分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题 目：持久化内存应用

学　　号 M202173830

姓　　名 丁佳鹏

专　　业 电子信息

课程指导教师 施展 童薇

院（系、所） 计算机科学与技术学院

2022 年 1 月 7 日

持久化内存应用

丁佳鹏[1)]

(华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 持久化内存具有单位价格容量大、相比普通 SSD 速度快以及持久性的特点，在很多场景下有着天然的优势。本文综述了持久化内存的三个应用，分别是在 OLTP、KV 存储和 OLDA 场景下的应用。以 LevelDB 和 RocksDB 为代表的 LSM-tree、key-value 存储，在生产环境中被广泛的应用。由于历史原因，在顺序写和随机写相差 100 倍、甚至 1000 倍的 HDD 时代，LSM-tree 的收益是明显的，反之在顺序写和随机写相差不大的 SDD 时代，尤其是 NVME SSD 和 持久化内存设备的出现，远远缩小了顺序写和随机写的差距（<10 倍），所以大量研究开始思考使用持久化内存对 LSM-tree 进行优化。持久化内存针对 OLTP 负载做了优化。OLTP 负载的查询多是写密集的负载，这也是非常适合持久化内存发挥特长的场景，针对持久化内存不擅长的元数据写部分，以及持久化内存空间管理部分，论文做了相应的优化。OLDA 在实时欺诈检测、个性化推荐等方面有着广泛的应用，从多个时间窗口通过一个预先训练的模型来评估新的数据，以支持决策。特征提取是通常在许多 OLDA 数据中最耗时的操作。在论文研究中，作者首先研究了如何利用现有的内存数据库来有效支持实时特征提取，但是发现这些数据库的延迟不满足实际需求，于是研究出了新的数据库系统降低延时并以持久化内存作为硬件进一步对系统进行效率的优化。

关键词 持久化内存；OLDA；OLTP；K-V 存储

Persistent memory application

DING Jia-Peng[1)]

[1)](School of Computer Science and Technology, School of Computer Science and Technology, Wuhan 430074)

Abstract * Persistent memory has the characteristics of large unit price capacity, fast speed and persistence compared with ordinary SSD, and has natural advantages in many scenarios. This paper reviews three applications of persistent memory in OLTP, KV storage and OLDA scenarios. LSM-tree and key-value storage represented by LevelDB and RocksDB are widely used in production environment. Due to historical reasons, the benefits of LSM-tree are obvious in HDD era when the difference between sequential writing and random writing is 100 times or even 1000 times. On the contrary, in SDD era when the difference between sequential writing and random writing is not big, especially the emergence of NVME SSD and persistent memory devices, the gap between sequential writing and random writing is far reduced (< 10 times), so a large number of studies began to think about using persistent memory to optimize LSM-tree. Persistent memory is optimized for OLTP payload. The query of OLTP load is mostly write-intensive load, which is also very suitable for the scene where persistent memory plays its specialty. For the part of metadata writing that persistent memory is not good at, and the part of persistent memory space management, this paper makes corresponding optimization. OLDA is widely used in real-time fraud detection, personalized recommendation and so on. It evaluates new data through a pre-trained model from multiple time windows to support decision-making. Feature extraction is usually the most time-consuming operation in many OLDA data. In this dissertation, the author first studies how to use the existing main memory database to support real-time feature extraction effectively, but finds that the latency of these databases can not meet the actual needs, so a new database system is developed to reduce the latency and use persistent memory as hardware to further optimize the efficiency of the system.

Key words Persistent memory; OLDA; OLTP;K-V storage

1 引言

通过减少磁盘 I/O 时间，持久化内存可以极大地提高有持久性要求的系统的性能。因此，使用持久化内存作为主存储的 OLTP 数据库正在成为一种有前途的设计选择。近年来，并发控制方法的研究取得了进展。单机主存 OLTP 事务吞吐量（没有持久性）每秒已经超过一百万个事务。但是，用持久化内存取代 DRAM 会减少系统的速度，因为持久化内存执行相对于 DRAM 慢，持久化内存写的带宽要比读更低，并且将写从 CPU 缓存持久化到持久化内存会产生额外的开销。充分考虑持久化内存的性能，进行 OLTP 的引擎设计，可以提升 OLTP 引擎的事务性能。预计这样的持久存储器将具有与 DRAM 相当的读等待时间，与 DRAM 相比更高的写等待时间（高达 5 倍）和更低的带宽(5-10 倍)。持久化内存将具有比 DRAM 更高的密度和更大的容量。然而，持久化内存有望与 HDD 和 SSD 等磁盘共存。特别是，对于大规模 KV 存储，数据仍将存储在磁盘上，而新的持久存储器将用于提高性能。有鉴于此，早先已经为持久化内存重新设计了一个基于 LSM-tree 的 KV 存储库系统。然而，寻找一种基于持久化内存和磁盘混合系统的 KV 存储的新设计，其中持久化内存承载的角色不仅仅是一个大型内存写缓冲区或读缓存，对于实现更好的性能也是至关重要的。大多数实时特征都需要在多个时间窗口内计算。为了实现低延迟的特征提取，理想情况下可以利用现有的内存数据库进行这些操作。特别地，对于内存中的数据库，在线特征提取过程包括插入新记录，然后是与新记录相关的大量并发分析查询。然而，论文的研究表明，在两个最先进的内存数据库中，提取实时特征的延迟与时间窗口的数量成比例地增长。因此，对于有严格时间约束的 OLDA 系统，如果增加时间窗数目以获得更好的预测精度，则响应时间将难以接受。为了进一步降低 OLDA 的总拥有成本，论文建议利用最近的持久存储产品英特尔®Optane™DC 持久存储模块(PMEM)。与 DRAM 相比，PMEM 具有更低的每 GB 成本、更高的密度和非易失性。以前的研究已经表明，使内存中的数据结构在 PMEM 中持久化而不损害数据一致性是一项困难的任务。具体来说，论文中用一个 PMEM 持久化内存扩展了 FEDB。

2 原理和优势

由于持久化内存是字节寻址和非易失性的。持久化内存将通过内存总线而不是块接口连接，因此，写入持久化内存的原子性单元（或粒度）通常预计为 8 字节。持久化内存具有比传统存储设备更小的故障原子单元，在其中持久化数据结构时，必须确保即使系统崩溃，数据结构也保持一致。因此，需要小心地更新或更改数据结构，方法是确保内存的写入顺序。然而，在现代处理器中，存储器写入操作可以以高速缓存行为单位重新排序，以最大化存储器带宽。为了进行有序的内存写操作，需要显式地使用昂贵的内存栅栏和缓存刷新指令（Intel x86 体系结构中的 CLFLUSH 和 MFENCE)。并且，如果写入持久内存的数据大小大于 8 字节，系统故障时可以部分更新数据结构，导致恢复后状态不一致。在这种情况下，需要使用常规的技术，如日志记录和 Copy-On-Write（CoW）。因此，需要对持久化内存中持久化的数据结构进行仔细的设计。持久化内存为克服现有 KV 存储的缺点打开了新的机会。利用永磁材料制造 KV 储存器已引起越来越多的兴趣。基于 LSM-tree 的 KV 存储已经开始用持久内存重新设计。然而，探索基于持久内存的 KV 存储的新设计也很重要。在后文的介绍中，论文提出了一个 KV 存储的设计，它使用了在持久化内存中的索引持久化。

由于持久化内存的特点：像 DRAM 一样是字节可寻址的，比 DRAM 慢一些，但比 HDDs 和 SSD 快几个数量级，提供非易失性可以比 DRAM 大得多的主存储器（例如，在双套接字服务器中高达 6TB)，相比于读数据，它的写操作具有较低的带宽以及确保数据在断电后的持久性，使用高速缓存线 flush 的特殊持久性操作并且需要内存栅栏指令（例如，clwb 和 sfence）

将数据从 CPU 缓存持久保存到持久化内存，导致明显高于正常写操作的开销。得到了三个常见的设计原则：(i)将经常访问的数据放入持久化内存中，如果它们是需要持久化的或者需要在恢复时重建；(ii)尽可能减少写入； (iii)尽可能减少持久性操作。把这些设计原则应用到 OLTP 引擎的设计中。

在工业级的应用中，巨大的内存需求和实际供给之间的差距显著增加了硬件的成本，DRAM 容量的限制明显增加了各种解决方案的总成本。与之相比的非易失性随机存取存储器（NVRAM）提供了

更大的容量并能够很好地解决成本问题。NVRAM 是一种持久存储技术，能够提供字节可寻址随机访问并且在断电时也能保存数据。英特尔的傲腾持久内存（PMEM）是最早的商业产品。PMEM 能够在解决内存容量的不足问题中提供相近的性能同时维持低廉的成本。PMEM 可以用两种模式使用，一种是内存模式，另一种是 AD 模式。

内存模式即把 PMEM 当作一个直接和缓存相联的 DRAM，操作系统可以直接使用 PMEM 作为一块大内存。但是即使 PMEM 是非易失性的，在这种模式下使用 PMEM 不能维持数据的持久化。这种模式不需要编程，而是直接替换 DRAM 使用。

AD 模式能够让程序员进行编程实现 PMEM 的持久化内存功能，通过使用相应的编程 api 重新设计持久化数据结构和逻辑，实现期望中的内存数据持久化，唯一缺点就是带来的额外的开发工作量。

3 研究进展

在本节中，本文将介绍三种近年来持久化内存的主要应用：分别是在 Zen、SLM-DB 和 OLDA 系统下的应用。

3.1 Zen

Zen 的体系结构。每个基本表中都有一个混合表（HTable）。它由 NVM 中的元组堆、DRAM 中的 Met-Cache 和每个线程的 NVM 元组管理器组成。然后，Zen 在 NVM 元数据中存储表模式和粗粒度分配结构。另外，Zen 将索引和事务私有数据保存在 DRAM 中。

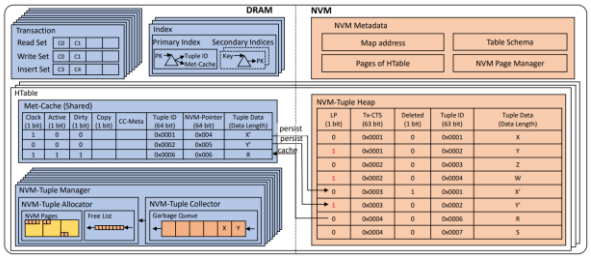

图 1 Zen 架构

元组堆：每个 NVM-元组是 NVM 中的持久元组。Zen 将基表中的所有元组存储为 NVM 元组堆中的 NVM-元组。每个堆由固定大小（例如 2MB)的页组成。每一页包含一个固定数目的 NVM-元组位。NVM-元组由 16B 的报头和元组数据组成。NVM-元组堆可能包含逻辑元组的多个版本。元组 ID 和事务提交时间戳（Tx-CTS）唯一标识元组版本。删除位显示逻辑元组是否已被删除。最后持久化(LP)位显示元组是否是提交事务中持久化的最后一个元组。LP 位在无日志事务中起着重要作用。其中头部不包含用于特定领域的并发控制方法。NVM-元组插槽与 16B 边界对齐，使得 NVM-元组的报头始终驻留在单个 64B 高速缓存行中。通过这种方式可以使用一个 clwb 指令，和 sfence 来持久化 NVM-元组头。

Met-Cache：Met-Cache 为相应的 NVM-元组堆管理 DRAM 中的 tuple-grain 缓存。Met-Cache 包含元组数据和七个元数据字段：指向 NVM-元组的指针（如果存在）、元组 ID、脏位、指示该条目可能被活动事务使用的活动位、为了支持缓存替换算法的时钟位，需要一个 copy 位来指示条目是否已被复制，以及一个 CC-Meta field，该 CC-Meta field 包含对正在使用的并发控制方法指定的额外的每个元组元数据。Zen 支持广泛的并发控制方法。使用 Met-Cache，可以让 Zen 完全在 DRAM 中执行并发控制。

DRAM 中的索引：论文为 DRAM 中的每个 HTable 维护索引。在崩溃恢复时重建表。主索引是必需的，辅助索引是可选的。对于主索引，索引键是元组的主键。该值指向(i)Met-Cache 或(ii)NVM 元组堆中元组的最新版本。论文使用该值的一个未使用的位来区分这两种情况。对于辅助索引，索引值是元组的主键。Zen 要求索引结构支持并发访问，事务只能看到提交的索引项（以前由其他事务修改）

事务-私有数据：Zen 支持并发处理事务的多个线程。每个线程在 DRAM 中为事务私有数据保留一个局部线程空间。它记录事务的读、写和插入活动。OCC 和 MVCC 变体将读、写和插入集作为独立的数据结构进行维护。2PL 变体以日志条目的形式存储更改。

NVM 空间管理：Zen 使用两级方案来管理 NVM 空间。首先，NVM 页面管理器执行页级空间管理。它分配和管理 2MB 大小的 NVM 页面。映射地址和 NVM 元数据中的 HTable 页面维护从 NVM 页面到 HTable 的映射。第二，NVM-元组管理器执行元组级空间管理。每个线程为该线程访问的每个 HTable 拥有一个线程本地 NVM-元组管理器。每个 NVM-元组管理器由一个 NVM-元组分配器和一个 NVM-元组收集器组成。分配器在 HTable 中维护空闲 NVM 元组插槽的不相交子集。其中有两种空闲

插槽：新分配页中的空插槽或垃圾回收的插槽。论文中提到在系统开启时的时刻初始化 NVM，并使用 Tx-CTS=0 表示空槽。收集器垃圾收集陈旧的 NVM-元组并将其放入空闲列表。同一 HTable 的所有收集器协同工作以回收 NVM-元组。

3.2 SLM-DB

Algorithm 1 Insert(*key*, *value*, *prevNode*)

```
1: curNode := NewNode(key, value);
2: curNode.next := prevNode.next;
3: mfence();
4: clflush(curNode);
5: mfence();
6: prevNode.next := curNode;
7: mfence();
8: clflush(prevNode.next);
9: mfence();
```

算法 1 插入操作

在 SLM-DB 中，MemTable 是一个持久化的跳表。跳表的操作，如插入、更新和删除，可以使用原子级的 8 字节写操作来完成。算法 1 描述了插入到跳表的最低一级的过程。为了保证 MemTable 中 KV 操作的持久化，首先需要持久化一个新节点，通过调用内存栅栏和缓存线刷新指令来设置它的下一个指针，然后更新前一个节点的下一个指针（8 字节），并保存更改。更新 MemTable 中现有的 KV 对的方法类似，不需要对值进行就地更新（类似于 LevelDB 的 MemTable 更新操作）。通过使 PM 保存 MemTable，SLM-DB 无需依赖 WAL 来实现数据的持久性。类似地，对于较高级别的跳表不需要持久化保存机制，因为在系统故障时，它们可以很容易地从最低级别重新构建。

SLM-DB 采用 B+-树索引，以加快对存储在 SSTable 中的 KV 对的搜索速度。当将不可变 MemTable 中的 KV 对刷新到 SSTable 中时，键被插入到 B+-树中。该键被添加到 B+-树的叶节点，指针指向一个 PM 对象，该对象包含关于该 KV 对存储在磁盘上的位置信息。密钥的位置信息包括 SSTable 文件 ID、文件内的块偏移量和块的大小。

如果 B+-树中已经存在一个键（即更新），则将该键的新值写入新的 SStable。因此，为键创建一个新的位置对象，并更新它在 B+-树叶节点中的关联指针，以失败原子方式指向新的 PM 对象。如果插入了键的删除标记，则从 B+-树中删除该键。位置对象的持久内存分配和解除分配由持久内存管理器（如 PMDK）管理，过时的位置对象将由管理器进行垃圾回收。其中，SLM-DB 像 LevelDB 一样支持 stringtype 键，并且当 string-type 键添加到 B+-树时，它将被转换为整数键。在 SLM-DB 中建立一个 B+-树的过程是，当固化 MemTable 刷新时，在 B+-树中插入固化 MemTable 中的 KV 对。对于刷新操作，SLM-DB 创建两个后台线程，一个用于创建文件，另一个用于插入 B+-树。在文件创建线程中，SLM-DB 创建一个新的 SSTable 文件，并将 KV 对从固化 MemTable 写入该文件。一旦文件创建线程将文件刷新到磁盘，它就会将存储在新创建的文件上的所有 KV 对添加到一个队列中，该队列由一个 B+-树插入线程创建。B+-树通过将队列中的 KV 对逐一插入到 B+-树中插入线程。一旦队列变为空，插入线程就完成了。然后，将 LSM-树组织的更改（即，SSTable 元数据）作为日志附加到清单文件中。最后，SLM-DB 删除固化 Memtable。扫描 B+-树的方法，SLM-DB 提供了一个迭代器，它可以用来扫描 KV 存储中的所有键，方式类似于 LevelDB。迭代器支持 seek、value 和 next 方法。seek(k)方法能定位在 KV 存储中的迭代器，使得迭代器指向键 k 或比 k 大的最小的键。next（）方法将迭代器移动到 KV 存储中的下一个键，而 value（）方法返回迭代器当前指向的键的值。在 SLM-DB 中，实现了一个 B+-树迭代器来扫描存储在 SSTable 文件中的密钥。对于 seek(k)方法，SLMDB 在 B+-树中搜索键 k，以定位迭代器。在 FAST+FAIR B+-树中，键在叶节点中排序，叶节点有一个同级指针。因此，如果 k 不存在，它可以很容易地找到大于 k 的最小密钥。另外，通过将迭代器移动到 B+-树叶节点中的下一个键，很容易支持 next（）方法。对于 value（）方法，迭代器查找键的位置信息，并从 SSTable 中读取 KV 对。

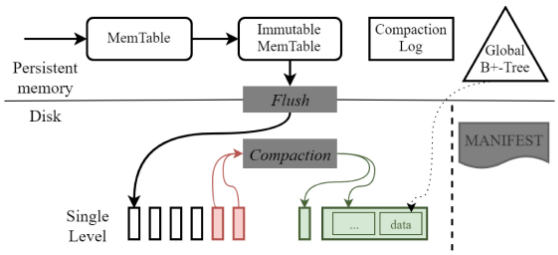

图 2 SLM-DB 架构

3.3 OLDA

论文首先在 FEDB 代码中使用最小修改的 PMEM（即用 PMEM 直接替换 DRAM）。与基于 DRAM+SSD 的存储引擎（图中最左边的子图）相比，使用内存模式下的 PMEM 作为核心数据结构（中间子图）的工作存储器不需要更改代码。相反，

使用 Intel 提供的工具(ipmctl)来混淆 PMEM，使操作系统将其视为主存。然而，这种直接的方法仍然需要在 SSD 中保留日志和快照而不能利用 PMEM 的非易失性。在 AD 模式下使用 PMEM，并利用其非易失性创建一个新的存储引擎，使得无需日志和快照就可以保证数据的可恢复性。为了实现这一点，论文实现了一个基于 PMEM 的持久跳表并将其集成到双层跳表结构（图中最右边的子图）中。实验结果表明，与 DRAM+SSD 方法相比，该方法不仅消除了同步日志对性能的负面影响，而且实现了系统重启后的实时恢复。

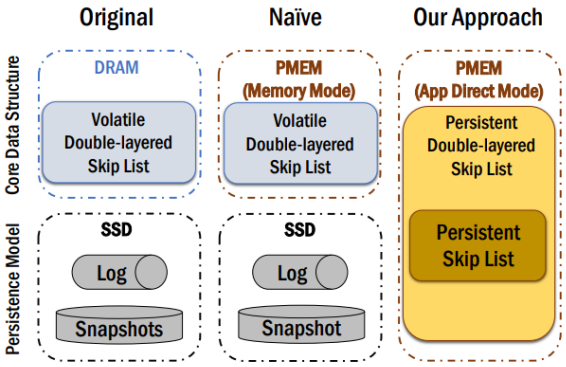

图 3 FEDB 中使用 PMEM 的不同方式

为 PMEM 实现持久化跳表并不简单，因为它需要额外的逻辑来处理持久化内存的空间管理和系统故障时内存中数据的一致性。实现持久化数据结构有两个主要难题：1.原子级的持久内存分配/释放对于持久内存的空间管理至关重要。例如，当在分配对象的持久内存空间和持久存储其地址之间发生系统故障时，可能会发生内存泄漏和悬空指针问题。2.由于修改内存中数据结构的大多数基本操作不能在单个 CPU 指令中完成，因此系统故障时的数据一致性会出现问题。因此，当系统发生故障时，在任何时间点中断执行都可能损坏整个数据结构。在此之前，必须做出额外的操作来保证数据一致性。此外，内存存储指令不能保证数据可以持久化在 PMEM 中，除非特殊指令-FLUSH，而 FLUSH 的内存对象超过了 8 个字节，不是原子级。

论文通过调用 Intel 的 PMDK 和 libpmemobj-cpp 提供的 API 来原子级地分配/释放持久内存来解决第一个问题。在内部，PMDK 跟踪持久化中分配/释放的所有对象的地址内存，以便在恢复期间可以撤消/重做中断的操作。使基于指针的数据结构在 PMEM 中持久化。虽然成本较高，但节省了大量的工程工作量。在论文的持久化跳表的实现中，通过手动管理持久化跳表节点的“空闲列表”，以从数据处理的关键路径中删除分配/释放操作。因此，可以从使用 PMDK 的简单实现中受益，而不会对性能造成太大损害。

解决第二个问题的两种现有方法。第一个是在用户应用程序本身的数据处理逻辑中实现传统的日志/复制-写入(COW)算法。这种方法对用户来说很难实现，而且它可能容易出错。第二种方法是使用 PMDK 中的事务支持，它可以保证多个内存操作的原子性，这在内部使用 COW。虽然这两种方法日志/COW 都能够保证正确性，但它们都出现本身的和额外的写开销问题。因此，在论文中持久化跳表的实现都不采用这两种方法。受现有持久内存数据结构的启发，论文重新设计了持久跳表中的插入、删除和搜索过程，使其仅使用 8 字节的原子写以及 FLUSH。通过这种方式，可以在系统故障时保持数据一致性，而无需使用昂贵的日志/COW。

CompareAndSwap(CAS)技术是持久化跳表在单写入器-多读取器场景中维护非阻塞执行的关键。然而，如果它在持久内存中运行，需要处理由先写后读依赖关系引起的可能的不一致性，其中一个线程持久地写入从读取一些可能不持久的数据的结果中计算/导出的新数据。这个问题可以通过 flush-on-read 方法解决：对此类数据的任何读取操作都必须在刷新之前进行。

为了高效实现 flush-on-read 来保证 CAS 能在持久内存中正确使用。论文利用 64 位机器上正常的 64 位指针中的低四位空位，进一步改进它，将“deleted”和“dirty”位嵌入到特殊指针中，表示为 SmartRef。SmartRef 本质上是一个 64 位无符号整数（大小与普通指针相同），它使用最低位作为“dirty”位，第二个最低位作为“deleted”位。PersistRead（）将 SmartRef 作为输入。如果它被标记为脏，它将被刷新。

鉴于 PMEM 中的写操作比 DRAM 中的写操作慢，论文进一步优化了基于 PMEM 的持久跳表,减少了 PMEM 上的写操作次数。如图所示，只有级别 0 上的下一个指针（即 SmartRef）和实际数据是持久存储在 PMEM 中，而其余级别上的则使用 DRAM 中的正常指针。论文还开发了一个恢复过程来在系统故障时重建它们，该过程通过级别 0 上的 SmartRef 扫描跳表并根据每个节点的高度重新链接上层指针。在其他数据结构中，恢复时重建的概念已经被广泛地应用于持久存储器。

```
Algorithm 1: RebuildUpperLevel
  Input: head
  Output: success
1 for i=1:maxHeight-1 do
2 |  cur[i] ← head;
3 end
4 cur0 ← head;
5 while cur0 ≠ tail do
6 |  next0 ← PersistRead(cur0.next[0]);
7 |  for i=1:cur0.height-1 do
8 |  |  cur[i].next[i] ← next0;
9 |  |  cur[i] ← next0;
10|  end
11|  cur0 ← next0;
12 end
13 for i=1:maxHeight-1 do
14 |  cur[i] ← tail;
15 end
16 return TRUE;
```

算法 2 重建指针代码

```
Algorithm 2: Insert
  Input: K, V
  Output: success
1  found ← find(K, preds[], succs[]);
2  if (!found) then
3  |  new_node ← GetFreeNodeAndInitial(V);
4  |  new_node.level ← RandomHeight();
5  |  for i=0:level-1 do
6  |  |  AtomicStore(new_node.next[i],
   |  |   SmartRef(succs[i], FALSE, FALSE));
7  |  end
8  |  for i=0:level-1 do
9  |  |  while TRUE do
10 |  |  |  pred ← preds[i];
11 |  |  |  succ ← succs[i];
12 |  |  |  expected ← SmartRef(succ, FALSE, FALSE);
13 |  |  |  dirty ← (i == 0?TRUE : FALSE);
14 |  |  |  if CAS(pred.next[i], expected,
   |  |  |   SmartRef(new_node, FALSE, dirty)) then
15 |  |  |  |  break;
16 |  |  |  end
17 |  |  |  if i=0 then
18 |  |  |  |  PersistRead(pred.next[i]);
19 |  |  |  end
20 |  |  end
21 |  end
22 |  RemoveFromFreeList(new_node);
23 |  return TRUE;
24 end
25 return FALSE;
```

算法 3 插入跳表代码

算法 2 显示了系统恢复时在上层重建下一个指针的伪代码，具体步骤如下：（1）初始化迭代器类结构 cur[1.……max_height-1]，该结构表示每一层（除 0 层外）的迭代器所指向的节点（第 1-3 行）；（2）通过级别 0 上的下一个指针遍历跳表(cur0.next[0]，其中 cur0 表示级别 0 的迭代器指向哪个节点），对于每一个 cur0，重新链接 1 到 cur0 高度之间所有级别的下一个指针，以指向 cur0.next[0]（第 4-12 行）；（3）在到达 0 级的尾部时，链接下一个每一层上最后一个节点指向尾部的指针（第 13-15 行）。这个搜索过程类似于标准的基于 DRAM 的跳表，除了（1）在进入目标节点之前执行 PersistRead（）和（2）忽略标记为“要删除”的节点。删除过程从持久化地将目标节点标记为“to delete”开始，然后利用 CAS 从顶层向下更新前面节点的下一个指针（级别 0 上的 SmartRef）到级别 0。

插入从搜索具有所提供键值对的新节点的位置开始插入。在前一个和后一个节点上，新节点在每一层上的下一个指针被链接到后一个节点上，然后更新前一个节点在每一层上的下一个指针以指向新节点。算法 3 显示了将键值对<k，v>插入持久跳表的伪代码。所涉及的步骤是：（1）用 K 定位插入新节点的位置，并通过 find（）获得各级前/后节点（行 1）；（2）调用 GetFreeNodeAndInitial（）从空闲列表中获取一个空闲节点（通过 make_persistent_atomaby()API 预分配），并用 V 初始化它，下一个指针指向后面的节点（第 3-7 行）；（3）从级别 0 到顶层，将所有级别上的节点的下一个指针更新为新的节点（8-21 行）；（4）从空闲列表中删除新节点（第 22 行）。当 CAS 完成时，只有前面节点中级别 0 上的 SmartRef 标记为“dirty”。

4 总结与展望

对于 Zen 来说，有两个进一步的设计方向。为了支持更快的恢复速度，需要使用持久化索引；为了支持集群扩展，需要考虑 RDMA 网络和日志系统。替代索引设计：在目前的设计中，论文将这些指标放在 DRAM 中，并证明了方案的合理性。其中索引设计与 Zen 的三个主要技术正交，即 MetCache、无日志持久事务和轻量级 NVM 空间管理器。可以使用像 NVTree、WB-Tree、FP-Tree、HiKV 和 LB+Tree 这样的持久索引以提高恢复性能。此外，可以利用以前的索引设计来减少索引的 DRAM 空间消耗。双阶段混合索引体系结构通过将老化的索引条目放置到更紧凑的结构中来节省空间。可以选择性将 NV-Tree，FP-Tree 和 LB+-tree 处的持久化 DRAM 中 B+-树的非叶节点和 NVM 中的叶节点。这些替代设计与原始的基于 DRAM 的

索引相比较已被证明具有类似的索引性能。基于 DRAM 的日志：Zen 删除了可持久化内存上的日志以减少写入来获得更大的 OLTP 吞吐量，但是论文中提到可选择性地写入基于 DRAM 的日志以支持日志的热备份，并存档日志以支持时间点恢复和灾难恢复。因为不需要日志持久化到 Zen 中并且不需要“提前写”，这种方式可以在对事务性能影响很小的情况下处理。

在优化 K-V 存储的过程中，论文中的研究也存在一些问题：B+ 树的引入带来的写放大问题如何解决、B+ 树索引结构的维护的一致性如何保证、针对大量小写的情况，B+ 树是否会成为对应的性能瓶颈，这些还需要进一步的研究。另外，已经有一些研究来提供最佳的持久数据结构，如 radix tree、哈希表和持久化内存的 B+-树，它们在保持持久化内存中 8 字节原子写入失败的数据结构一致性的同时提供了最佳的写技术。

OLDA 系统的论文作者目前正在准备一个实用程序库，以允许用户使用他们的持久化跳表并对类似的系统进行优化。同时，还致力于整合 SparkSQL 和论文中的特征数据库 FEDB。这将让 SparkSQL 用户使用 FEDB 无缝地加速他们的人工智能驱动程序。

参 考 文 献

[1] Chen, C. , Yang, J. , Lu, M. , Wang, T. , & Rudoff, A. . (2021). Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory. Proceedings of the VLDB Endowment, 14(5), 799-812.

[2] Liu, G. , Chen, L. , & Chen, S. . (2021). Zen: a high-throughput log-free oltp engine for non-volatile main memory. Proceedings of the VLDB Endowment, 14(5), 835-848. 0

[3] Kaiyrakhmet, O. , Lee, S. , Nam, B. , Noh, S. H. , & Choi, Y. . (2019). SLM-DB: single-level key-value store with persistent memory.