数据库内核月报 － 2025 / 07

CloudJump II：云数据库在共享存储场景下的优化与实现（发表于SIGMOD 2025）

Author: 王康, 陈宗志

云原生数据库的一个核心理念是计算与存储的解耦（计存分离），这种解耦将数据库系统划分为两个独立的层次：**计算层**（负责查询和事务处理）和**存储层**（管理日志和数据页的持久化），两者可以各自独立扩展。在《CloudJump: Optimizing Cloud Database For Cloud Storage》论文中，我们分析过计存分离架构下，存储层从本地存储转向云存储后，这种介质变化对数据库设计上的挑战，提出一系列的优化框架来应对这些挑战，并发挥计存分离的优势。更进一步，共享存储采用是计存分离之后，自然而然又非常重要的选择，是让多个计算节点共享同一份远端存储，从而给数据库带来了高效的计算弹性、快速原子的节点切换、更低的主从延迟等一系列优势。与计存分离一样，共享存储也为数据库带来了新的挑战和机遇。为此，我们在CloudJump II论文中，详细的分析共享存储数据库涉及中面临的问题，提出MVD（Multi-Version Data）技术来应对挑战，探索并获得更多的共享存储架构优势。

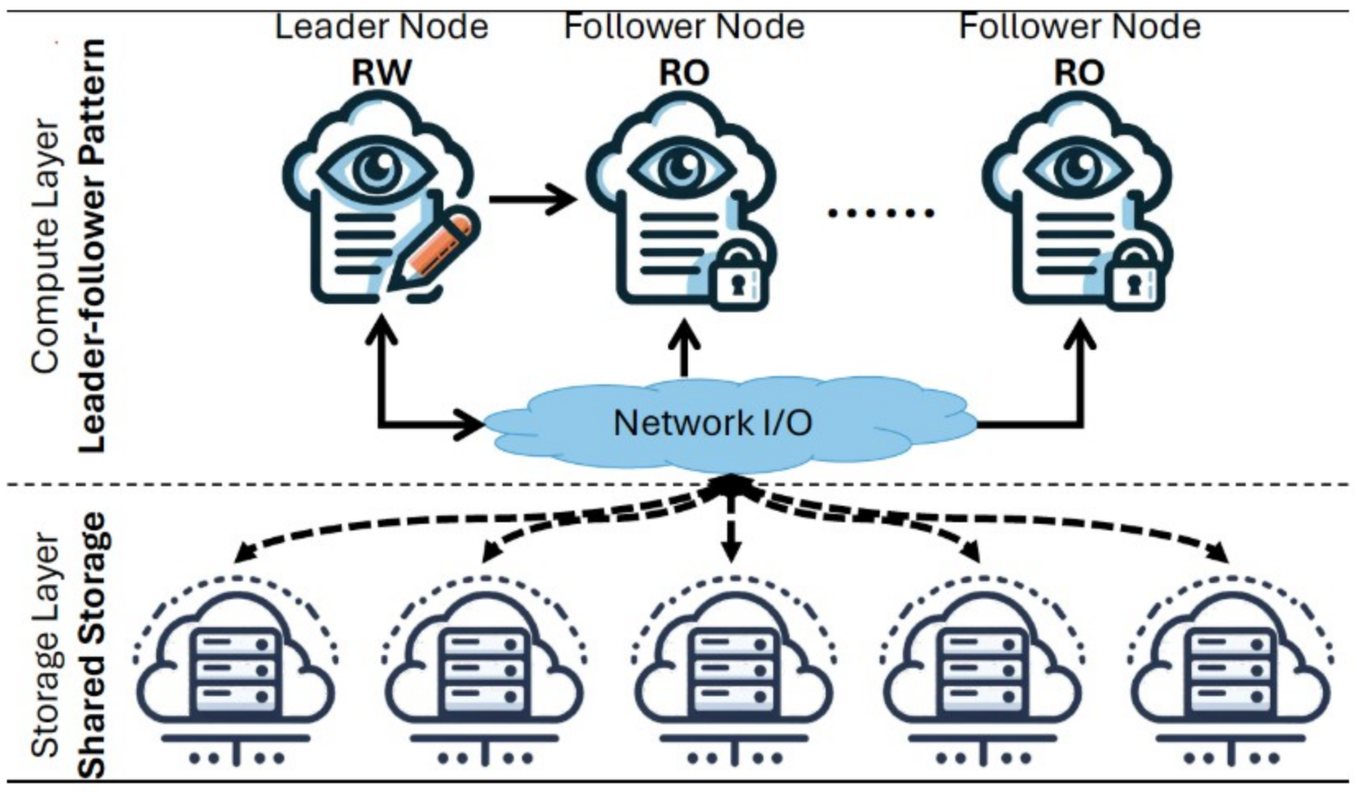
这篇论文已经发布在2025 SIGMOD上，感兴趣的同学可以下载阅读：
《CloudJump II: Optimizing Cloud Databases for Shared Storage》。

共享存储

云数据库将计算层和存储层分离解耦之后，让多个计算节点共同访问同一份云存储便是一种很自然的选择， 如图所示，多个计算节点，包括1个可以读写的Leader（RW）节点和多个只读的Follower（RO）节点，通过网络连接到一组由很多存储节点组成的存储集群上。这种架构相对于传统的主从结构数据库，或者以 Google Spanner 为代表的“无共享”（shared-nothing）数据库，最大的特点就是完全消除了节点间的数据拷贝，进而可以获得一些明显的优势，例如：

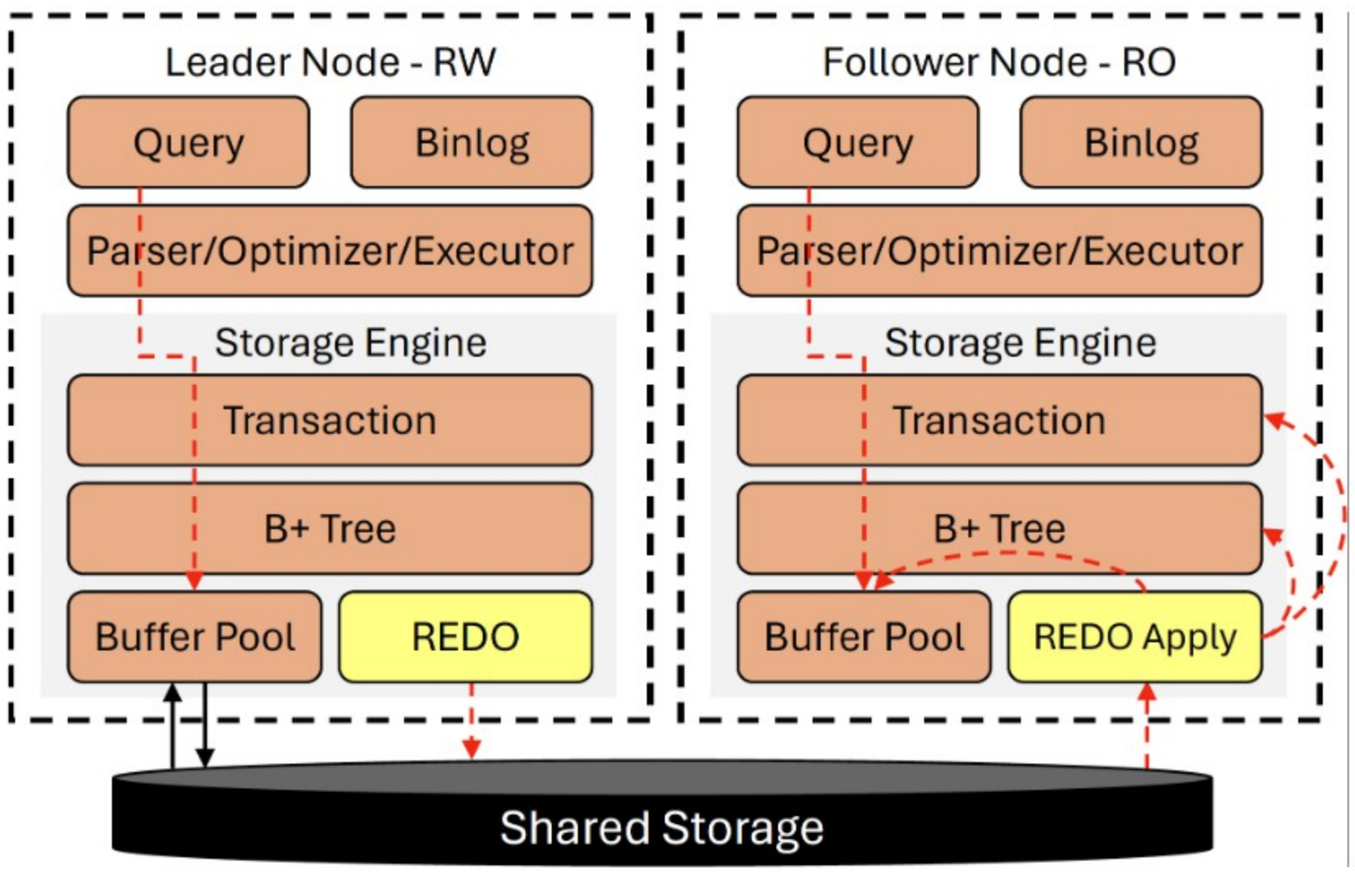
- 正常运行过程中，Follower节点并不需要从Leader节点拉取日志，并完整的重放Leader上的操作，取而代之的只是一些包括写入日志位点在内的元信息同步，以及一些内存状态的维护更新。因此可以用很低的网络IO开销实现**非常低（ms级）的主从时延**；
- 增加计算节点的过程中，并不需要数据的拷贝，而是直接对接同一份存储数据，做必要的内存状态初始化后就提供服务，从而获得数据量无关的**快速弹性能力**；

不同的计算节点虽然可以共享同一份存储数据，但不同的节点的内存状态仍然是独立的，最直接的就是各个节点维护在内存中的页缓存Buffer Pool，只读节点仍然需要通过日志重放来获得读写节点上，包括页缓存在内的最新修改，并以此来更新自己的内存状态。但不同于传统的主备结构，共享存储数据库的不同节点需要一个完全一致的物理数据视角。这一点是传统基于类似Binlog这种逻辑日志的复制方式不满足的，取而代之的，是需要基于物理日志的复制方式，比如MySQL下的Redo日志，这种复制方式我们称为**物理复制**。



Amazon Aurora作为最早的满足我们上面讨论的，这种采用物理复制的共享存储工业级数据库产品，基于极致减少网络IO开销的考虑，进一步将存储层定制为专用的Page服务，向计算节点提供写Redo和读指定版本Page的服务，而从计算层到存储层完全没有Page的写入流量，由存储层独立重放Redo日志来推进Page版本，由于通常Redo的修改量会远远小于Page页的大小，因此这种方式能极大的降低网络写入流量。Aurora这种方式深刻的影响了后来的各种云原生共享存储数据库产品，包括Microsoft的Socrates和各大云厂商的类似产品。

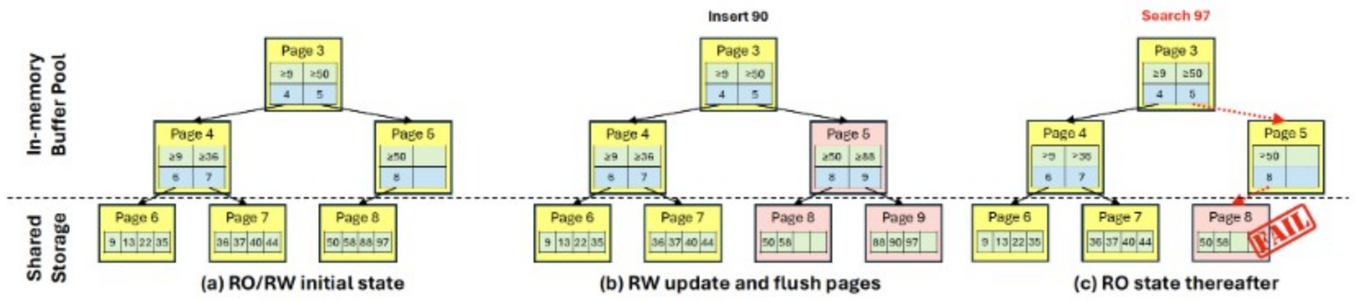
然而，这种方法由于深度定制了存储层的服务，将更多的数据库复杂度放到存储层实现，限制了存储层利用各大云厂商持续优化的标准云存储服务的能力，增加了存储层实现的复杂度，扩大了故障半径。为此，我们提出一种不同的探索：基于 **CloudJump** 框架，利用标准云存储服务来构建云数据库的共享存储层。该方案增强了存储解决方案的灵活性与可扩展性，更好地满足云原生应用的动态需求。借助标准化组件，CloudJump 能够在多种云平台上构建高质量的云原生数据库服务，更好的支持存储层的独立演进，更好享受由于新硬件、新架构带来的存储层升级红利。



以阿里云 **PolarDB 在共享存储架构**中的实现为例，我们可以更清晰地理解这一机制。PolarDB 支持基于单一共享数据集的Leader-Follower模型，包括一个读写节点（RW）和多个只读节点（RO）。共享存储对计算节点提供的，是支持标准文件系统接口的分布式存储服务。在执行写操作时，RW 节点会生成Redo日志文件，并写入共享存储，每条日志都通过一个日志序列号（LSN）标识，对应数据库的一个特定版本。同时会通过网络通知所有的RO节点最新Redo日志的LSN位置。RO 节点则需要重放这些日志以同步最新的更新，包括 Buffer Pool 中的数据页、事务状态以及各种内存缓存结构，这种同步机制称为**主动日志追加机制**（Active Log Update Chasing）。除此之外，对于之前不在内存Buffer Pool的Page，会通过**被动按需访问机制**（Passive On-Demand Access）在第一次被访问时从共享存储中加载，并通过重放Redo日志恢复到RO需要的最新状态。

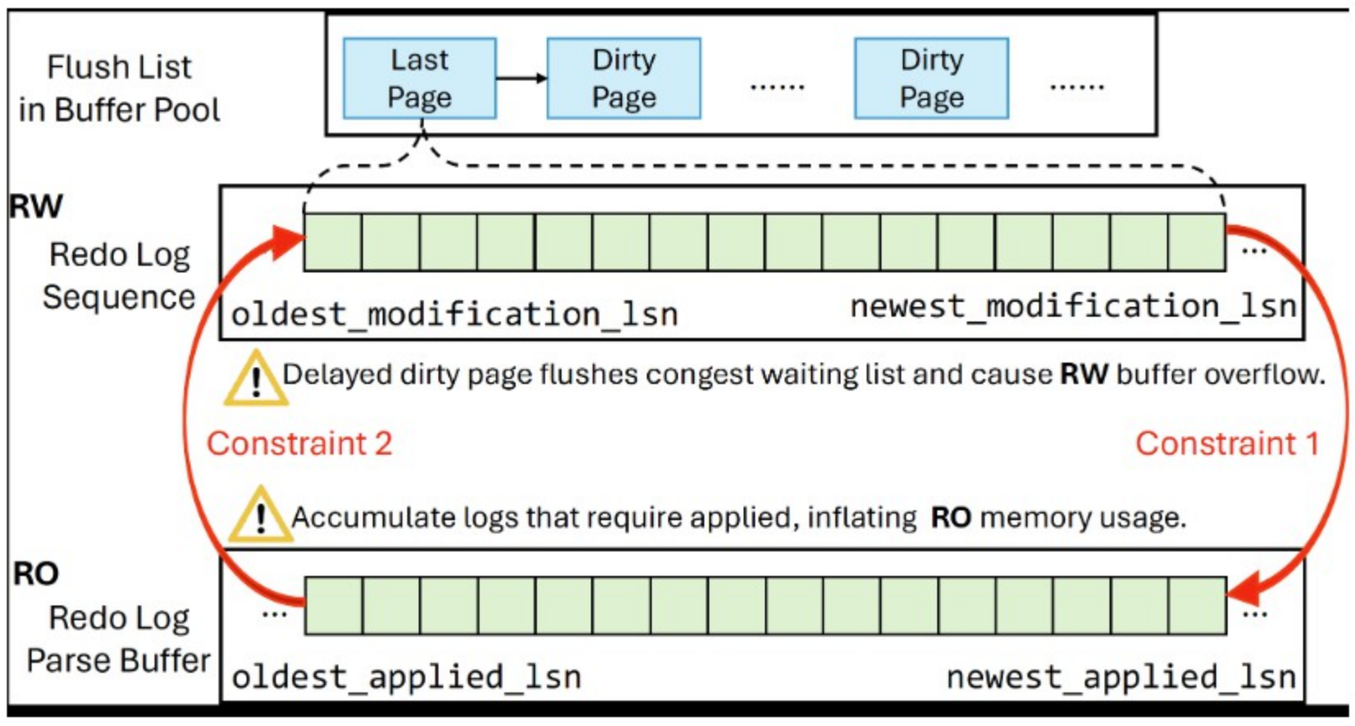
挑战与分析

前面介绍过，在这种采用Leader-Follower模型的共享存储数据库中，RO节点的内存数据更新依赖于异步重放Redo日志，而共享存储中是的数据是通过RW上的Page刷脏来更新的，那么就会存在RO上数据的一致性问题，以如下一个B+Tree节点分裂的场景为例：



如图（a）中是RW和RO内存中的一个Buffer Pool中Page的初始状态。这时，Insert 90导致Page 8发生了分裂，产生了新的Page 9拥有了部分之前在Page 8上的元素，97就是其中之一，如图（b）所示，然后RW对Page 8进行了刷脏，使得在共享存储上的Page 8被最新版本覆盖；由于异步复制，RO上Buffer Pool其实还停留在发生分裂之前的状态，这时一个请求发起了对97元素的查找，自然的RO上会通过B+Tree定位到Page 8，由于Page 8未在内存，RO发起了从共享存储的Page读，看到了Page 8的分裂后状态， 上面并没有元素97。注意这种情况是完全错误的，并不是一个正常的最终一致的状态。我们分析这种情况，由于异步复制RO的延迟是被允许的，但RO应该看到的是一个完整的、自己所在的历史状态，也就是分裂发生前，拥有元素97的Page 8。

那么解决这个问题的关键就是要让***RO可以获得自己需要的、一致的、可能落后于RW的数据版本**。对于在RO的BufferPool中缓存的Page，通过主动日志追加机制，随着RO自己的位点推进应用Redo，可以容易的将其维护在一个正确的位置。麻烦的部分在于，之前不在RO内存中的，如上图Page 8这样的Page。为了实现这一点，就必须保证在RO访问这样的一个Page的时候，1）必须可以从共享存储中，获得一个比当前RO的LSN位点更老的Page版本，并且2）可以获得所有这个Page老版本之后，针对当前Page的Redo日志，并通过应用获得所需要的版本。为此一种可行的方案，是对计算节点增加如下约束条件：



约束条件 1：限制 RW 刷脏页行为（Restricting RW flush dirty pages）

在RW节点的Buffer Pool中，被修改过的Page（脏页）会维护其当前最早修改的起始LSN（oldest_modification_lsn）以及最后修改的结束LSN（newest_modification_lsn）。当 RW 将脏页写入到共享存储时，必须确保该页的 `newest_modification_lsn` 不超过任何 RO 节点当前已应用的日志LSN（newest_applied_lsn）。这样做的目的是防止任何 RO 节点获取到一个“未来的”，过于超前的数据页。通过这个保证，上面例子中分裂后的Page 8就不会被RW写入到共享内存而被RO看到。

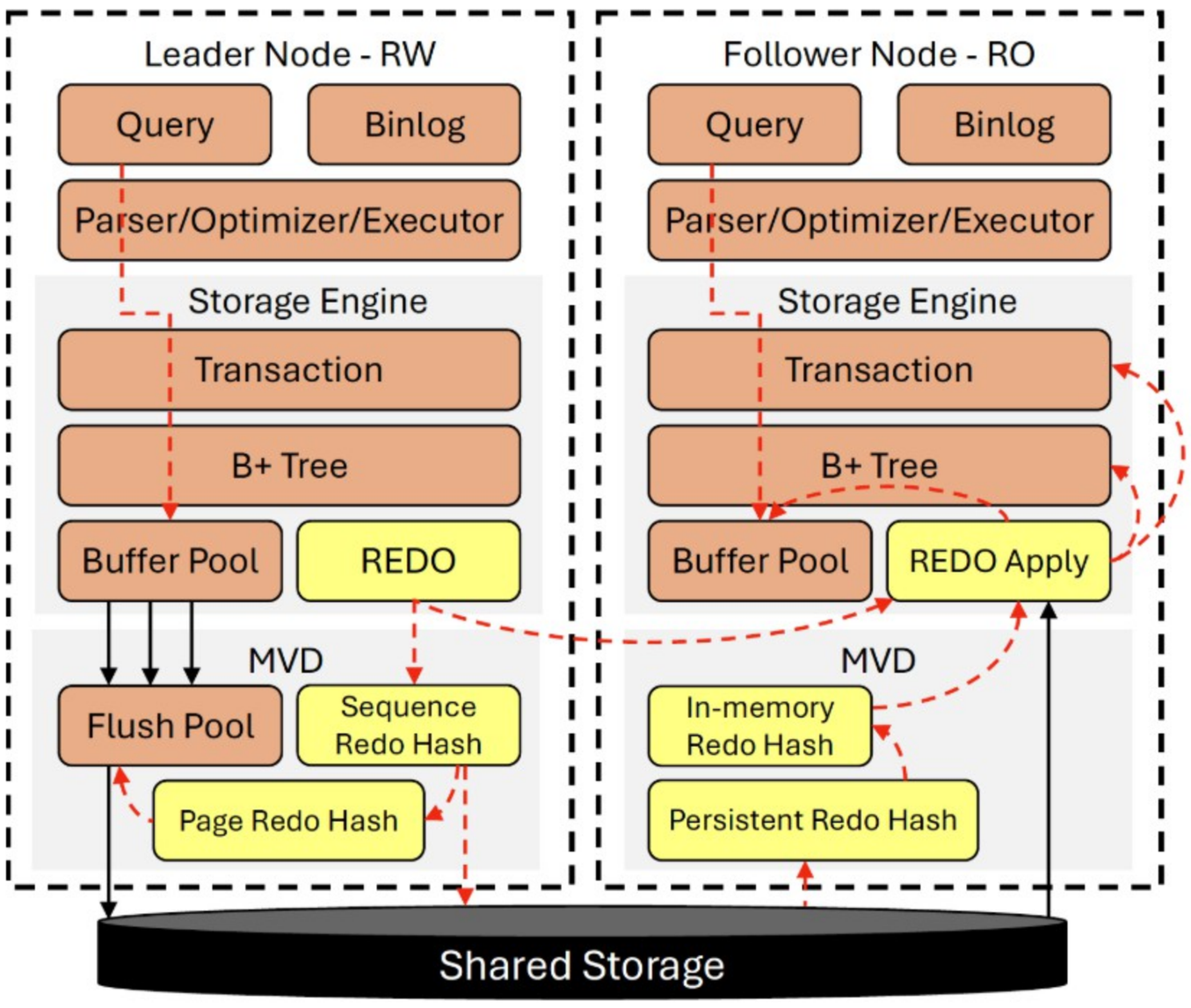
约束条件 2：增强 RO 内存处理能力（Augmenting RO memory）

当 RO 读取一个数据页时，它必须处理其日志解析缓冲区（log parse buffer）中与该页相关的所有Redo日志，并将这些修改应用到从共享存储中获取的数据页上，以确保数据页更新为最新的状态。而为了保证日志足够，RO就需要维护所有RW当前缓存池中的Page的最小\oldest_modification_lsn之后所有Redo日志。

然而，这种通过双重约束来讲解决共享存储读取数据页一致性的问题的方案，并不理想，因为他带来了性能和灵活性方面的缺陷，首先，由于RW刷脏会受最慢的RO复制延迟的限制，可能导致RW缓冲区中大量的Page不能及时刷脏，而造成其缓冲区效率下降，影响正常的读写请求性能；其次，对于频繁修改的Page，由于其newest_modificateion_lsn一直维持在高位，因此很难被刷脏，然后这样的Page就容易持有很宽的[oldest_modificateion_lsn, newest_modificateion_lsn]范围，进而导致RO上需要维护更多的Redo在日志解析缓冲区中，造成RO内存上涨甚至溢出。

CLOUDJUMP II—THE MVD APPORACH

上述问题的根本原因在于**共享存储上的页修改会强制完整的数据覆盖**，因为在云存储或POSIX协议下多版本控制并不普遍。CloudJump通过集成MVD（多版本数据）来解决这一问题，使得计算节点内可以进行多个有效版本的读写操作，从而克服了单一版本覆盖的限制。



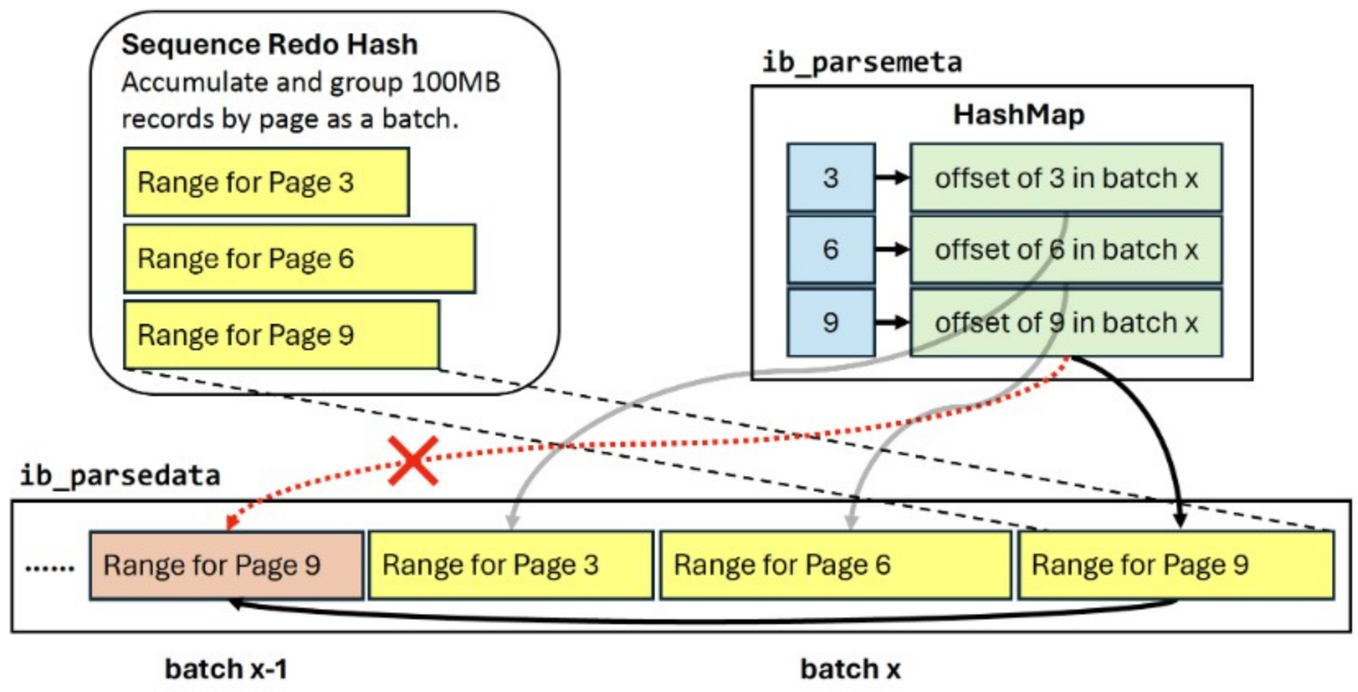
Cloudjump在计算节点内的存储引擎与存储层之间集成了MVD模块，在Leader节点（RW）中，Redo在生成的过程中会同时被整理成按Page索引的Redo Hash，如图**Sequence Redo Hash**，并按照脏页需求整合在**Page Redo Hash**中；**Flush Pool**缓存脏页的回写，不同于Buffer Pool，这里并不维护整个的Page内容，而是只维护这个Page在**Page Redo Hash**中的增量Redo，Page在刷脏时会根据修改量等条件的判断是否要进入Flush Pool中缓存。而在Follwer节点（RO）上，会在主动日志追加过程中，随着自己位点的推进，维护最新的一段**In_memory Redo Hash**，以及一个**Persistent Redo Hash**用于按照Page索引需要的Redo日志，从而支持RO节点通过应用Redo获得自己需要的精确的页面版本。

MVD引擎提供了从垃圾回收（GC）版本到最新版本范围内任意LSN访问页面的能力。这一大特性支持了拥有多个数据库节点的架构，满足了对页面版本的不同需求，同时也支持基于这种能力的共享存储数据库，取得我们后面会提到的，相对于传统数据库的更多优势。而这种以日志为中心的设计，是建立在Redo日志保证完备性和局部性的的基础之上的：

- **完备性（Completeness）**：Redo日志中包含数据库修改的全部信息；
- **局部性（Locality）**：每个Redo日志仅涉及单个页面，因此使用过程可以只关注单个页面，提高效率和准确性。

MVD中存在一个关键过程：当请求某个数据页时，需要检索该页缺失的所有Redo日志。因此，有必要设计一种按页面对Redo日志进行分类的机制，称为**日志索引（Log Index）**。在数据库的运行过程中维护日志索引是不容易的，主要的挑战包括：1）Redo日志本身的生成是极致优化的，在现代数据库中，也会通过多线程、无锁以及分片等技术显著提升了写入效率；2）对同一个Page的修改内容在Redo日志文件中分布又会比较分散，这会导致维护日志索引带来性能下降和元数据体积膨胀的问题；3）Redo文件是顺序不断产生的，我们无法获知未来的修改会涉及哪些Page，因此维持一个全局的日志索引是不现实的；4）数据库整体的资源是有限且重要的。那么如何在不显著增加CPU和IO资源的前提下，使日志索引的生成速率

与快速产生的Redo日志保持同步呢?



为了解决这些问题，我们采用了一种**异步分段排序（Batch）的日志索引生成方法**。该方法保留了标准的Redo日志写入流程，并利用Redo Buffer临时保存最新的Redo日志段。随后，一个异步解析线程读取这些日志、进行解析并生成按Page的日志索引段（Range），一个Batch内所有的Page的这个日志索引段（Range）就是上文提到的Sequence Redo Hash。当日志索引总量积累到一定数量后（Batch），再将其批量刷新到持久化存储中。权衡Sequence Redo Hash的内存占用、日志索引落盘的IO开销，以及一个Batch内Page的聚集程度，一个实践的值是将一个Batch设置为比如100MB。日志索引持久化的时候会用Append Only的方式写入ib_parsedata，并更新一个，记录Page及对应日志Range Offset的内存头信息，这个头信息会周期性的覆盖写入ib_parsemeta文件。由于RO上在主动位点推进过程中维护了In-memory Redo Hash，只有超过这个范围的Redo才需要从Persistent Redo Hash中获取，因此，这里允许日志索引创建有500MB到1GB的延迟，这样给了Log Index生成过程中很大的IO合并的空间。经过测试，这种日志索引的生成方式的造成的开销很小，包括解析并维护内存Redo Hash的约3%到5%的CPU开销，缓存Sequence Redo Hash Batch的约100MB内存开销，以及Append Only写Logindex的IO开销。

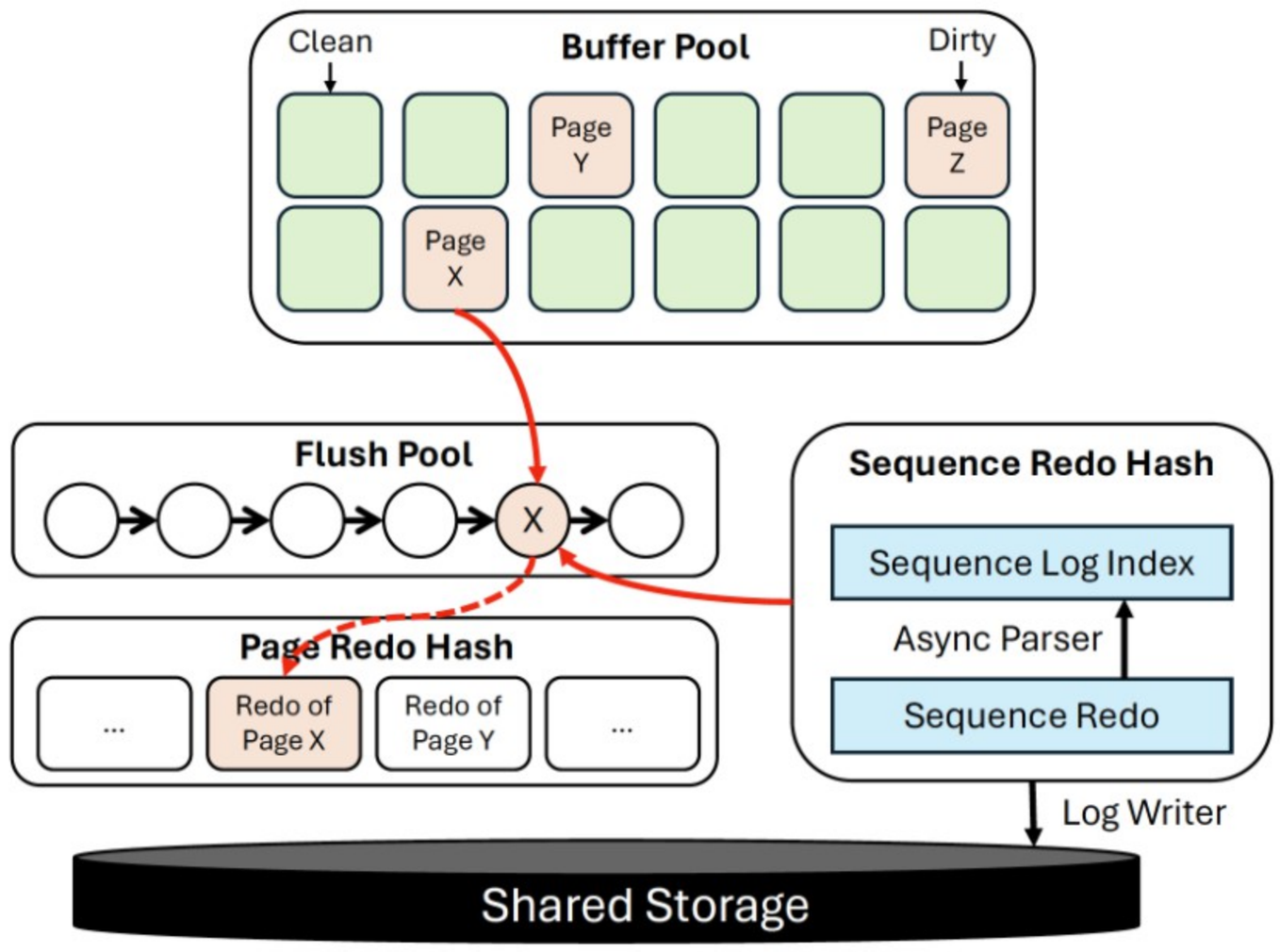
在运行过程中，RO节点持续从共享存储中读取Redo日志，对其进行解析并更新Buffer Pool中的现有Page及各种内存状态。在此过程中，一个基于Page组织的内存日志索引（即图中In-memory Redo Hash）也会同步生成。如果用户请求访问一个新Page，并且共享存储中该Page的LSN已过期，则不仅需要In-memory Redo Hash，还需要通过Persistent Redo Hash，对应ib_parsedate和ib_parsemeta文件中维护的索引加载日志。通过按Page排列Redo记录，可以有条不紊地将这些记录应用于Page，使其恢复到目标版本。因此，日志索引的实现使RO免受内存扩展问题的影响，有效解决了**约束2**，并通过维护最优的Apply LSN间接解决了**约束1**。

DB能力增强

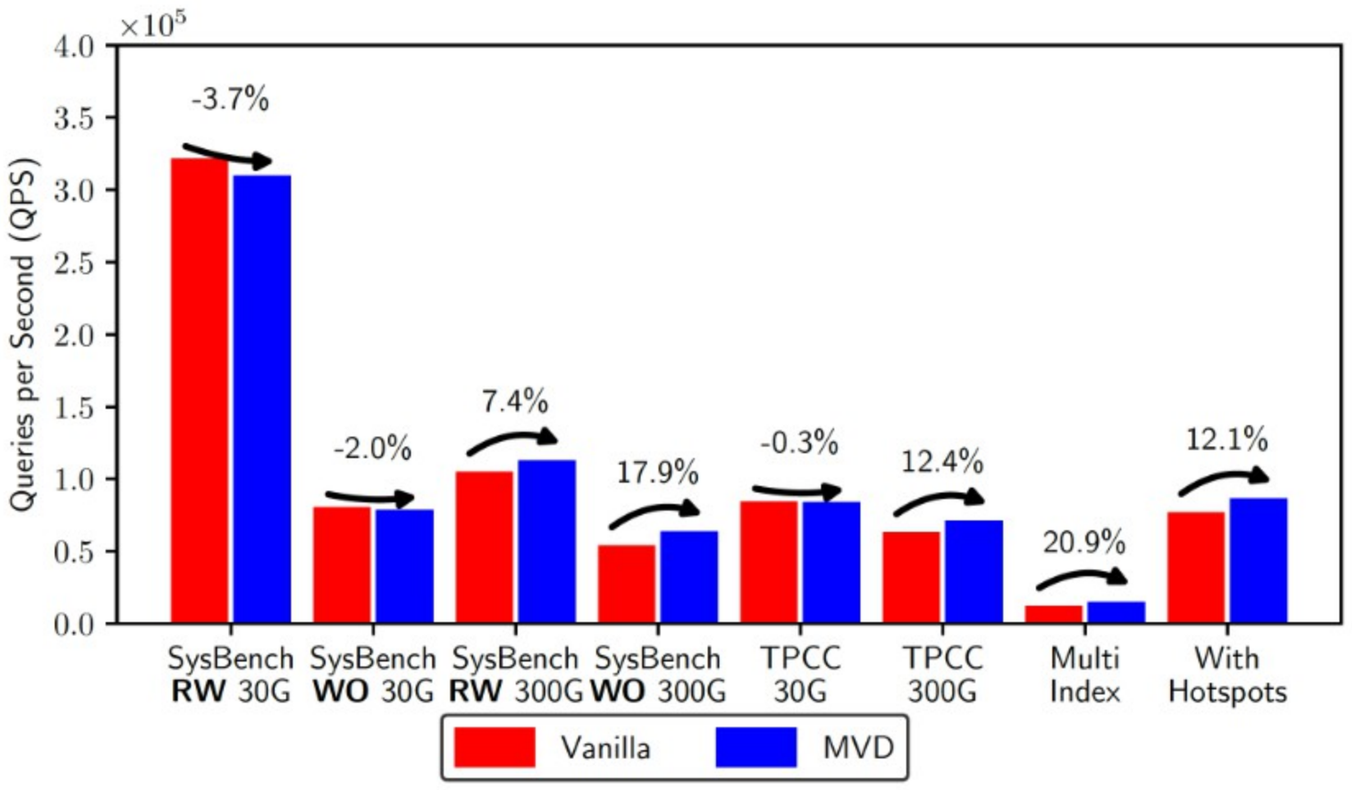
MVD引擎通过对日志索引包括内存和持久化的维护，让DB拥有了：**在任何时候，通过较老的Page版本及之后的Redo日志，在线获得Page的任意版本**的能力。进而不仅解决了共享存储架构下的主从一致性的问题，也让更多的DB能力得以进化。

1. Write Elision（写省略）

在基于WAL的数据库引擎架构中，对数据页的修改通常会生成简短的Redo记录，并使该页在Buffer Pool（缓冲池）中标记为“脏页”。由于Buffer Pool容量有限，当可用空间不足时，就需要通过诸如最近最少使用（Least Recently Used, LRU）等策略来选择被驱逐的页面。如果被驱逐的页面是脏页，则必须先将其写入共享存储，从而触发一次页面大小的写IO操作。在数据量远大于Buffer Pool容量的场景下，这种事件变得非常频繁：一个页面一旦被加载进Buffer Pool并经过轻微修改，就可能很快被驱逐并写入磁盘。这一系列微小修改引发大量写IO操作的现象，不仅造成IO资源的浪费，还可能成为数据库性能的瓶颈，即典型的**IO-Bound**情况。



MVD 所引入的 **Write Elision**（写省略）机制，提供了一种全新的解决方案——在页面被驱逐时**跳过脏页写回过程**，从而避免页面级别的IO操作。随后对该页面的访问通过日志索引获取必要的Redo日志进行变更应用。如上图所示，根据LRU等策略选定待从Buffer Pool刷新的Page后，进入一个多版本写省略策略的选择流程。该流程综合评估多种因素，包括当前用户负载、脏页修改程度以及内存使用情况。被选中进行写省略的Page，其对应的Redo日志将通过ID，从Sequence Redo Hash中提取日志索引以及对应的Redo日志，并整理到Page Redo Hash中，然后被纳入Flush Pool管理，从而跳过当前的刷盘周期。未被选中的Page则按照传统方式刷盘。在后续对该Page的访问完成IO操作后，将从Flush Pool中提取对应的Page Redo Hash，并应用相关的Redo日志以重建完整页面内容。Flush Pool中的Page将在满足刷新条件后，由之后的脏页写回机制，或者由写省略后台线程定期检查，被写入持久化存储，之后从Page Redo Hash中移除。



写省略的核心假设是：通过在Flush Pool中**聚合同一页面的多个IO请求**，可以提升整体效率，同时通过对Flush Pool之外的缓存页面进行管理，防止缓存过度占用。同时有了写省略机制之后，脏页写回的时机有了更多的选择，也缓解了上面讲的节点之间的刷脏约束1。如上图所示的实验结果中，可以看出越是数据量相对于Buffer Pool大的场景（如300GB VS 30GB），越是单次修改的Redo量相对于Page Size小的场景（Multi-Index VS Sysbench），这种提升越明显。

2，Instant Recovery（快速恢复）

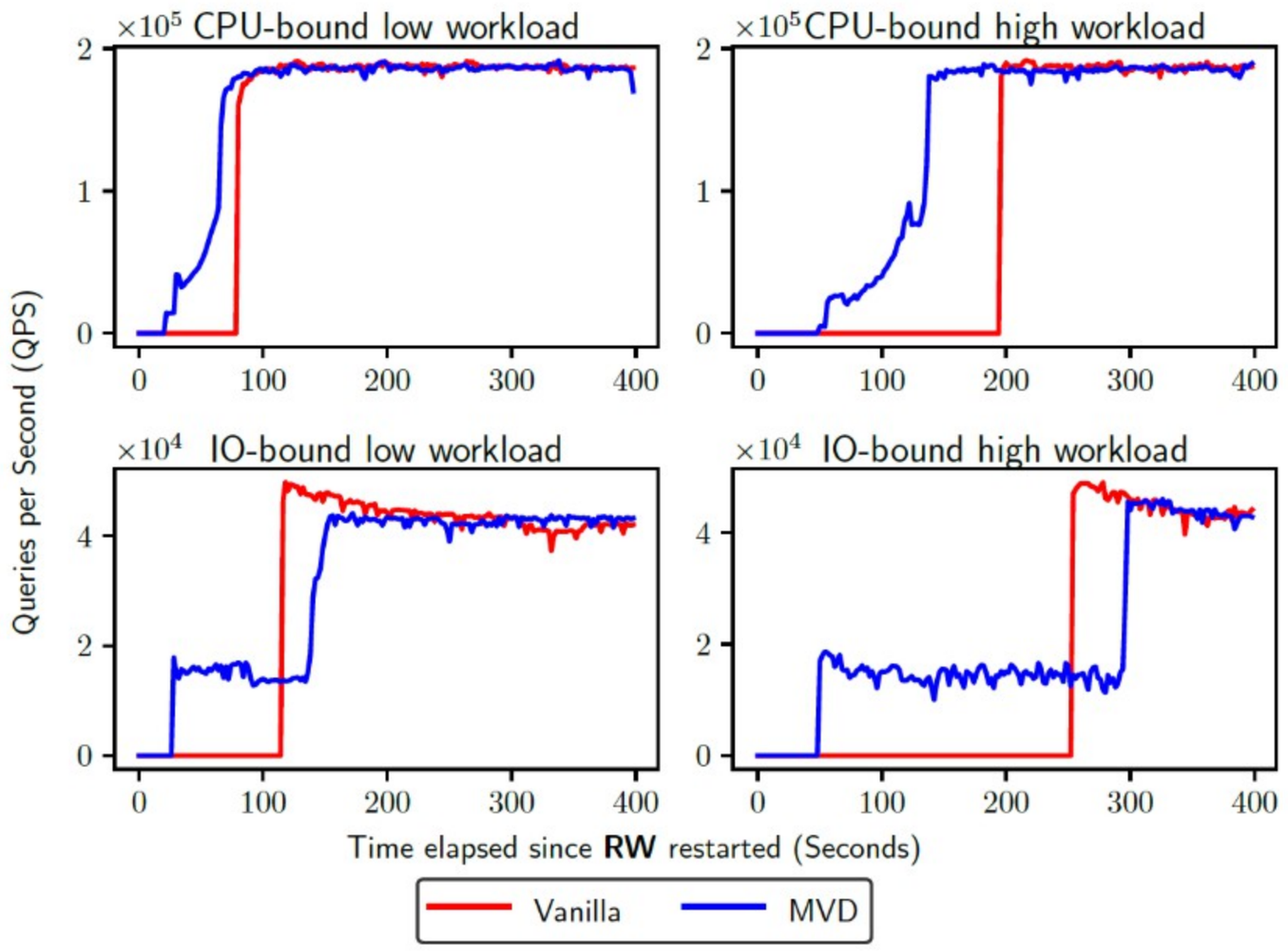
故障恢复是数据库系统的一项关键功能，旨在通过日志将数据库状态还原到发生故障之前。这一功能不仅对数据从故障中恢复至关重要，还在整个产品生命周期中支持各种管理操作，尤其是在需要重启数据库的重大变更场景下。**故障恢复的速度至关重要**，因为它直接影响用户何时可以重新访问数据库。以PolarDB为例，UNDO阶段是在服务重启后异步进行的，不会延长启动时间，因此恢复过程中最耗时的部分就是执行和应用Redo操作。这个恢复过程大致如下，1）从Checkpoint位置开始顺序扫描Redo Log，一直到找到最后一个完整的mtr；2）扫描的过程中，会不断地Parse遇到的所有Redo Log，并按照Page有序将Redo Record维护在内存的Hash Map中。3）扫描结束或者Hash Map占用的内存过多的时候，会触发异常Page Apply，也就是用Hash Map中维护Redo Record对Page内容进行重放，获得更新的Page版本。这个过程中会导致时间不可控的因素主要有三个：

1. Redo日志总量：这三个过程的耗时都是与Checkpoint之后的Active Redo量正相关的。而Buffer Pool的正常行为又会尽量积攒一批Page在内存中，再叠加大内存实例规格、压力增大、IO瓶颈等可能的情况，Active Redo的重放时间会非常可观，除此之外，在MVD Write Elistion中我们也会倾向在运行过程中适当增加Checkpoint的落后来获得更好的写IO合并。
2. 页面IO放大：Apply过程是按照Redo中的记录顺序进行的，就会导致一个Page在不同的Redo区段都有对应修改的情况，当总数据量大于Buffer Pool能缓存的Page之后，这些Page就需要不断的被从共享存储中读取及写回。
3. 没有充分利用存储特性：在分布式存储系统中，IO操作相比本地磁盘具有更高的延迟，需要更多并发读取来抵消延迟，而恢复过程的Redo应用由于Parse阶段的串行使得整体并发度不足。

Table 1: Comparison of Recovery Stages

Stage	Instant Recovery	
	Without	With
Redo Scans	• All Active redo log • Single-thread	• Only after log index • Fetch Register Page from `ib_parsemeta`
Redo Parse	• All Active redo log • Single-thread	
Redo Apply	• Synchronous • In Redo sequence • Limited parallelism	• Asynchronous background • By segments • Intra-segment concurrency
When available?	After **Redo Apply**	After **Redo Scans**

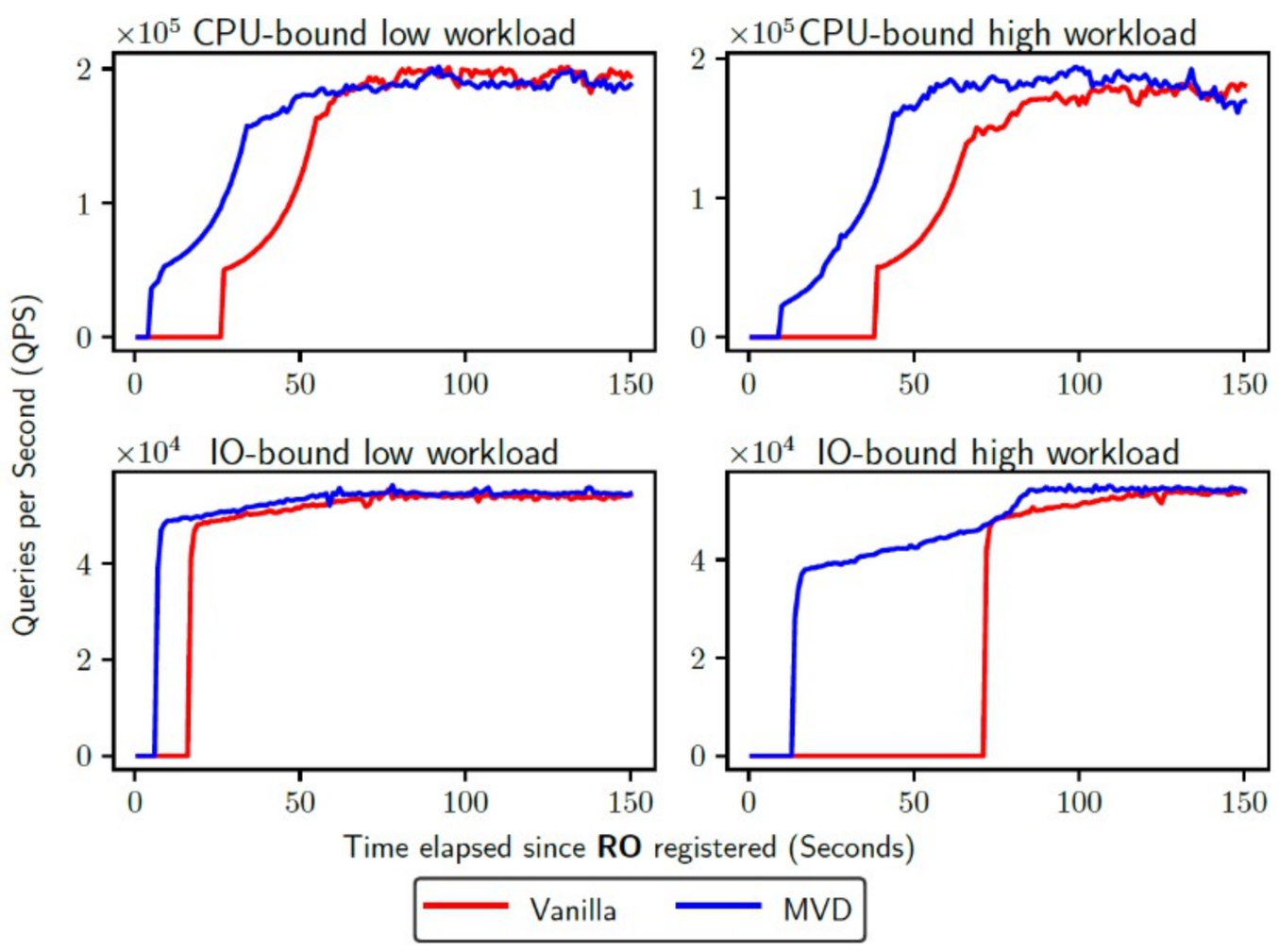
MVD 中引入的**单页恢复能力**使得可以将耗时的Redo阶段推迟到服务上线之后。借助Redo日志的页面导向特性和分布式共享存储的高吞吐能力，可显著缩短停机时间，并可能在后台加速整个恢复过程。改进后的流程包括：1）从日志索引生成位置而非Checkpoint开始扫描；2）从 ib_parsemeta 直接读取哪些页面参与了检查点后的 Redo 日志，标记为"注册页面"（Register Pages）。并将这些页面的真实恢复推迟到实例提供服务后异步进行；3）实例提供服务，用户请求出发的IO过程或后台批量恢复的任务线程触发单个Page的真实还原，完成恢复的Page从注册页面中剔除。如上的表格展示了这种MVD的恢复策略如何显著提前实例服务可用的时间，从原本需要扫描、解析、应用完整的Redo日志，简化为仅需扫描一小段日志即可。除此之外，MVD提出了分段式恢复来最大化的利用有限的内存资源加速后台的Page应用。



如上图所示MVD的快速恢复策略即使是在CPU或者是IO瓶颈的场景下，都可以大幅缩短实例不可服务的时间。

3，RO横向扩展

在高压力场景下，为了提升**横向读扩展能力**，通常会通过快速增加 RO 节点（只读节点） 的数量来实现负载分担。快速的横向扩展能力是共享存储架构的一大优势。RO刚加入集群的时候，为了获得RW内存相对于共享存储更新的数据，需要重放Checkpoint之后的Redo日志，实践上可以触发一次RW的Checkpoint位点推进，来避免这个过程RO需要重放过多的日志导致内存膨胀，无论哪种策略，都会导致RO加入集群的耗时增加，影响弹性能力。而引入MVD后，这一问题得到了根本性的改变。当一个新的 RO 节点连接到 RW 节点时，不再需要等待 Buffer Pool 的页面刷新操作；相反，它可以直接基于日志索引的位置启动复制关系。任何缺失的 Redo 日志都可以根据需要从日志索引中获取。实践上，可以保持这个位点跟最新写入位点的高度接近，从而显著提升了扩容效率和系统稳定性。如下是引入MVD前后RO加入集群并提供服务的效率对比。



4，实例还原：One-Pass Restore & Backtrack

在数据库的使用过程中，对数据的还原或回查是非常常见的业务需求，用来应对例如业务操作失误、数据错误等情况。备份还原的速度直接关系到通常很紧急的还原需求的满足周期。备份还原的过程，包括全量历史备份数据的拷贝，之后在这个备份数据的基础上应用增量的Redo，获得指定时间点的实例状态。共享存储数据库由于不需要全量历史备份的拷贝已经具有了明显的优势，但如果从备份生成时间点到还原的目标时间点，Redo日志的量很大，那么还原需要的时间依然是非常可观的。前面在故障还原中提到的，在IO瓶颈场景下，同一个Page被反复读写的情况，在实例还原场景下，由于Redo日志总量更大

会变得更加明显。这个问题的本质是，还原过程是按照Redo生成的顺序进行的，MVD提出了One-Pass Restore的核心思想是**按页面顺序而非Redo日志访问顺序进行恢复。**顾名思义，每个页面在整个恢复过程中只需经历一次读IO和一次写IO，同时应用其所需的所有Redo日志。

在One-Pass恢复过程中，针对一个页面，会通过多版本日志索引访问其所有的Redo内容。考虑到不同页面的Redo日志交错存在于连续的Redo文件中，且日志索引具有分段排序的特点，必须避免因访问Redo日志或日志索引带来的额外IO放大。为此，我们实施了一种**日志合并策略**，主要包括以下三个层面：

1. **日志索引合并**：扩展日志索引格式，使其不仅包含Redo日志的位置信息，还直接存储Redo内容。此举消除了在获取日志索引后随机访问Redo日志所带来的开销。
2. **段内合并**：如前所述，单个`ib_parsedata`文件可能包含多个段，每个段内部按页面排序，但段之间并不相连。One-Pass Restore的第一步就是合并这些段，实现文件内Page的全局有序。
3. **跨文件日志索引多路归并**：之后对所有日志索引文件进行多路归并，以按顺序获取每个页面对应的所有Redo日志进行还原。

One-Pass Restore的备份还原策略相对于传统的按Redo顺序的还原可以取得巨大提升，原因在于，1）单个Page仅读写一次消除IO放大；2）页面全局视角，提前识别页面复用，文件删除等情况避免不必要的还原；3）消除并行瓶颈，每个阶段都可以充分并发利用存算分离带来的巨大的IO带宽优势。这一综合策略有效地缓解了IO受限场景下的IO放大问题，确保了数据恢复过程的高效与简洁。

但这种需要还原到一个新的实例并等完成后提供服务的策略，在一些场景下仍然显得迟缓。对此，MVD引擎还支持提供Backtrack的能力，优先考虑服务恢复，将耗时的页面处理推迟到后台进行，并接受恢复后短期内的性能下降。当启用Backtrack时，用户可以在需要时通过控制台发出"回溯至时间戳（Backtrack to Timestamp）"命令，以恢复到指定的时间点。然后，实例将重新启动，并在重启后反映该指定目标时间点的状态，并通过日志索引在真实用户请求时完成页面状态的回溯。

总结

在CloudJump II的工作中，我们分析了云原生数据库，从计存分离更进一步到共享存储之后，在获得扩展性和安全性优势的同时，需要面对的主从节点一致性的问题。不同于之前很多云数据库采用的Page Server的方式，CloudJump提出来一种更通用的，基于标准的云存储服务来促进共享存储架构的思路，通过在计算节点层引入MVD引擎来处理主从一致性问题、增强可扩展性、恢复能力以及数据持久化能力，且无需自定义存储层。

更多的内容可以参考论文：《CloudJump II: Optimizing Cloud Databases for Shared Storage》。

相关论文：

Chen, Z., Yang, X., Sha, M., Li, F., Wang, K., Miao, Z., … & Wang, S. (2025, June). CloudJump II: Optimizing Cloud Databases for Shared Storage. In *Companion of the 2025 International Conference on Management of Data* (pp. 336-349).

Chen, Z., Yang, X., Li, F., Cheng, X., Hu, Q., Miao, Z., … & Wang, S. (2022). CloudJump: optimizing cloud databases for cloud storages. *Proceedings of the VLDB Endowment*, *15*(12), 3432-3444.