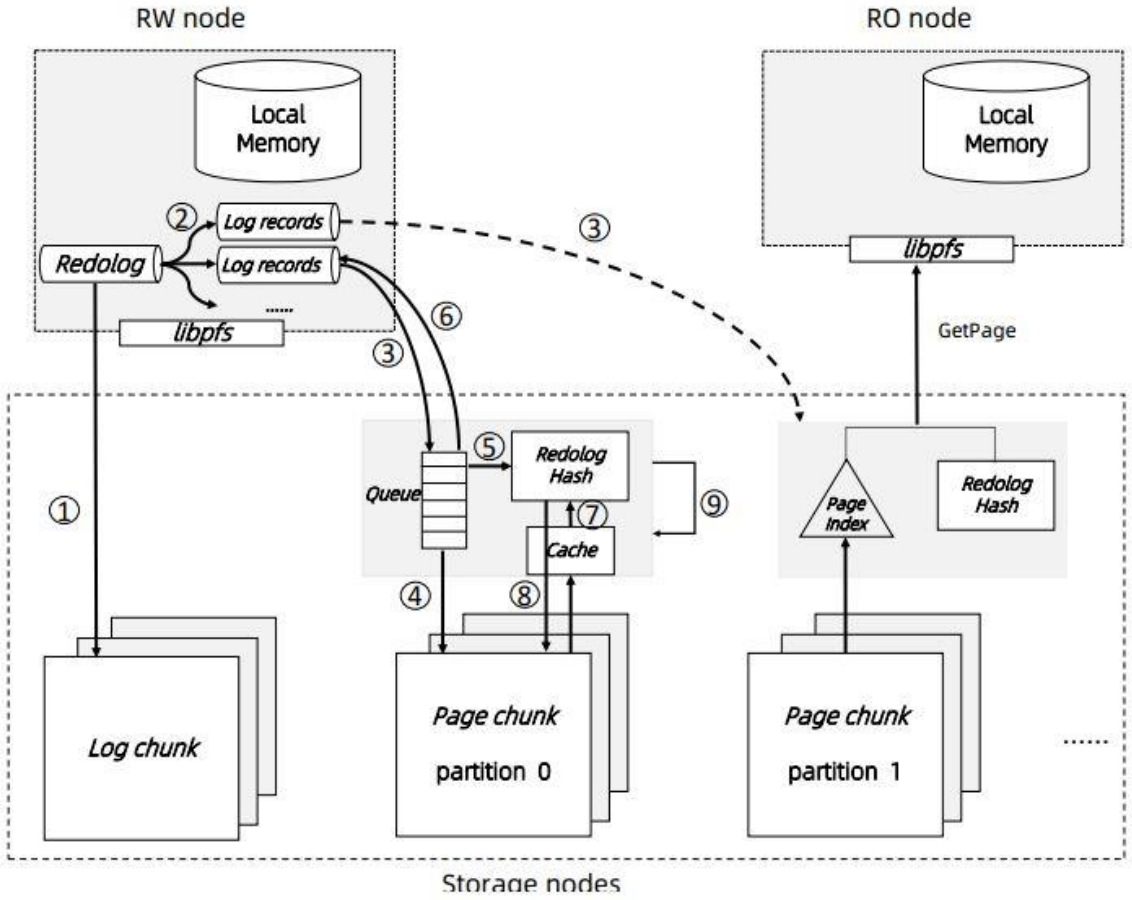


Figure 7: Page Materialization Offloading

- 写事务到达RW后，redo logs被持久化到log chunks之后事务就可被提交①。
- RW把redo logs分解成log records②，按照log record涉及的page id，将其分为若干组，每一组log records映射到一个page chunk partition。
- 这些log records会被发送到对应的page chunk partition完成**第二次**持久化③④。然后将log records写入内存哈希表（page id为key）⑤，最后给RW返回ACK⑥。RW在收到ACK之前会一直把dirty page缓存在local cache。
- 在存储节点后台，旧版本pages从cache或磁盘读出⑦，重放log records得到新的pages，再将其写回⑧。
- 底层存储引擎会把一个page的不同版本保留一段时间，用以支持按“时间点”的恢复。⑨是后台GC任务，负责收集redo logs和旧版本pages。

04 PolarDB Serverless 计算层与存储层的交互

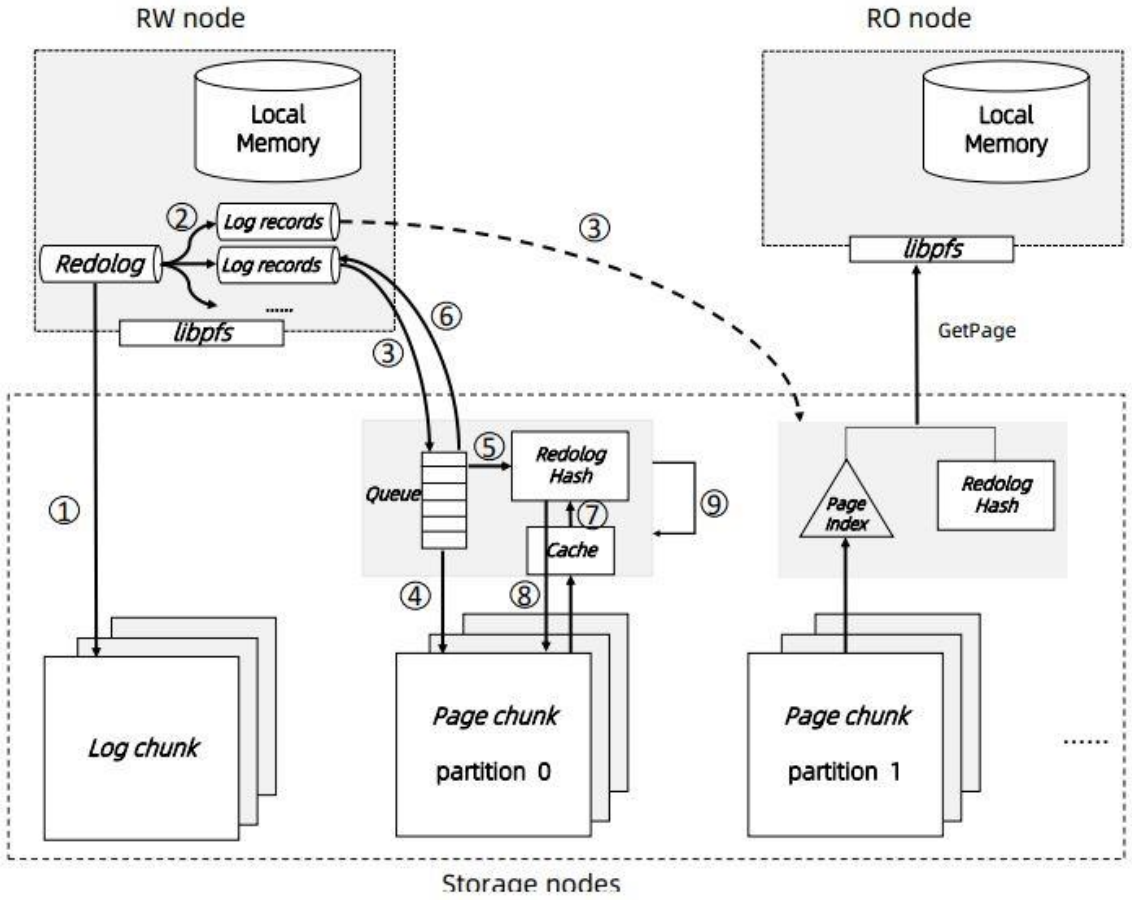


Figure 7: Page Materialization Offloading

存储节点响应GetPage请求时，把cache和disk里的page合并，然后检查Redolog Hash是否包含该page的log records，如果有的话就将其应用到page，最后将最新的page返回给上层。

04 总结

- 云原生数据库的核型逻辑：
  - 按照分布式存储方案进行部署
  - 将不同类型的资源解耦，根据业务需求自由地对计算、存储资源进行扩缩容等操作
- Amazon Aurora —— Log is Database
  - Amazon Aurora对传统架构下MySQL上云方案的改进
  - 只在网络中传输Redo Log和必要的Metadata，降低网络IO
- Microsoft Scorates
  - 扩展了 log is database 的思想，经日志服务从存储层分离出来，进一步实现资源解耦
- PolarDB & PolarDB Serverless
  - 借鉴 log is database 思想的同时，进一步实现资源解耦：对计算节点的CPU和内存解耦

实现云原生数据库的高扩展性、高可用性、快速迭代、低成本