



云原生数据库PolarDB

2021.12.17

李林峰



目录

C O N T E N T S

1. 云原生数据库
2. PolarDB
3. PolarDB Serverless
4. 总结

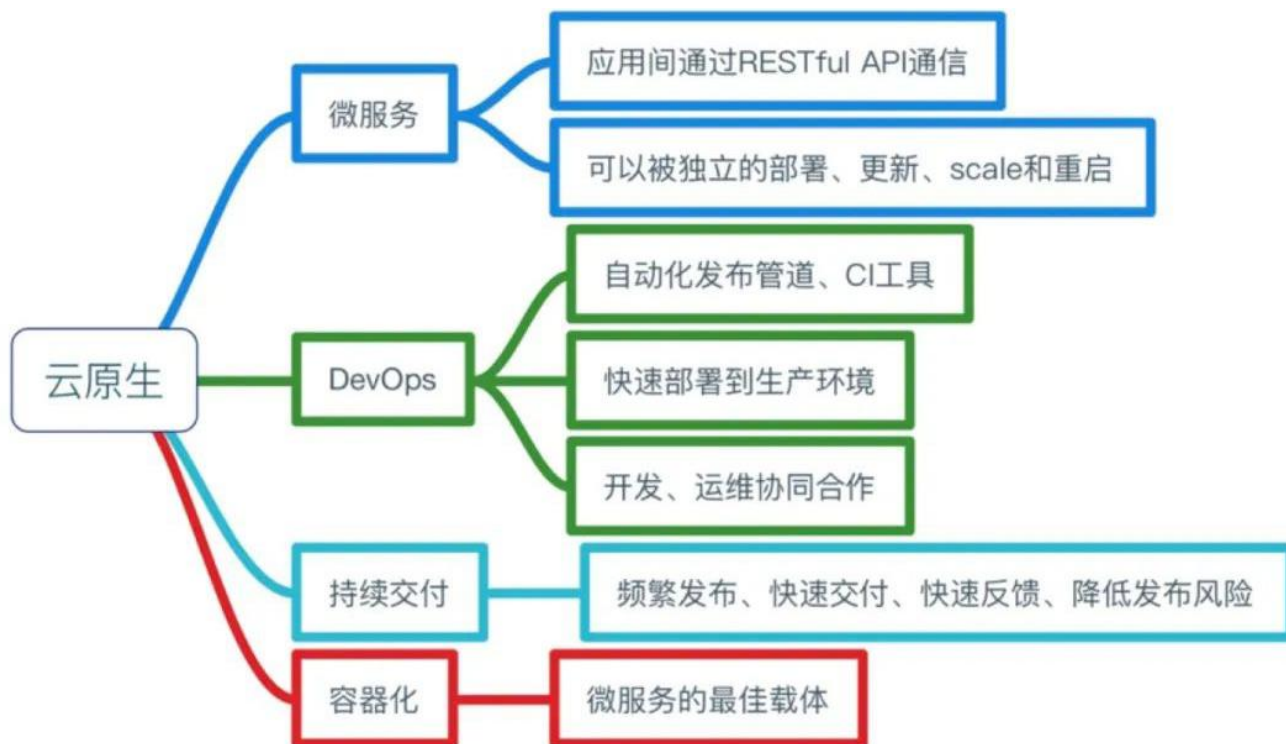


目录

C O N T E N T S

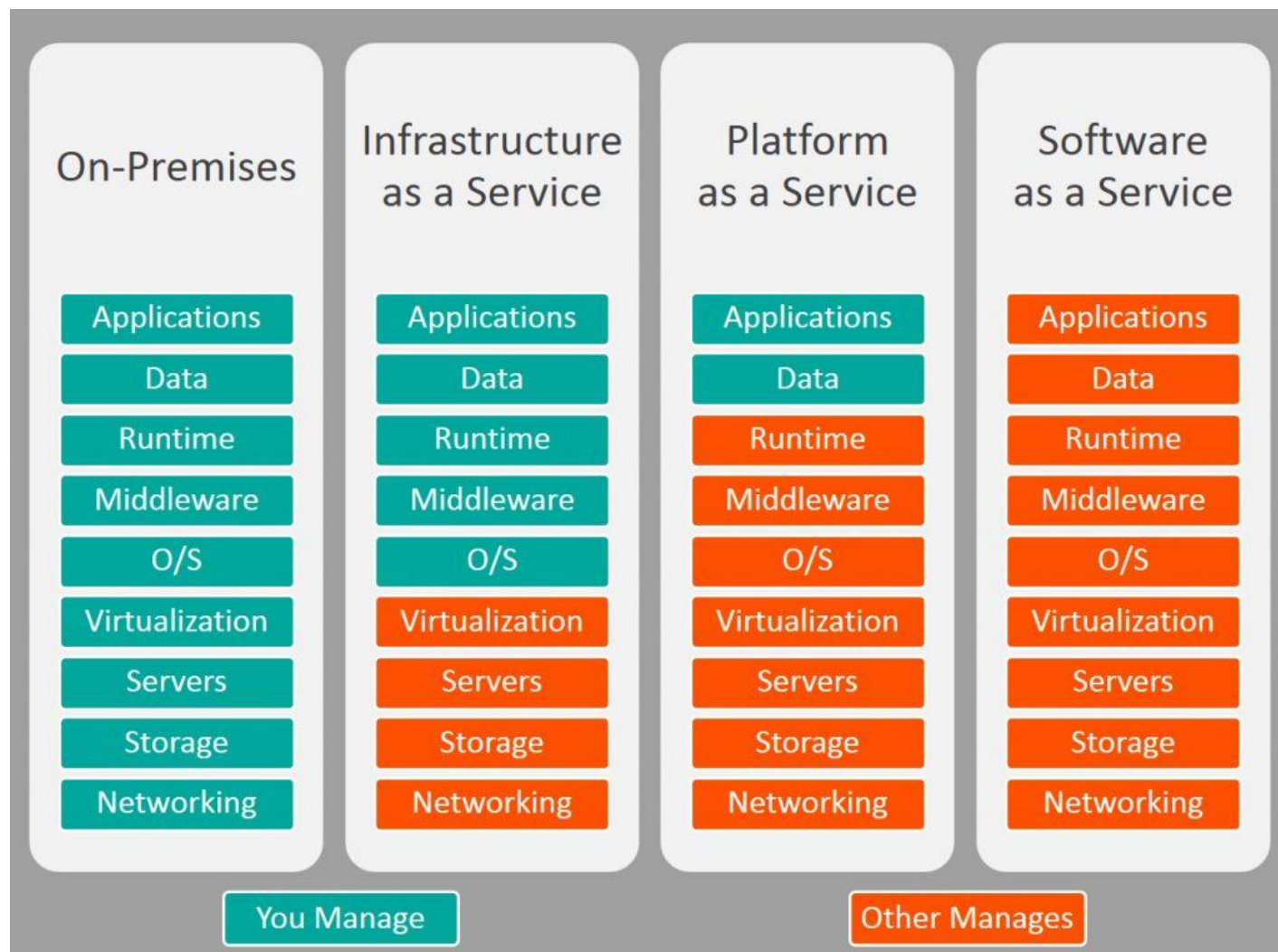
1. 云原生数据库
2. PolarDB
3. PolarDB Serverless
4. 总结

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。



云原生四要素:

微服务、DevOps 、持续交付、容器化

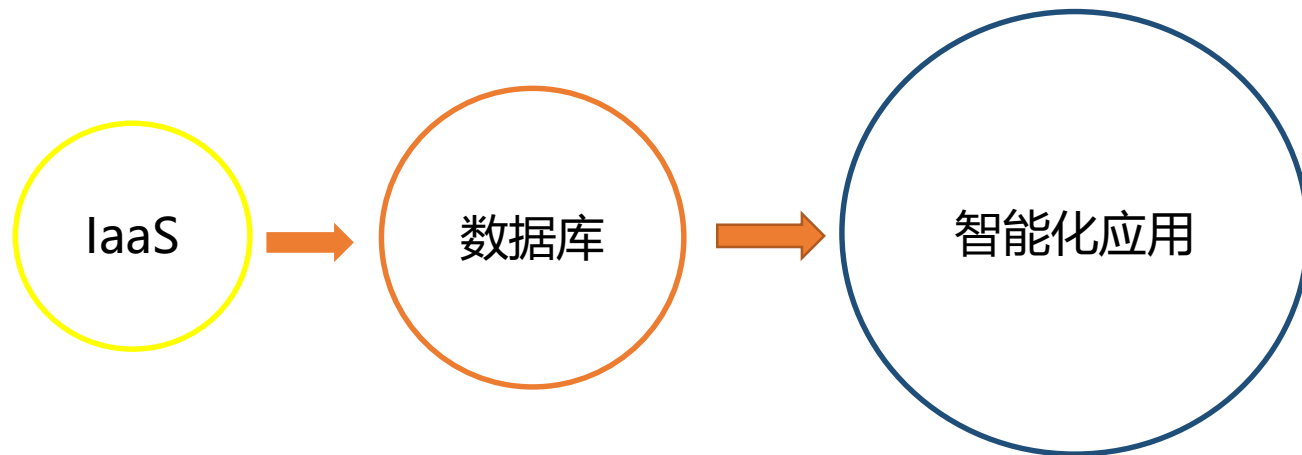


云服务分类：

IaaS、PaaS、SaaS

下一代企业级数据库关键技术

HTAP: 大数据数据库一体化	行列混存+混合负载+分布式计算与分析
智能化	自感知+自决策+自恢复+自优化
软硬件一体化	RDMA/NVM 3DX point
Multi-Model 多模	Json/KV/Text/graph/Time-series/...
安全可信	可验证日志与计算+全链路加密
云原生+分布式	CPU/内存/存储分离 分布式处理



数据的产生，存储和消费

Oracle、Google、Amazon、Apple、Microsoft、IBM | 阿里巴巴、华为、腾讯

1. 借助 IaaS，直接将传统的数据库“搬迁”到云上，资源利用率，维护成本，可用性上都不太理想
2. 云原生数据库特点：
高可用性、安全可靠、动态可扩展、低成本



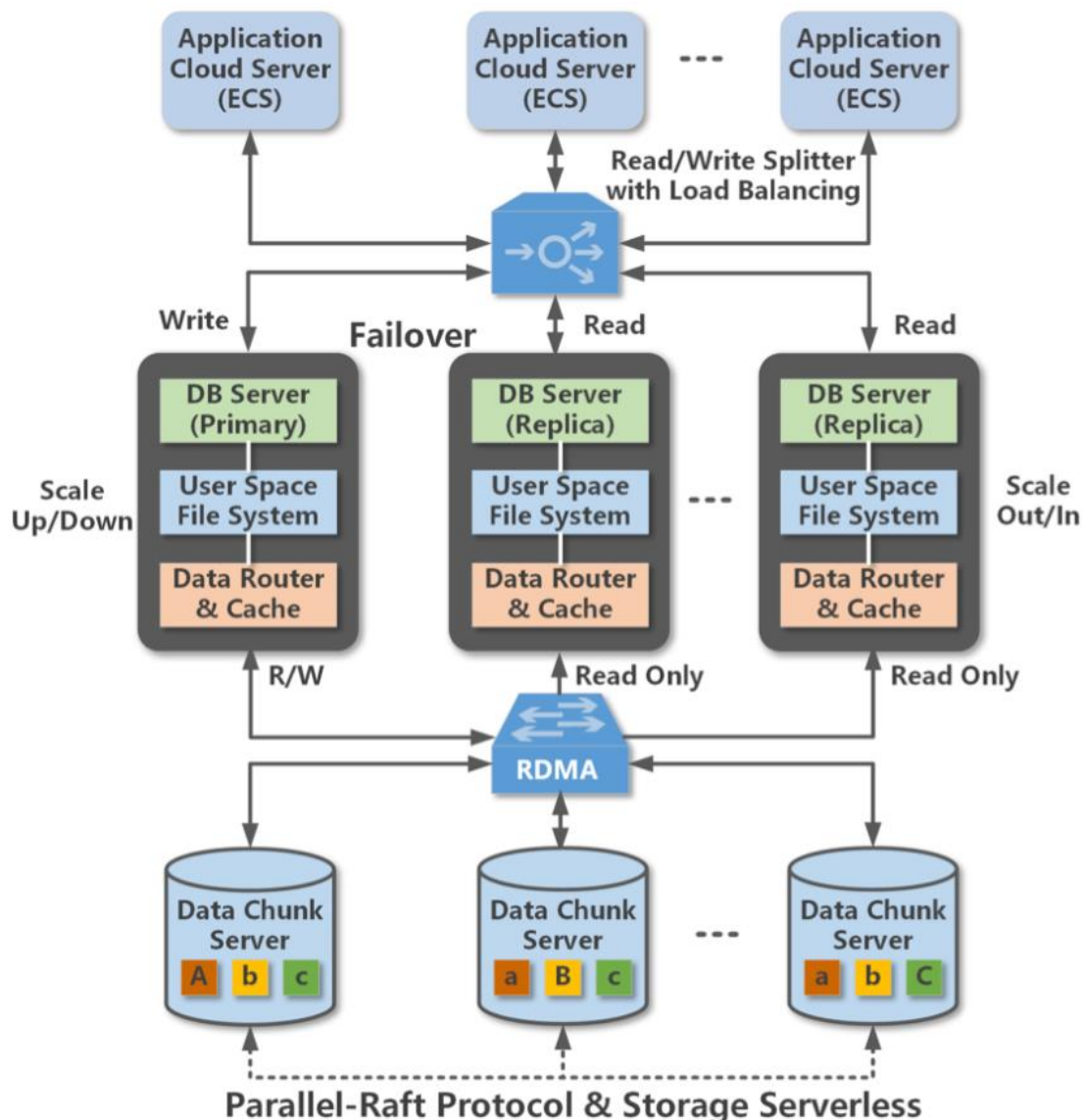
目录

C O N T E N T S

1. 云原生数据库
2. **PolarDB**
3. PolarDB Serverless
4. 总结



02 PolarDB架构



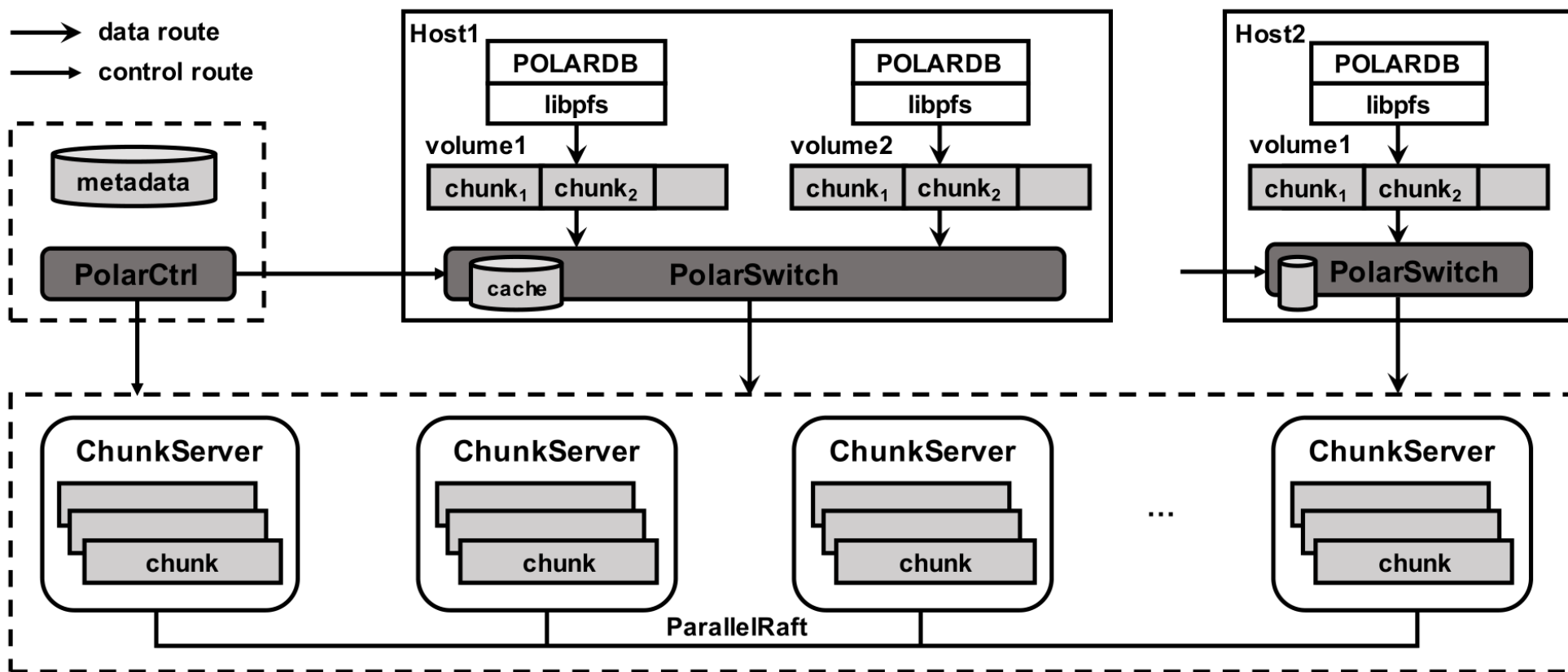
1.共享存储系统PolarFS

2.存储计算分离

2.高可用、高吞吐率 (DRMA、SPDK, parallel raft)

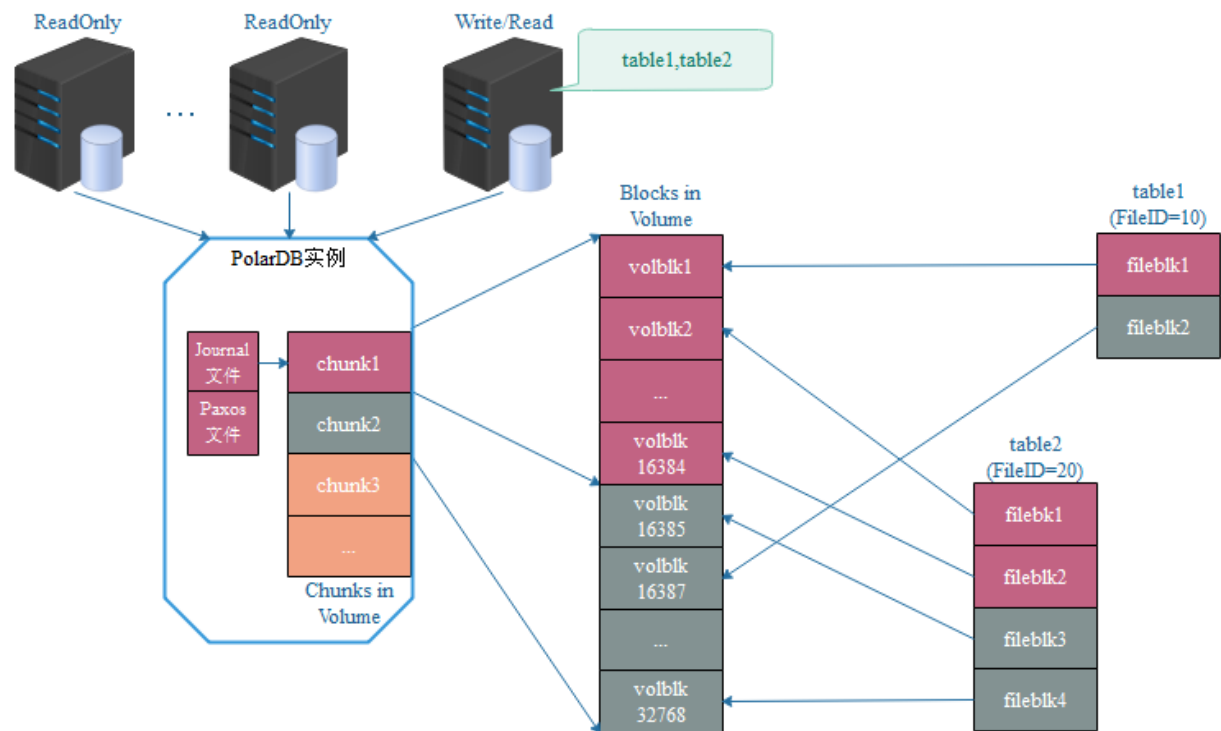
3.100%兼容MySQL,Oracle, Postgre SQL

02 PolarDB架构



1. libpfs
2. Volume
3. PolarSwitch
4. PolarCtrl
5. ChunkServer
6. Chunk

存储结构

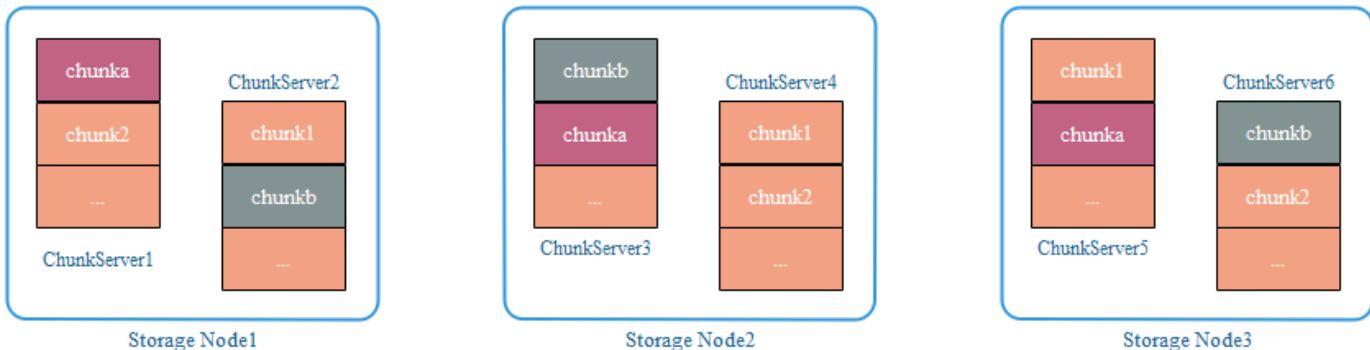


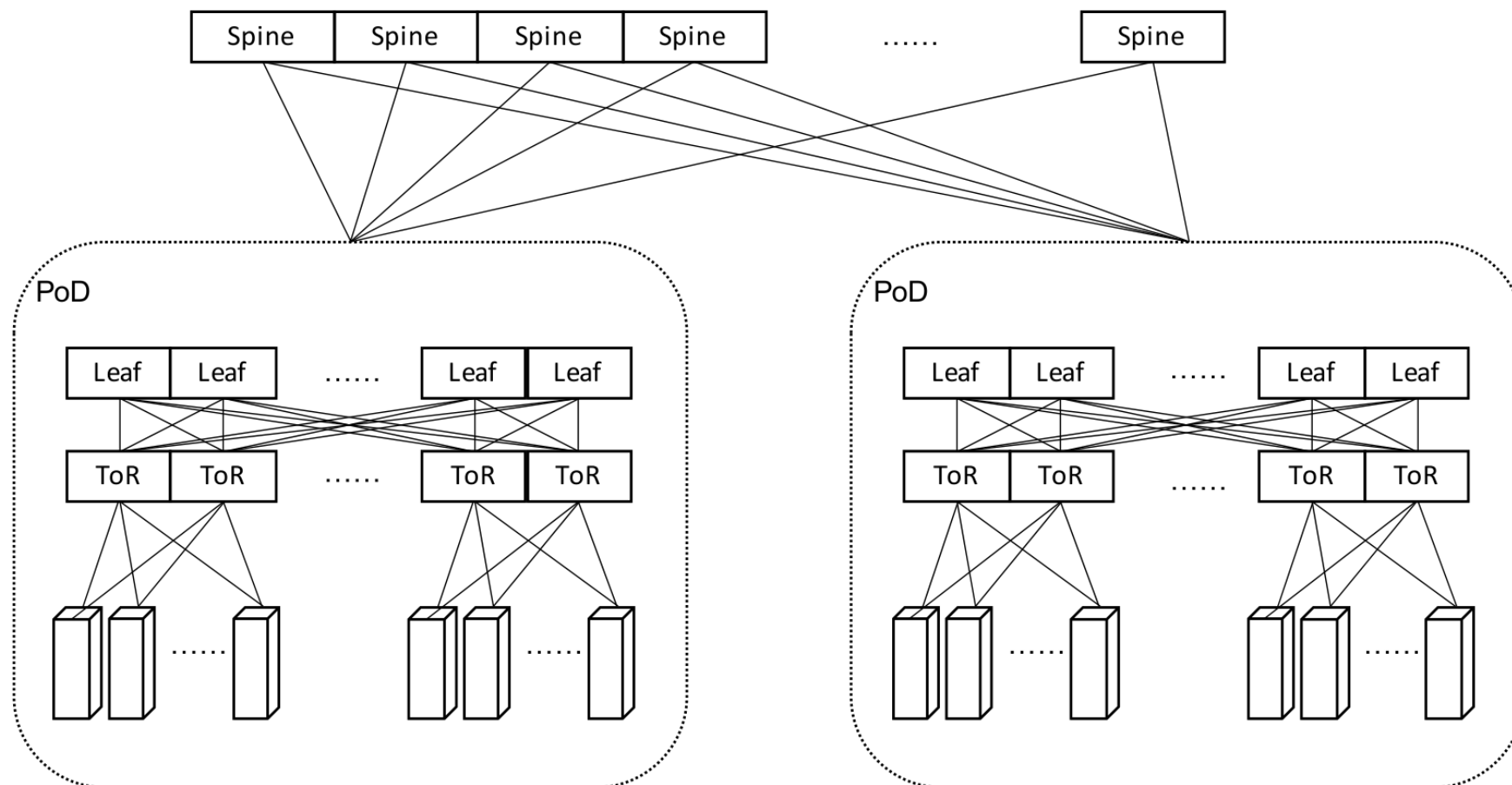
1. 磁盘的Paxos算法实现对Journal文件的互斥访问

2. Chunk大小10GB,可减少volume到chunk映射的元数据量大小

3. 每个Chunk都有3个副本，分布在不同的ChunkServer上

4. 每个ChunkServer绑定到一个CPU核，并管理一块独立的NVMe 盘，ChunkServer之间没有资源竞争





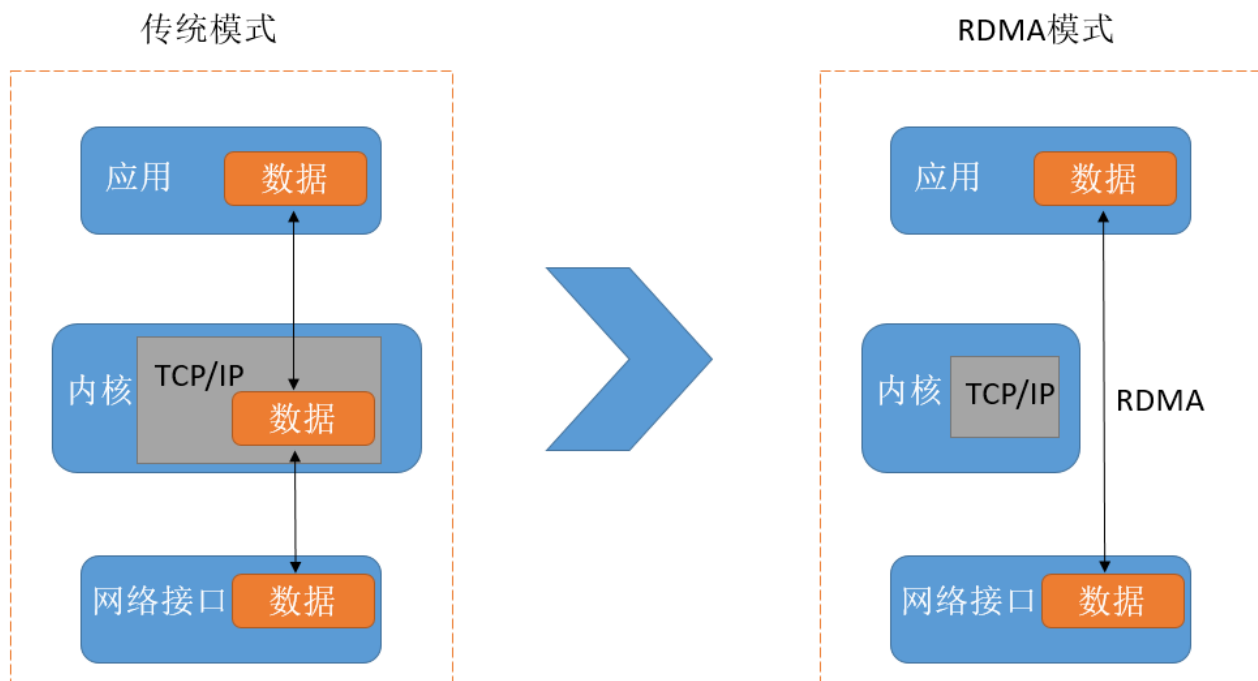
典型的数据中心包括三层:

1.spine layer

2.leaf layer

3.ToR layer

RDMA (Remote Direct Memory Access) 技术是一种**直接在内存和内存之间进行资料互传**的技术，在数据传输的过程中，**不需要和操作系统内核做沟通**，完全实现了 Kernel Bypass (内核旁路)

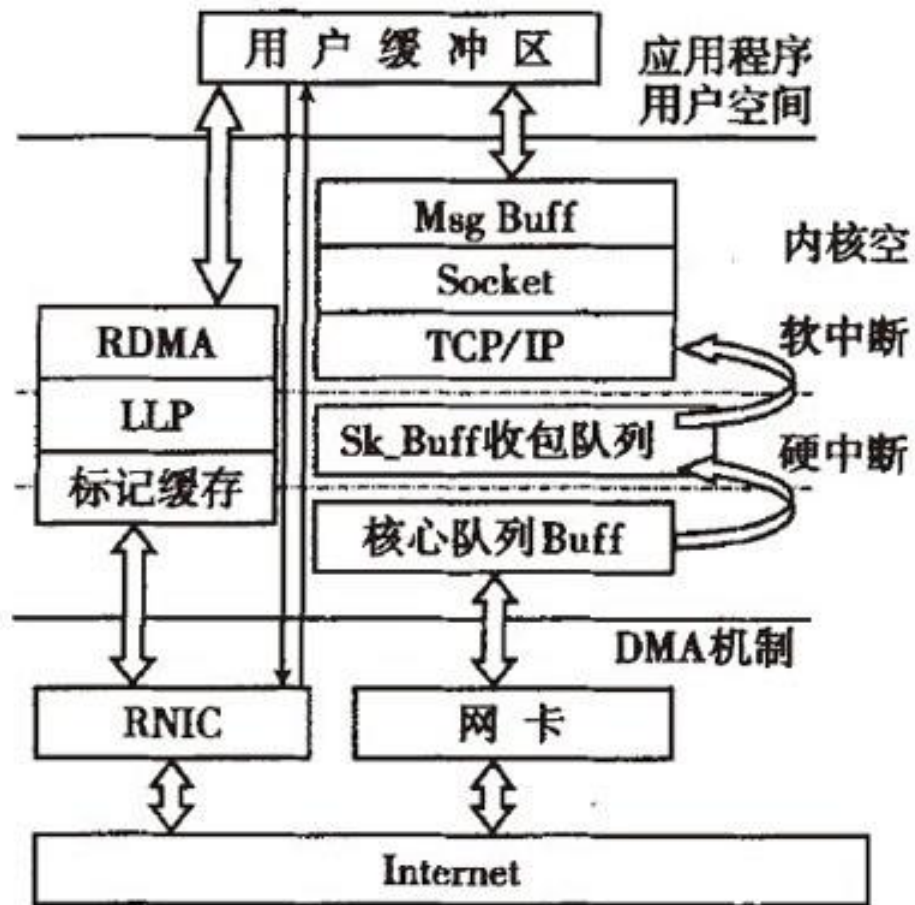


TCP/IP处理开销:

buffer管理

在不同内存空间中消息复制

消息发送完成后的系统中断

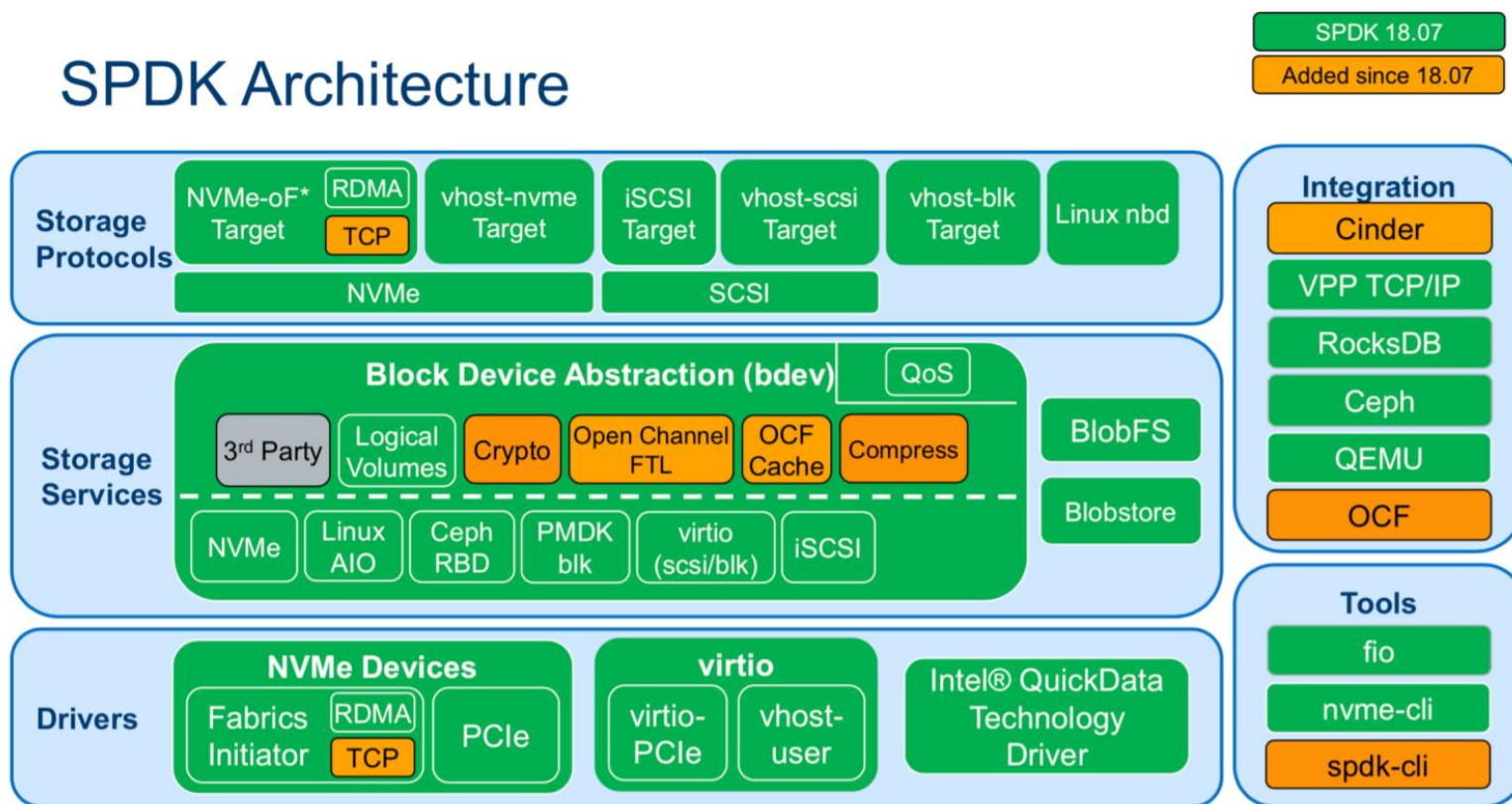


特点:

1. 零拷贝 (Zero-copy)
2. 内核旁路 (Kernel bypass)
3. 不需要CPU干预
4. 消息基于事务
5. 支持分散/聚合条目

旨在帮助客户**优化存储系统软件栈的性能**；主要应用场景是高性能的NVMe本地盘、 Fabric Target盘及虚拟化盘；
针对最新一代的CPU和存储介质的优化

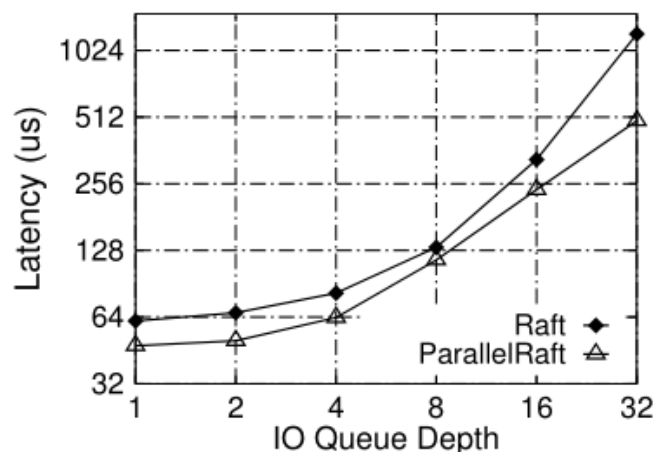
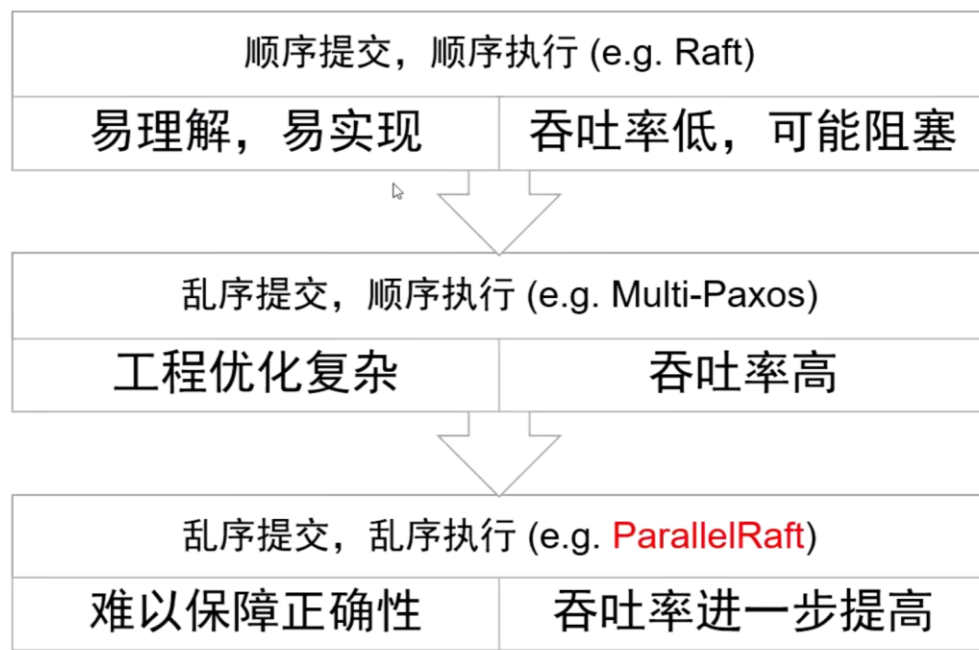
SPDK Architecture



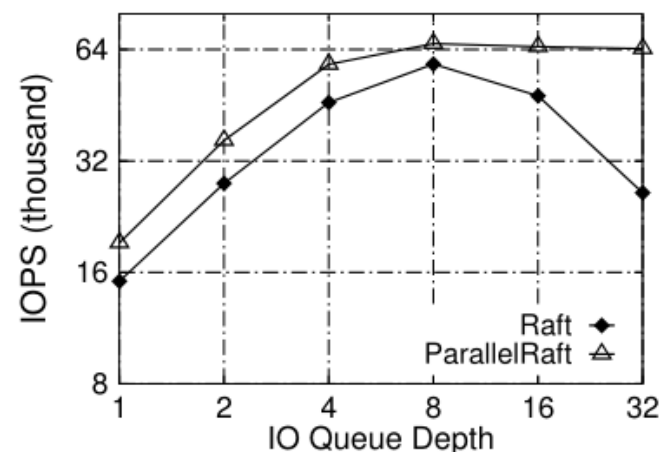
- 1.将存储用到的驱动转移到用户态
- 2.使用轮询模式

02 Parallel Raft

1. 普通raft日志条目由 Follower 确认、Leader 提交并按顺序应用于所有副本。不需要日志空洞，这会增加平均延迟并降低吞吐量
2. Raft 不太适合使用多个连接在主从之间传输日志的环境



(a) The latency impact.

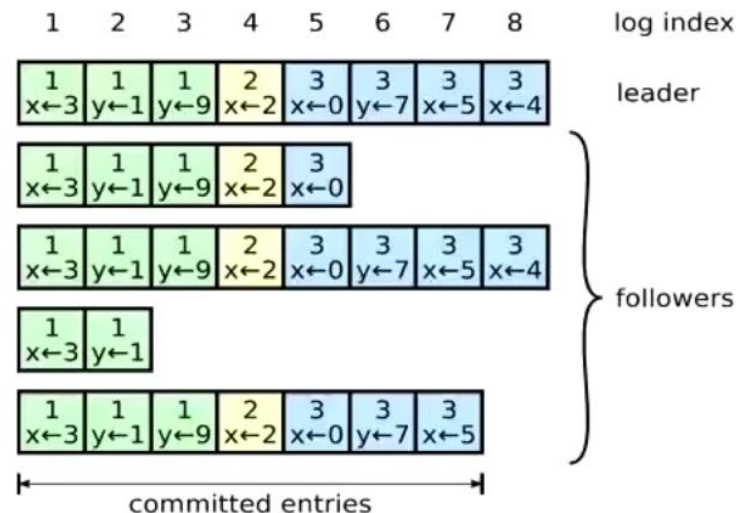


(b) The throughput impact.

02 顺序与乱序

➤ 提交阶段

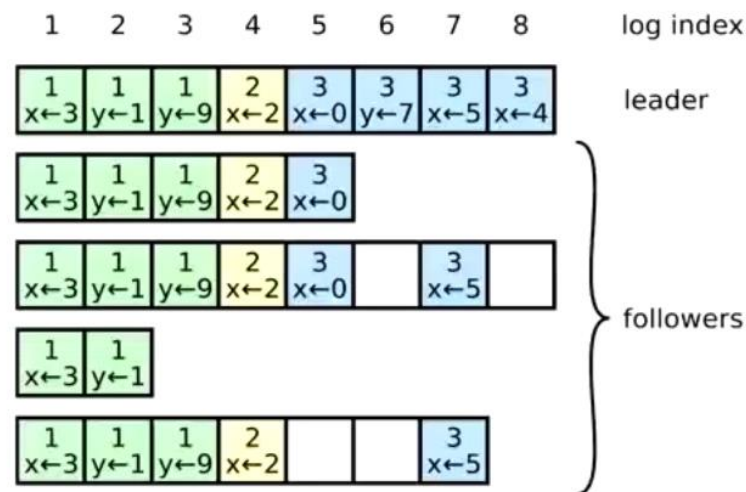
- 顺序提交：日志无空洞
- 乱序提交：允许日志有空洞



顺序提交

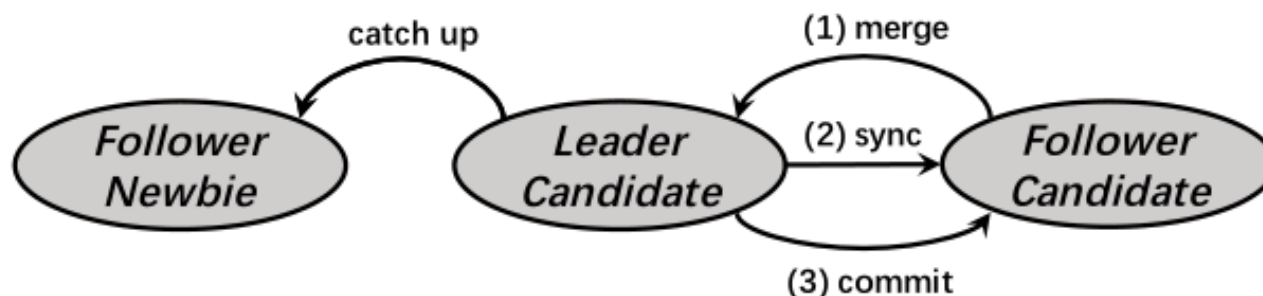
➤ 执行阶段

- 顺序执行：顺序执行日志中每一个已提交的命令
- 乱序执行：允许无冲突的情况下，提前执行靠后的命令

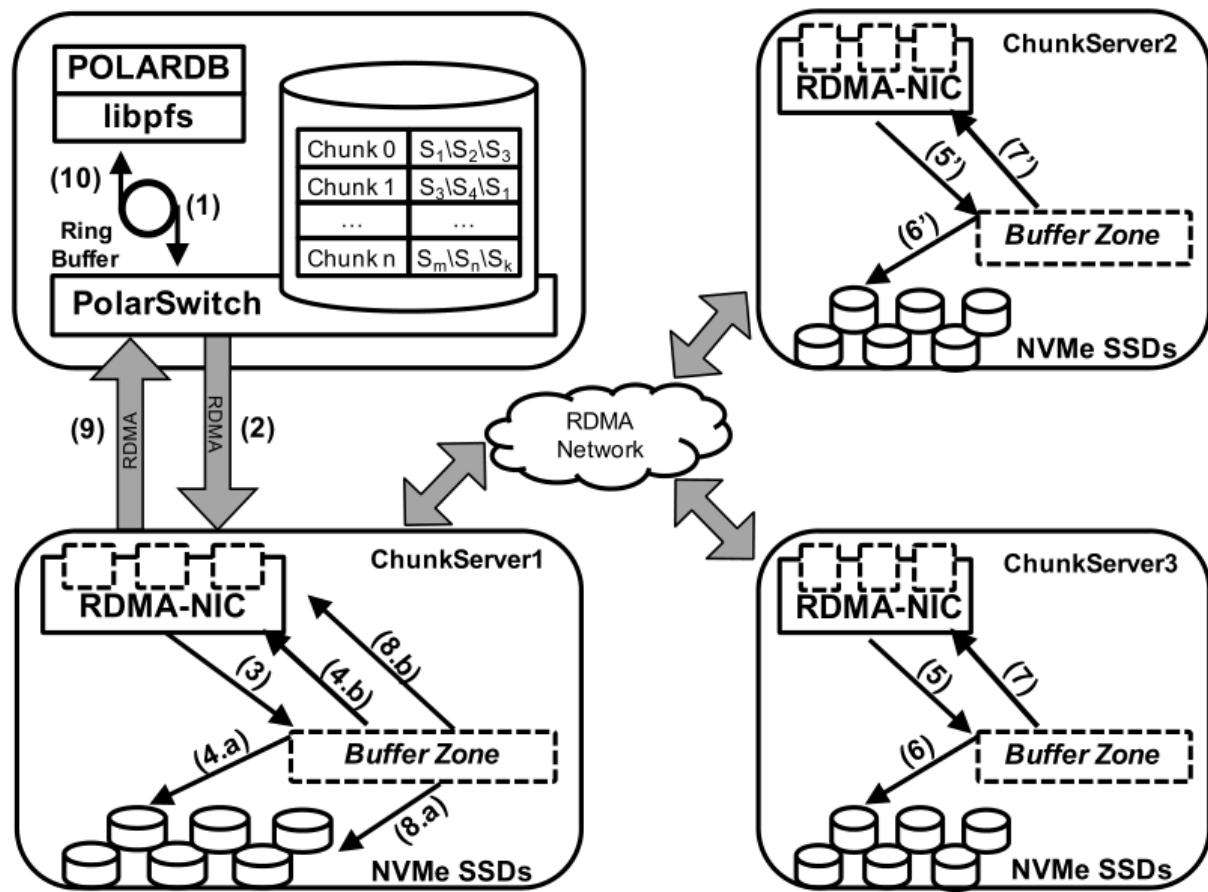


乱序提交

1. 实现乱序提交和执行关键数据结构: **look behind buffer**(存储前N条日志修改涉及的逻辑块地址)
2. 判断某个范围的写入与其它的写入是否重叠来决定是否能按照乱序来执行
3. 选举为leader之后状态是leader candidate, 成为真正的leader之前增加一个merge 阶段

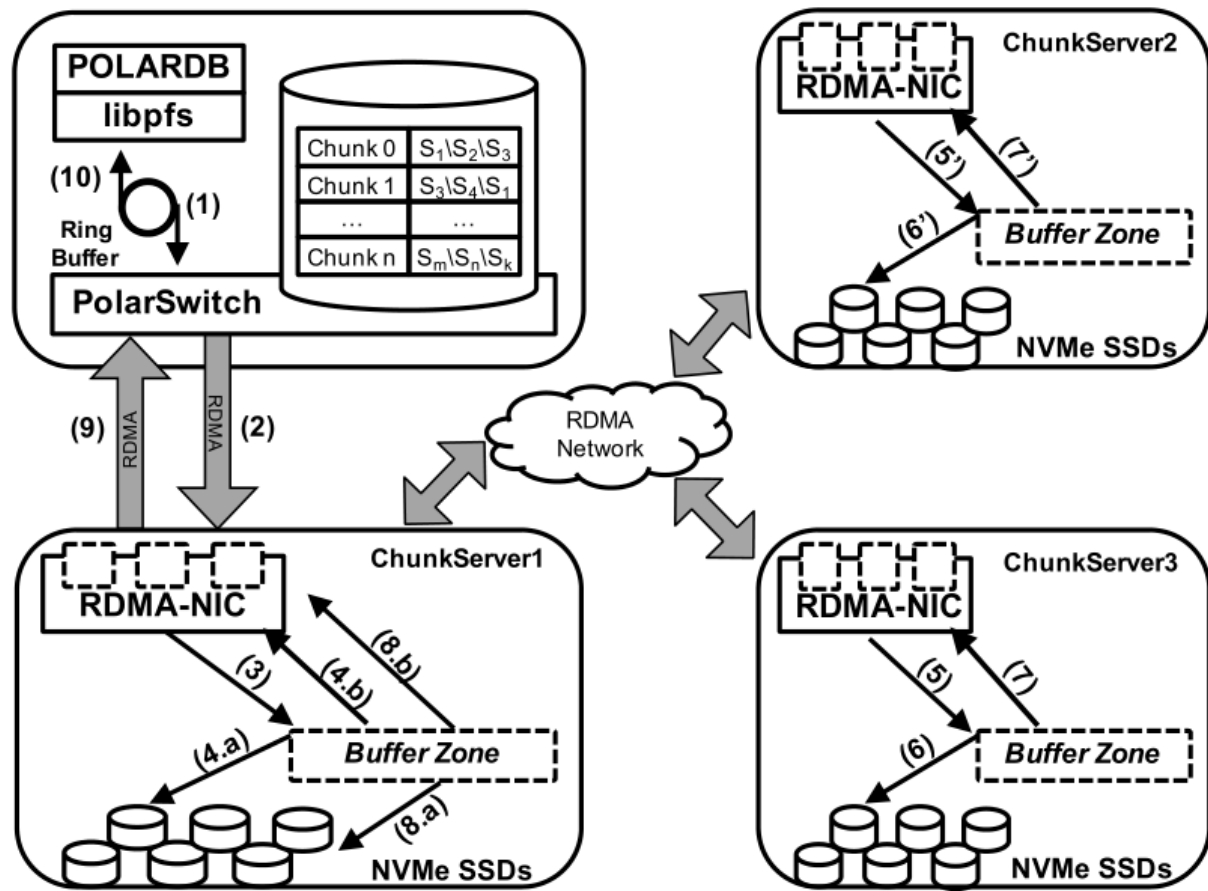


I/O执行模型



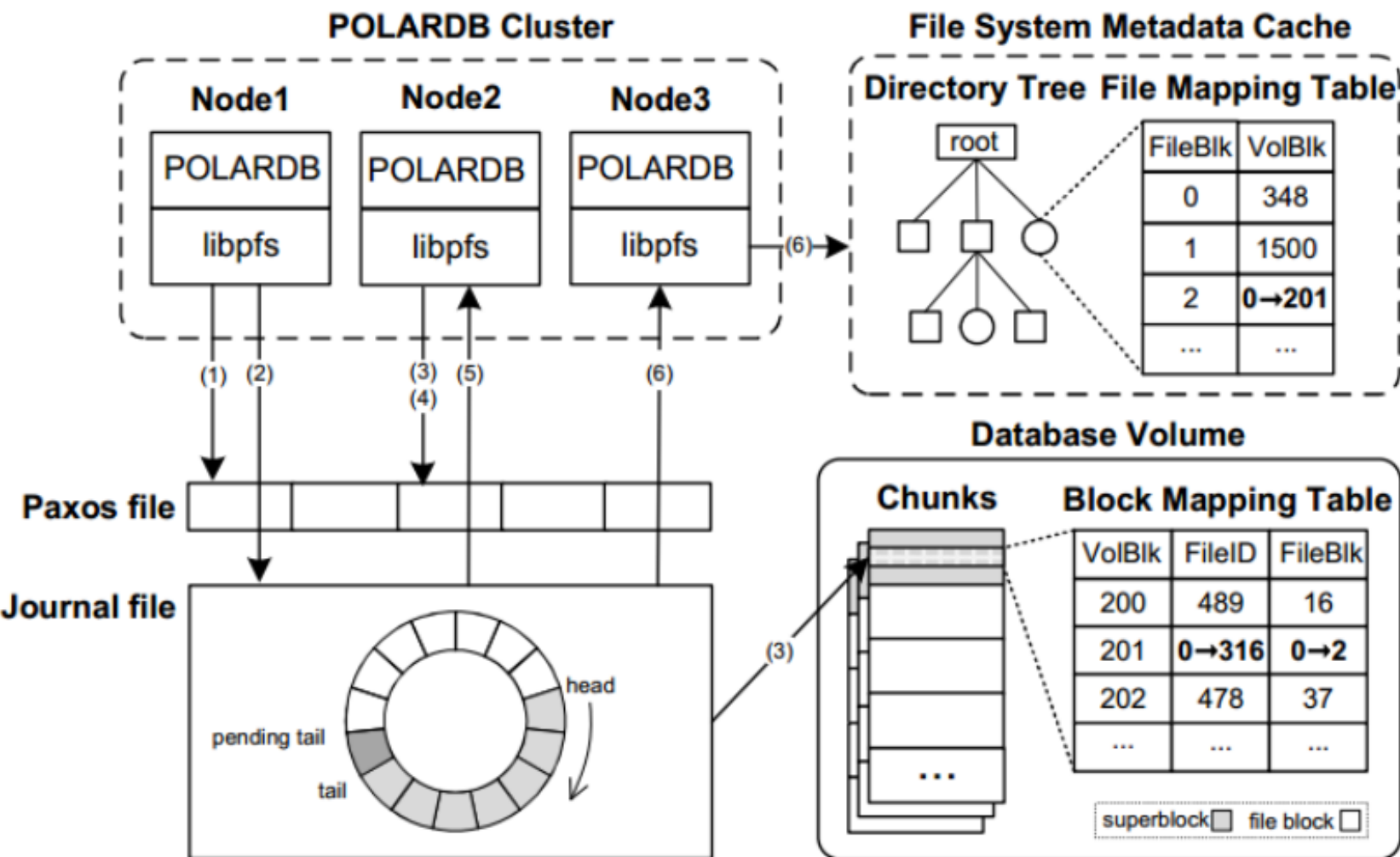
1. 写请求经由ring buffer发送到PolarSwitch
2. polarSwitch将请求传输到相应 chunk 的 Leader 节点
3. 将请求加入到chunkserver请求队列，通过IO轮询线程处理
4. 通过异步调用将请求写入chunk的WAL上，并发给Chunk的Follower节点
5. Follower节点的写请求成功后，通过回调函数向Leader节点发送应答响应
6. leader将数据写到指定的数据块，并向PolarSwitch返回请求处理结果
7. PolarSwitch标记请求成功并通知上层的PolarDB

I/O执行模型



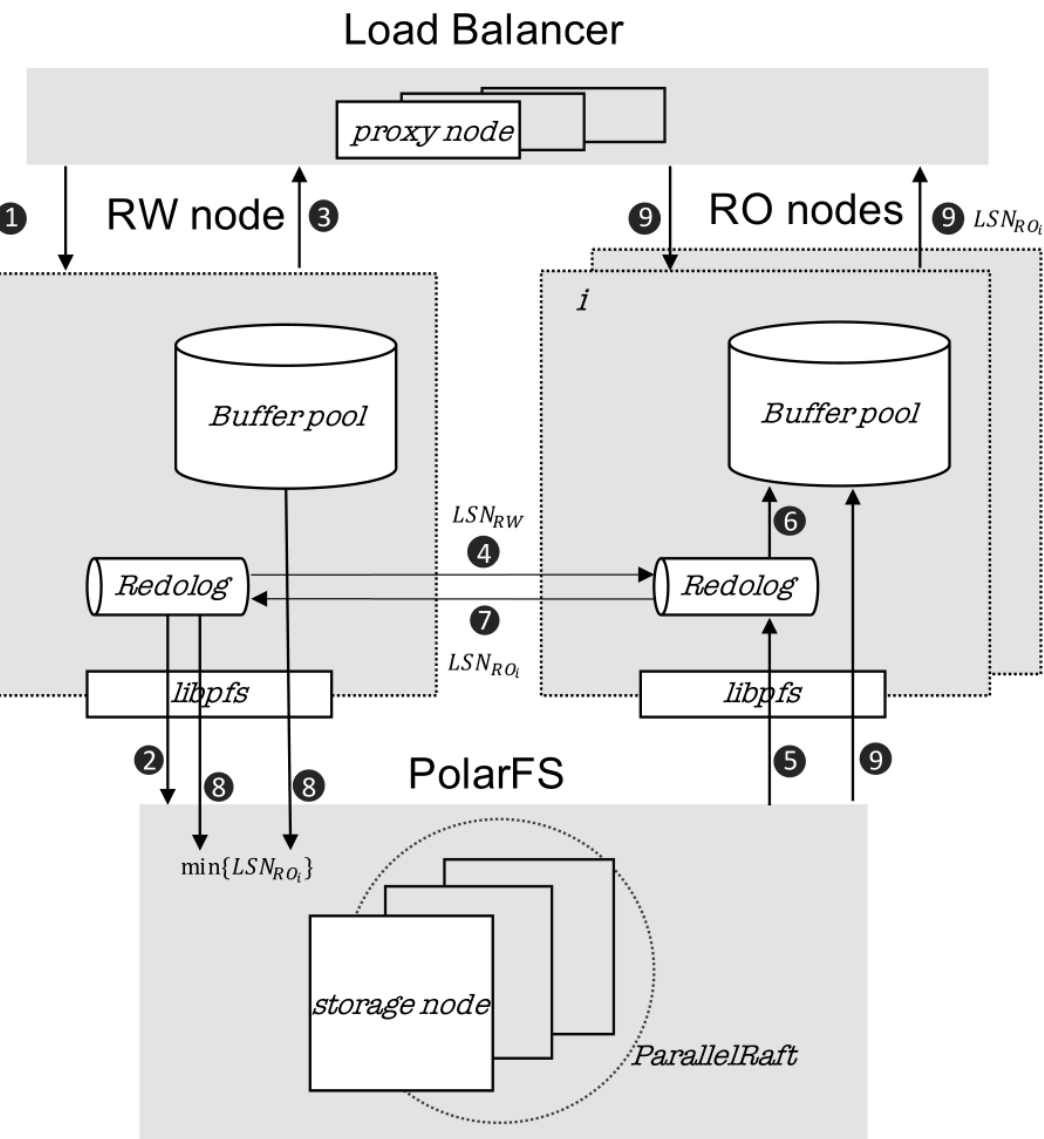
- 1.通过 `pfs_fallocate()` 预先分配block给文件，避免了读写节点之间昂贵的元数据同步
2. ChunkServer 使用WAL来确保原子性和持久性
- 3.大量使用轮询模式和异步调用来提高吞吐率

元数据同步



1. journal文件记录元信息修改
2. disk paxos 实现多个实例对journal 文件的写互斥
3. 通过原子IO无锁获取Journal信息，使得PolarDB可以提供近线性的QPS性能扩展

缓存一致性



1. 缓存在buffer pool上的数据无法被其他节点感知, 会导致缓存不一致
2. 通过redo log来实现内存状态的同步
3. LSN (InnoDB里redo log的偏移量) 来协调内存的一致性



目录

C O N T E N T S

1. 云原生数据库
2. PolarDB
3. **PolarDB Serverless**
4. 总结



03 Serverless



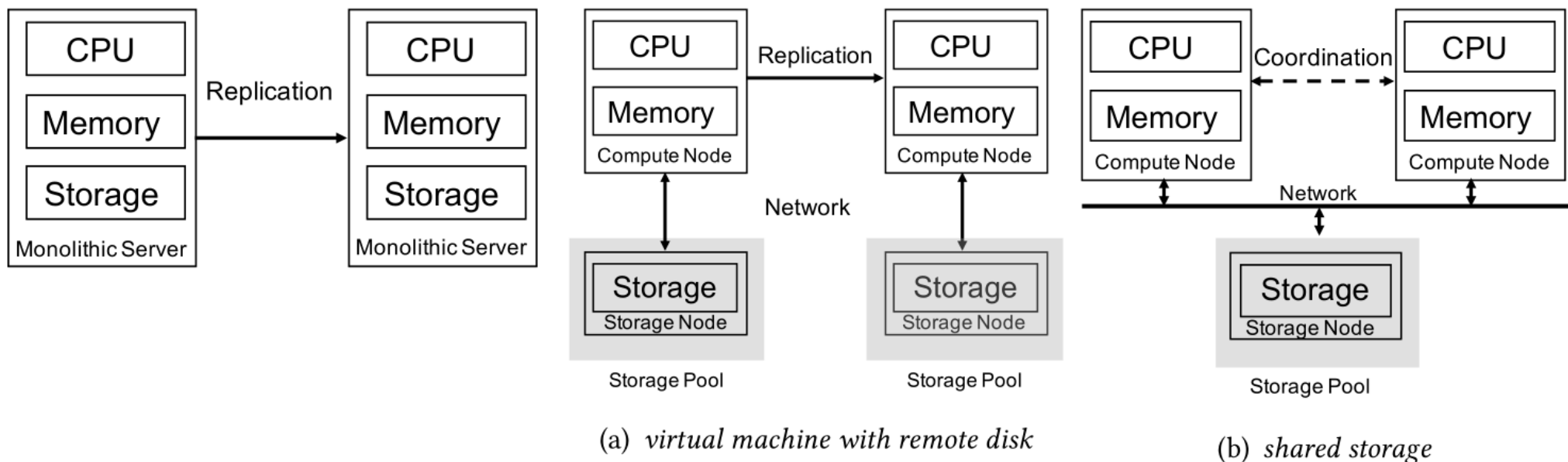
Serverless（无服务器架构）指的是由开发者实现的服务端逻辑运行在无状态的计算容器中，它由事件触发，完全被第三方管理，其业务层面的状态则被开发者使用的数据库和存储资源所记录

Serverless相比serverful，有以下3个改变：

- 1.弱化了存储和计算之间的联系。
- 2.代码的执行不再需要手动分配资源
- 3.Serverless按照服务的使用量（调用次数、时长等）计费

主要特点：**免运维、弹性伸缩、按需付费、高可用**

03 CPU、内存一体



CPU、内存一体缺点：

1. 缺乏灵活和可扩展的内存资源
2. 读取副本可能导致内存数据冗余
3. 不同的场景下CPU和内存资源绑定并不能实现资源的充分利用
4. 缩容：需要同时释放cpu和内存。在重建服务的时候，需要预热内存从而带来了时间开销。

03 CPU、内存分离

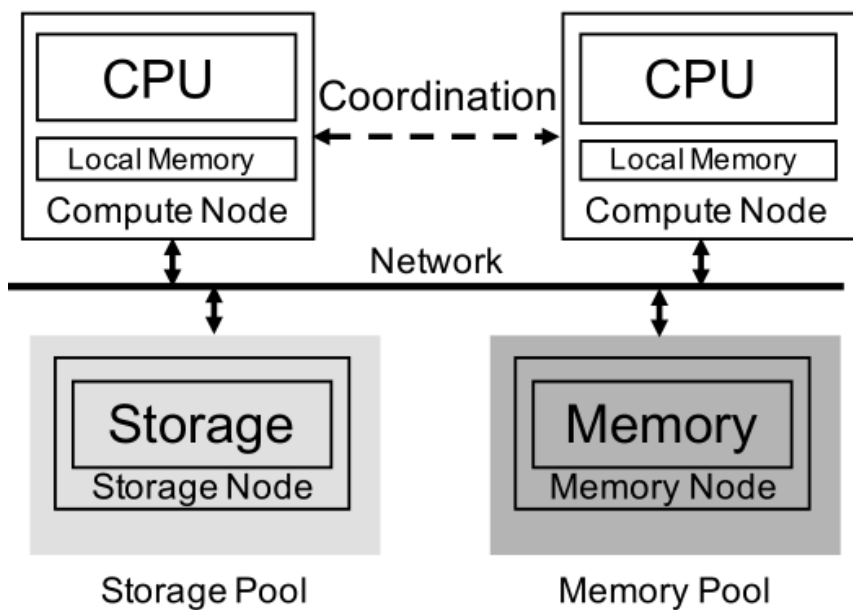


Figure 3: *disaggregation*

CPU、内存分离优点:

- 1.提高资源利用率并且独立可扩展
- 2.可以在多个数据库进程中共享远程内存池中的数据页面
- 3.在重建服务的时候, 无需预热内存降低时间开销

1.远程内存池分配的基本单元slab，大小为1GB

2.Home Node存储的元数据：

Page Address Table (PAT)

Page Invalidation Bitmap (PIB)

Page Reference Directory (PRD)

Page Latch Table (PLT)

3.实例分配内存过程：

a.数据库的进程向home node发送page_register请求

b.如果页面不存在，则home node负责遍历所有现存的slab来找一个空闲区域最大的slab。如果所有slab都没有空闲区域了，会找那些引用计数为0的Page来淘汰掉

c.写入后，把page位置信息记录到PAT，并且返回Page的远程地址和PL

内存池存在的问题：

1.访问远程内存带来网络开销，会比访问本地内存要慢？

➤ 采用缓存以及预测策略

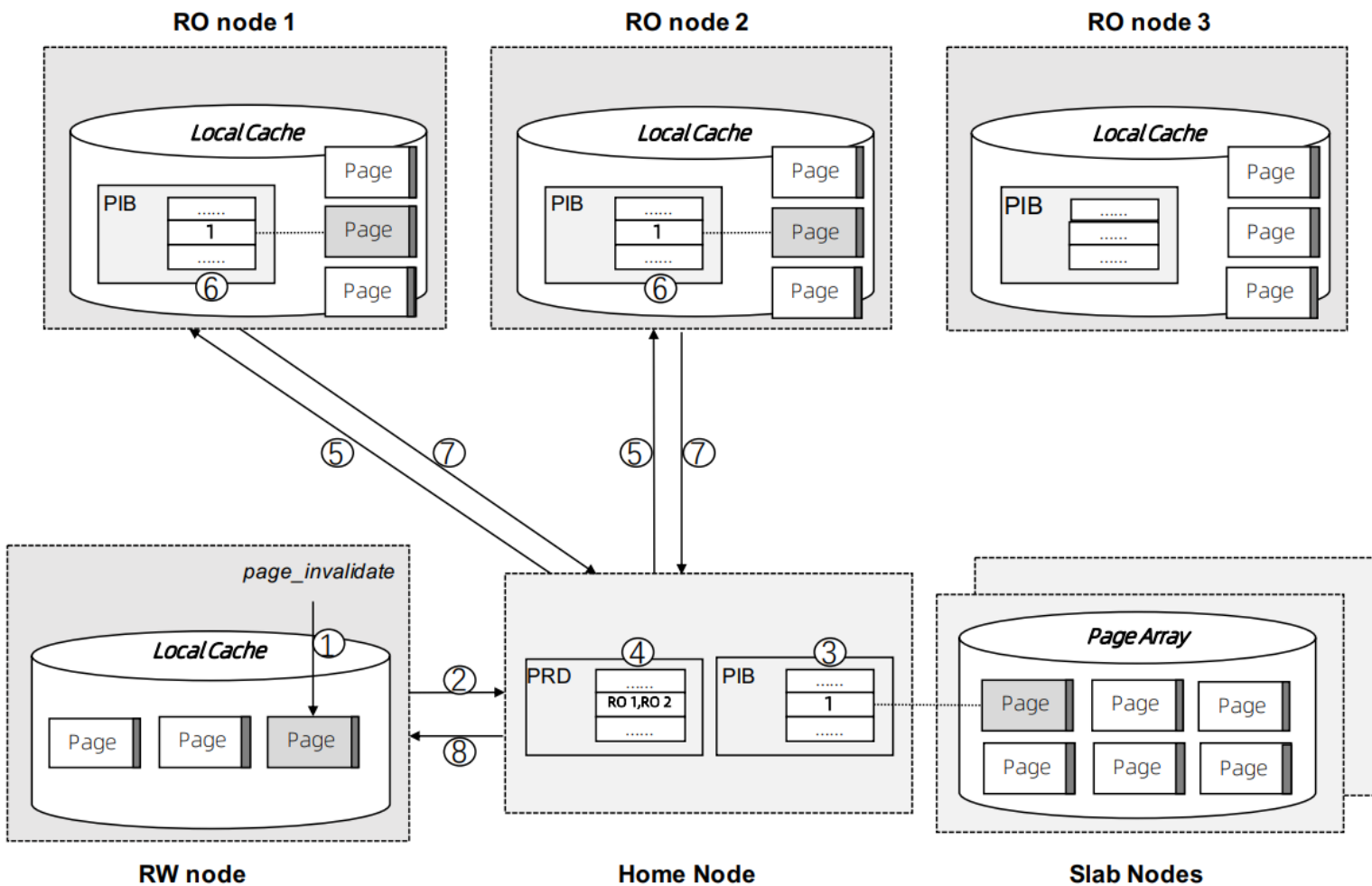
2.一些原来节点上私有的page数据现在都放在了共享的内存池，多节点访问会引起数据一致性问题？

➤ global latch机制、乐观并发控制策略来尽量减少对global latch的使用

3.flush脏页的时候会给网络带宽造成压力？

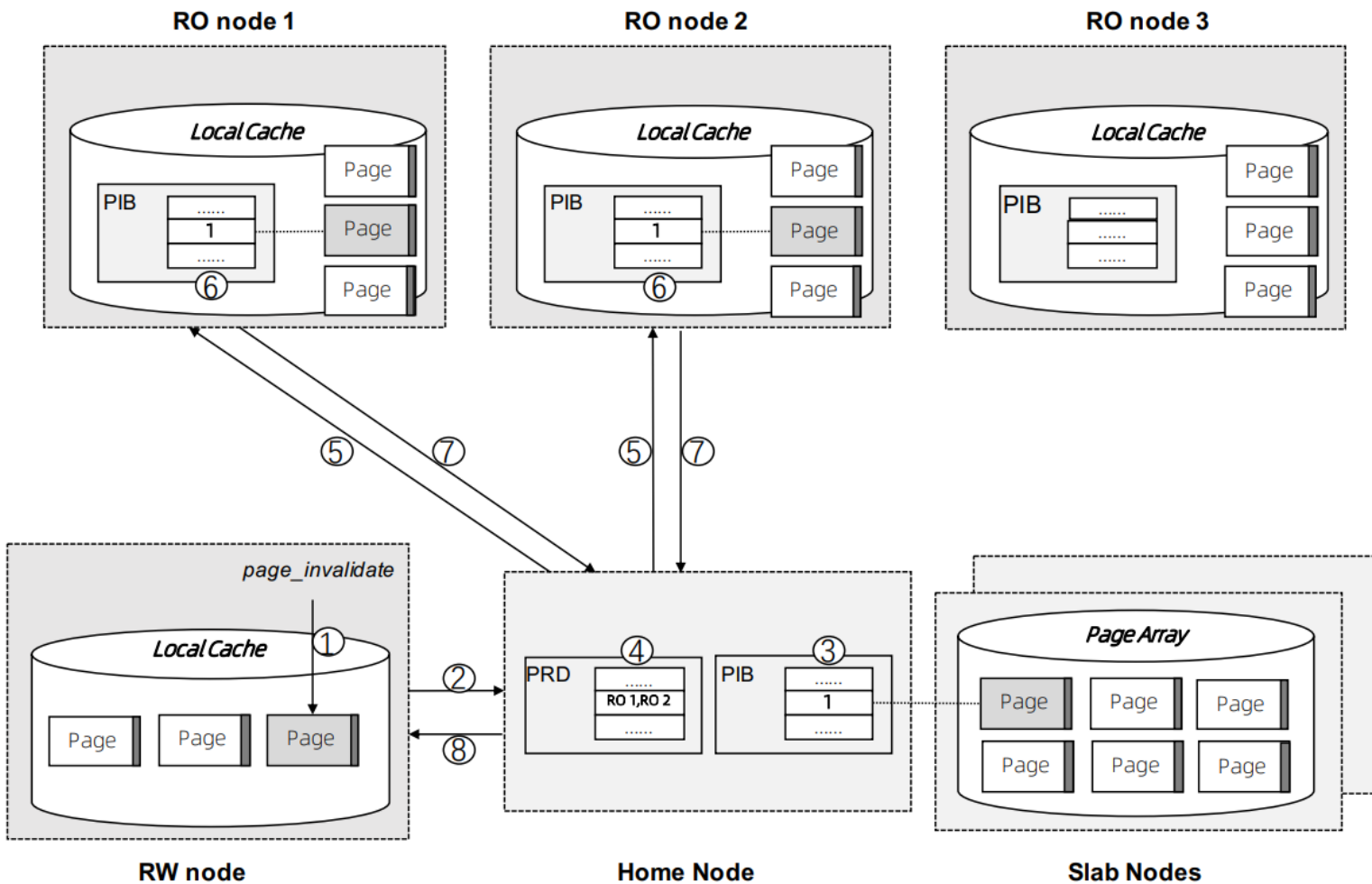
➤ 写redo log到存储层，利用redo log来物化page而避免flush脏页到存储层

本地缓存



1. 访问远程内存需要网络开销，有必要建立本地缓存
2. 修改page后不会马上同步，否则网络传输的开销会很大
3. RW节点修改了本地缓存，需要使RO节点本地缓存对应的page失效

03 缓存一致性



1. 设置RW node上的PIB
2. 调用page_invalidate
3. 设置Home node上的PIB
4. 查找PRD, 获取保存该副本的RO node
5. 设置RO node的PIB
6. 当所有的node都设置成功之后, 返回

03 B+树并发控制



1. Structure Modification Operations会在同一时间修改多个Page，会导致其他RO节点在遍历B+tree的时候会看到修改前后的不一致情况

2. 使用全局锁（global physical latch），两种锁的状态：S和X，被保存在PLT中。

3. 使用**乐观并发控制**，RO探测SMO发生的方法：

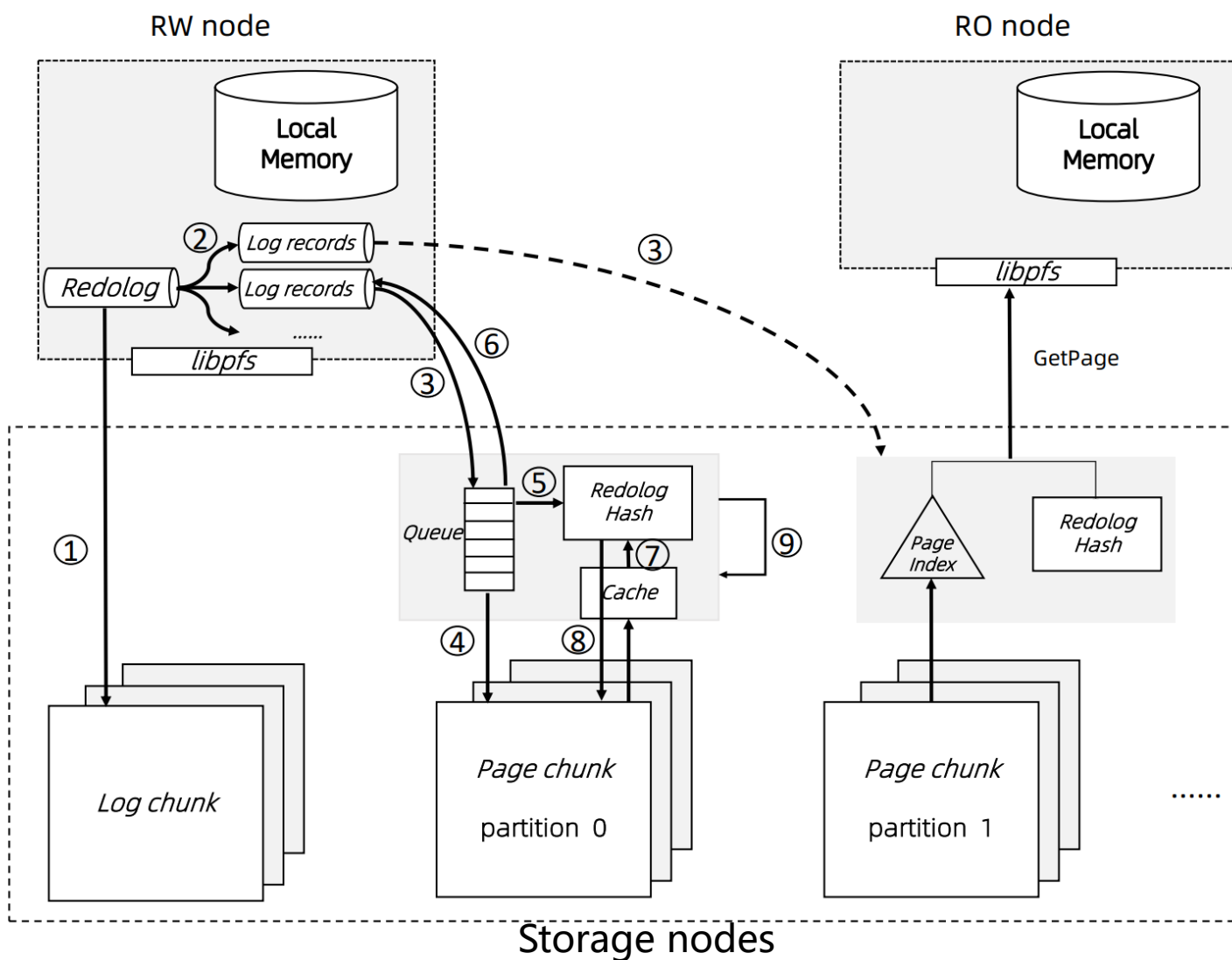
a: RW节点会维护一个counter，每次发生SMO的时候counter++；所有被修改的Page也会维护这个更新后的counter

b: 每次query执行开始的时候去获得这个SMO counter（SMO query）

c: RO上的query发现某个Page的SMO counter比SMO query还要大，说明在query执行过程中发生了SMO

d: 于是回滚到悲观并发控制，即获取全局PL来锁住整个B+ Tree的SMO更新

03 Page Materialization Offloading



- 1.将redo log写入到log chunk
- 2.log chunk中的log发送到page chunk
- 3.page chunk的leader收到log records
4. log records持久化到page chunk中
- 5.更新hash table, 将page id作为key插入到该表中
- 6.当完成后, 响应给RW节点
- 7.从cache或者磁盘中读取该page的老版本, 并与hash table中的log执行merge操作
- 8.将merge之后的数据写入到其他副本
- 9.在后台执行GC操作, 回收老版本



目录

C O N T E N T S

1. 云原生数据库
2. PolarDB
3. PolarDB Serverless
4. 总结

总结 04



- 1.云原生四要素：微服务、DevOps、持续交付、容器化
- 2.云服务分类：IaaS、PaaS、SaaS
- 3.云原生数据库特点：高可用性、安全可靠、动态可扩展、低成本
4. PolarDB使用共享存储PolarFS，主要的优化点是使用了新型的硬件和一些新的技术，如 NVMe SSD、RDMA，SPDK 及 优化了的 Raft 协议 ParallelRaft，尽可能的使用各种方法降低时延，如异步调用和轮询机制
- 5.通过使用disk paxos来确保只有真正的主节点被成功服务，实现元数据一致性
- 6.共享内存会导致缓存不一致，需要通过redo log来实现内存状态的同步

7.Serverless是云原生技术发展的高级阶段，特点：免运维、弹性伸缩、按需付费、高可用


8. CPU、内存分离：提高资源利用率并且独立可扩展；可以在多个数据库进程中共享远程内存池中的数据页面；在重建服务的时候，无需预热内存降低时间开销

9.内存池存在的问题：

- a.访问远程内存带来网络开销，会比访问本地内存要慢

- b.一些原来节点上私有的page数据现在都放在了共享的内存池，多节点访问会引起数据一致性问题

- c.flush脏页的时候会给网络带宽造成压力



谢谢