具有持久内存的 OLTP 存储引擎综述

邓天聪[1)]

[1)](华中科技大学计算科学与技术学院武汉 430000)

摘　要　持久内存 (PM) 通过在内存总线上实现持久性、高性能和（近乎）即时恢复，从根本上改变了数据库索引结构的构建方式。先前的工作提出了许多技术来为 PM 定制索引结构设计，但由于缺乏真正的 PM 硬件，它们大多基于具有模拟的易失性 DRAM。直到今天还不清楚这些技术将如何在真正的 PM 硬件上实际执行。最近的字节可寻址和大容量商业化持久内存 (PM) 有望将数据库即服务 (DBaaS) 推向未知领域。3DXpoint 等新兴的非易失性内存 (NVM) 技术有望为 OLTP 数据库带来巨大的性能潜力。文章一研究了如何利用 PM 重新审视传统的基于 LSM 树的 OLTP 存储引擎，该引擎专为 DBaaS 实例的 DRAM-SSD 层次结构而设计，(1) 提出了一个为 LSM-tree 定制的名为 Halloc 的轻量级 PM 分配器，(2) 利用 PM 的持久内存写入构建一个高性能的半持久性记忆表，(3) 设计一个名为的并发提交算法重新排序环以实现 OLTP 工作负载的无日志事务处理和 (4) 将全局索引呈现为具有非阻塞内存压缩的新全局排序持久级别，文章二提出了 Zen 解决了三个设计挑战：元数据增强元组缓存、无日志持久事务和轻量级 NVM 空间管理，文章三使用混合索引降低主存 OLTP 数据库的存储开销。具有内存压缩的半持久内存表和全局索引的设计使 PM 中的字节可寻址持久级别成为可能。

关键词　持久内存，存储引擎，内存压缩，非易失性内存，空间管理

中图法分类号　TP391

A Survey of OLTP Storage Engine with Persistent Memory

Deng Tiancong[1)]

[1)](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

Abstract　To ensure the correctness and efficiency of floating-point programs, automatic optimizing techniques for floating-point programs have been attached attention from academia in recent years. The core idea is to summarize transformation rules from classical numerical analysis theory and rewrite floating-point programs automatically with more stable algorithms, which produces more stable, efficient and maintainable programs. However, efficiency becomes the bottleneck of these techniques. As more and more transformation rules are discovered and summarized, the rule base used by the optimizing framework becomes larger and larger. It is more difficult to scan the whole rule base as usual. In this paper, we propose an experience-guided acceleration strategy for floating-point optimization. Based on the principle of similar causes of floating-point errors, the strategy extracts the corresponding symbolic and structural features of successfully optimized floating-point programs, and saves the rule sequences involved in the optimization of the programs into a hash-based optimization experience database. When optimizing a floating-point program, the optimizer first calculates the similarity between symbolic and structural features of the program and that of records in the experience base. Rule sequences associated with records having higher similarity are preferred to be used in rewriting and optimizing the program. With more and more programs to be optimized, the experience base increases gradually and a hashed-partition design ensures the efficiency of record-searching. The above approach has been evaluated on the FPBench benchmark and the open source engine OpenRelativity. The results show that compared to the traditional optimizing approach, it can achieve a 6.04x speedup in average. In addition, it improves accuracy for floating-point programs effectively as well as the traditional optimizing approach does. Therefore, our approach improves the usability of automatic optimizing techniques significantly.

Key words　acceleration strategy; program optimization; floating point programs; experience base

1　引言

下一代可扩展持久内存 (PM) 承诺在内存总线上具有低延迟（与 DRAM 相比）、字节寻址能力、可扩展性（大容量）和非易失性。这些特性使 PM 对 OLTP 系统中的索引结构（例如，B+-Tree 及其变体）很有吸引力：索引可以直接访问并保存在 PM 中，并且（几乎）立即恢复，节省了大量重建/加载时间，提高了性能（与基于磁盘的索引相比），并减轻管理大型索引的工作。

字节可寻址非易失性存储器 (NVM) 是一种新

型存储器技术，旨在解决 DRAM 缩放问题。NVM 提供了接近 DRAM 的速度、低于 DRAM 的功耗、经济实惠的大容量（双插槽服务器中高达 6TB）内存容量以及电源故障时的非易失性的独特组合。通过消除磁盘 I/O，NVM 可以显着提高具有持久性要求的系统的性能。因此，使用 NVM 作为主存储的 OLTP 数据库正在成为一种有前途的设计选择。

最近商业化的持久内存 (PM) 产品在推动数据库即服务 (DBaaS) 进入未知领域方面具有巨大潜力。与 DBaaS 实例中使用的主流易失性 DRAM（通常为 8GB 到 32GB）相比，PM 可以更大（数百 GB 或更大）并且以经济上可行的成本持久存在，并且具有相同的字节寻址能力。目前，许多用于 OLTP 工作负载的 DBaaS 系统依赖于持久事务的同步日志记录。

宝贵的 DRAM 容量迫使数据库以稳定的性能为代价频繁刷新脏页，或者放弃处理大内存占用的分析查询。通过 PM，我们现在能够重新审视 DBaaS，尤其是底层存储引擎，如何查看和利用主内存。例如，与 DRAM 相比，英特尔 3D XPoint 内存 [1] 是一种字节可寻址、大容量且持久的主内存。在本文中，我们利用它来实现具有竞争力的整体性能水平的持久内存写入，为未来的更多可能性铺平道路。作为起点，我们的工作基于流行的基于 LSM 树 [42] 的键值存储引擎。此类引擎已广泛用于 DBaaS 中的各种工作负载 [26, 39]。目前，这些引擎部署在传统的 DRAM-SSD 存储层次结构上。尽管 LSM-tree 数据结构本身通过仅追加插入实现了快速写入，但是 ACID 兼容事务所必需的同步日志记录会将慢速磁盘 I/O 拖入写入管道并限制最终写入吞吐量。同样，就消耗的 CPU 和 I/O 而言，定期将内存中的增量与 LSM 树中基于 SSD 的基础合并是必要且昂贵的。这些问题在公共云中变得越来越麻烦，其中数据库通常部署在多租户虚拟机上，客户为事务/查询处理付费，而不是为昂贵的后台操作付费。

最近对并发控制方法的研究已经将单机主内存 OLTP 事务吞吐量（无持久性）提高到每秒超过 100 万个事务。然而，在系统中用 NVM 替换 DRAM 往往会减慢系统速度，因为 NVM 的执行速度比 DRAM 慢（例如，2-3 倍），NVM 写入的带宽低于读取的带宽，并且从 CPU 缓存到 NVM 的持久写入会产生额外的开销。在本文中，我们将充分考虑 NVM 的特性，重新思考 NVM 的 OLTP 引擎的设计。我们的目标是实现类似于基于纯 DRAM 的 OLTP 引擎的事务性能。

2 原理和优势

2.1 NVM 特性

有几种相互竞争的 NVM 技术，包括 PCM、STT-RAM、忆阻器和 3DXPoint。它们具有相似的特征：(i) NVM 像 DRAM 一样可按字节寻址；(ii) NVM 比 DRAM 慢一些（例如，2-3 倍），但比 HDD 和 SSD 快几个数量级；(iii) NVM 提供非易失性主存储器，它可以比 DRAM 大得多（例如，在双插槽服务器中高达 6TB）；(iv) NVM 写入的带宽低于 NVM 读取的带宽；(v) 为确保断电时 NVM 中的数据一致，需要使用缓存行刷新和内存栅栏指令（例如 clwb 和 sfence）的特殊持久性操作将数据从易失性 CPU 缓存持久化到 NVM，从而导致显着更高的开销比正常写入；(vi) NVM 单元可能会在有限数量（例如 108 个）写入后磨损。

根据之前基于 NVM 的数据结构和系统的工作，我们获得了三个常见的设计原则：(i) 如果频繁访问的数据结构是暂时的或可以在恢复时重建，则将它们放入 DRAM 中；(ii) 尽可能减少 NVM 写入；(iii) 尽可能减少持久化操作。我们希望将这些设计原则应用到 OLTP 引擎设计中。

2.2 主存数据库中的 OLTP

主存 OLTP 系统是为 NVM 设计 OLTP 引擎的起点。我们考虑并发控制和崩溃恢复机制来实现 ACID 事务支持。

最近的工作研究了高吞吐量主内存事务的并发控制方法。主内存 OLTP 设计利用了乐观并发控制 (OCC) 和多版本并发控制 (MVCC)，而不是使用两相锁定 (2PL)，这是传统面向磁盘的数据库中的标准方法以获得更高的性能。MOCC 是一种基于 OCC 的方法，它利用锁定机制来处理热元组的高冲突。上述方法的一个共同特点是它们使用元数据扩展每个元组或元组的每个版本，例如读/写时间戳、指向不同元组版本的指针以及用于验证和提交处理的锁定位。这些方法在没有持久性的情况下实现了每秒超过一百万个事务 (TPS) 的事务吞吐量。

与传统数据库类似，主内存数据库 (MMDB) 将日志和检查点存储在持久存储（例如 HDD、SSD）上，以实现持久性。主要区别在于所有数据都适合 MMDB 的主内存。因此，只需要将提交的状态和

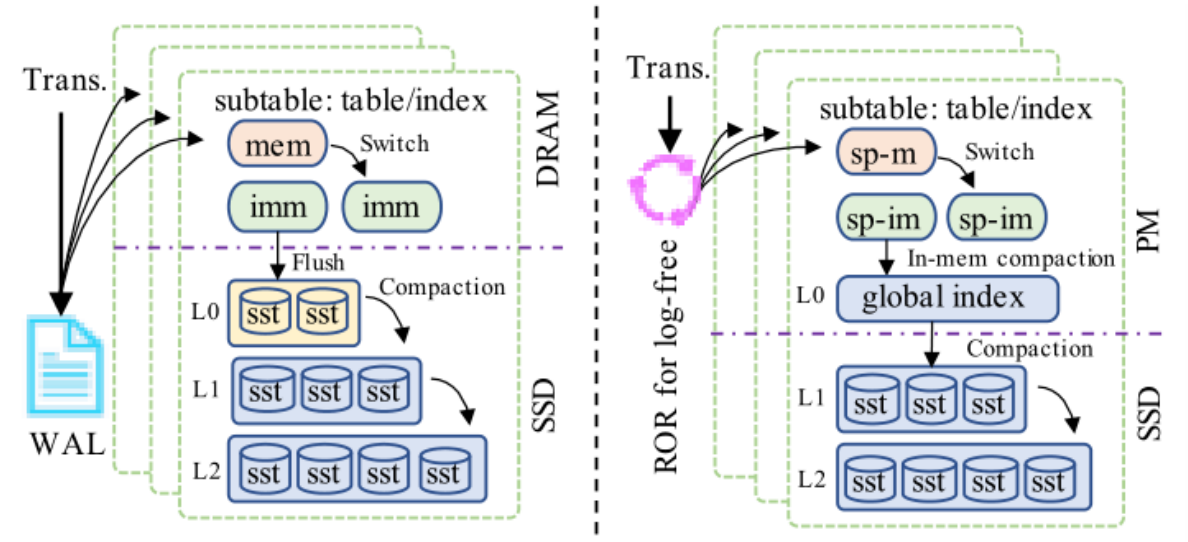

图 1　最先进的基于 LSM 树的 OLTP 存储引擎 v.s. 最新建议

重做日志写入磁盘。崩溃后，MMDB 通过将最新的检查点从持久存储加载到主内存中来恢复，然后读取重做日志并将其应用到崩溃点。

2.3　基于 LSM-tree 的 OLTP 引擎

如图 1 左半部分所示，典型的基于 LSM-tree 的 OLTP 存储引擎在一个数据库中使用多个 LSM-tree 实例，每个实例用于存储一个表，或者一个表的一个分区，或者一个索引，连同 WAL 以实现符合 ACID 的事务。每个 LSM 树实例通过仅附加方法在快速 DRAM 中缓冲更新或插入，并通过多级合并树将数据作为排序运行按顺序保存到磁盘。LSM-tree 的核心原则是应用不适当的更新并使用顺序磁盘访问来避免磁盘中的随机 I/O。由于插入的 KV 对首先缓存在 volatile 记忆表中，WAL 应写入持久存储以保证持久性。当前的 LSM-tree 实现，也使用 WAL 作为事务日志来保证持久性。当系统崩溃时，重放 WAL 以使系统进入一致状态。较大的 WAL 大小会导致较长的恢复时间。因此，必须经常将内存表刷新到磁盘以清除 WAL。

磁盘中的数据通常由具有固定容量比率 $T(T \geqslant 2)$ 的指数级增加 $(L_0, L_1, \cdots, L_n)$ 组成。磁盘中的数据级别通常由分区的 SSTable（排序字符串表）组织，并通过后台压缩慢慢地向下层级。由于级别通常包含不相交的键集，因此读取操作应以自上而下的方式访问这些级别。执行压缩以清除陈旧值并在磁盘级别应用删除，这比刷新要重得多，并且可能会阻止刷新或导致低级别的数据块堆积。因此，为 DRAM-SSD 设计的现代基于 LSM 树的 OLTP 引擎通常使用的分层压缩来提供快速刷新并减少压缩开销。然而，设计导致 L_0 中的数据块无序，特别是在写入繁重的工作负载的情况下，这带来了更高的读取放大。

2.4　持久内存

字节可寻址持久存储器使 CPU 能够在提供持久性的同时，通过加载和存储指令直接访问数据。新硬件提供比 DRAM 更低的功耗和成本以及更大的容量。不幸的是，由于 PM 和 DRAM 之间的编程模型存在很大差异，因此使用此类设备并非易事。如图 2 中的英特尔 DCPMM（一种商业化 PM）所示，从 CPU 寄存器到 DCPMM 控制器的大部分数据写入路径都是易失性的。Intel 启用 ADR（异步 DRAM 刷新）功能以保证到达 ADR 域的 CPU 存储的持久性。此外，现代 CPU 采用复杂的无序执行，因此应明确且昂贵地对持久指令进行排序以保证一致性。当前的 intel ISA 提供刷新指令（clflush、clflushopt、clwb）以将缓存中的数据刷新到持久内存，以及用于绕过缓存的 ntstore 和栅栏指令，以确保以正确的顺序持久保存以前的存储。

3　研究进展

有四种关键技术来利用 PM 来应对挑战，同时实现高性能（如图 3 所示）。这包括半持久内存表（第 4.1 节）、带有 ChainLog 的重新排序环（第 4.2 节）以启用无日志事务、全局索引（第 4.3 节）以在 PM 中维护一个大的全局排序持久级别作为 L_0 LSM-tree，以及专门设计的名为 Halloc(第 4.4 节)

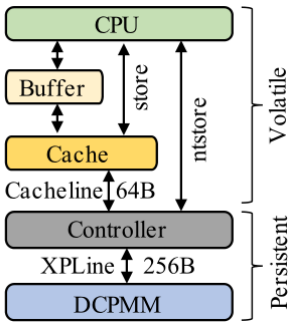

图 2 数据路径

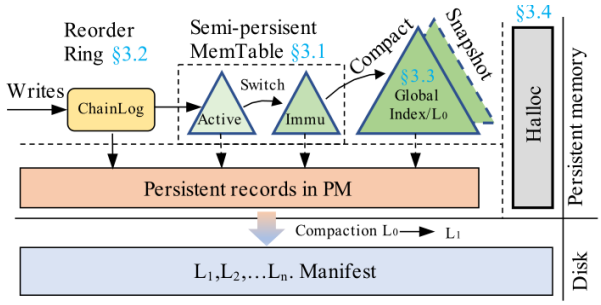

图 3 系统架构，其中所有持久内存都由 Halloc 管理

的 PM 分配器。其他级别 $(L_1, \cdots, L_n$ 在 SSD 中保持不变。请注意，我们仍然在应用程序运行时使用 DRAM 来进行块/行缓存和易失性索引。

如图 3 所示，更新的条目首先通过重新排序环进行批处理和排队，然后插入到活动的半持久内存表中。当活动的记忆表已满时，它会切换到不可变状态并通过轻量级内存压缩将其压缩为提议的 L_0。由于提议的 L_0 保证了 PM 中的持久性，因此不需要刷新到 SSD。当已满时，它的不可变快照会被创建并压缩到 SSD 中的 L_1 中，而不会阻止前台写入。查找通过探测半持久性内存表、提议的 L_0 和 SSD 中的其他级别来查找给定密钥的最新版本，并在目标密钥匹配时终止。

该设计为基于 LSM-tree 的存储引擎带来了三个显着的好处：（1）我们通过 PM 编程库避免了 WAL 和额外日志记录的开销，并通过重新排序环和半持久性记忆表实现快速恢复；（2）半持久性内存表和提议的保证了 PM 中的持久性，因此不需要刷新到 SSD。因此，我们可以显着减少写入 SSD 的数据量以及后台刷新开销；(3) 提议的是全局排序的，因此与用于 DRAM-SSD 存储架构的传统基于 LSM 树的引擎（例如 RocksDB 、LevelDB 和 X-Engine）。

3.1 半持久内存表

持久化索引的更新通常会覆盖多个单词，导致多次小的随机写入，这会导致写入放大和开销，以保证 PM 中索引节点的一致性。提出的半持久性内存表，它采用了两种优化来解决这个问题。

3.1.1 保持索引节点的易变性

将索引节点保持在记忆表中。该设计的灵感来自以下观察：基于行业 LSM 树的 OLTP 引擎由于两个原因不会保留非常大的单个内存表（通常为 64-256MB）。首先，发现云用户通常会购买每个只有 8-32GB 主内存的小型数据库实例，以可接受的成本满足他们的性能要求。其次，保持一个小的内存表可以减少每次刷新的数据大小，这有助于分摊 I/O 消耗。给定一个 256MB 的内存表，我们的实验表明，通过在 PM 中扫描来重建易失性索引节点需要不到 10 毫秒。恢复期间的这种性能足够快，因此可以容忍易变的索引节点。而且，不需要昂贵的持久范围索引，并且范围索引可以保持不稳定。采用具有乐观锁耦合（OLC）和基于时期的回收（EBR）的 ART 作为可变范围索引，因为 ART 具有良好的缓存局部性和快速前缀匹配。具体而言，具有一个版本值的 KV 对（例如，图 4 中的版本 9）在索引节点中直接存储为 8 字节指针。一个键的多个多版本值（例如，图 4 中版本为 1、2、3 和 5,8 的叶节点）存储在一个排序数组中，其指针附加到 ART 中的叶节点。PM 中的记录是通过将键和值与内置元数据封装在一起来组织的。键不会存储到索引节点中，因为索引是易失性的，并且是通过扫描 PM 中的键来重建的。

3.1.2 批处理以减少写入放大

随机写入在 PM 中作为单独的 256 字节访问进行本机处理，从而导致写入放大。为了避免这种情况，我们将小的写入批处理到一个大的 WriteBatch 中，并将其作为一个整体按顺序刷新到 PM。每个批次在发出一个栅栏后都被持久化。并且，记忆表中的这些批次在逻辑上链接在一起以实现正确的恢复。我们在此处导出批处理大小作为影响写入延迟和吞吐量之间权衡的调整旋钮。

3.2 重新排序环

与写入额外事务日志以保证 OLTP 工作负载中事务的原子性的传统方法相比，重新排序环 (ROR) 是一种基于并发环的事务日志无提交算法，用于 PM 中的持久事务处理。为了实现这个目标，ROR

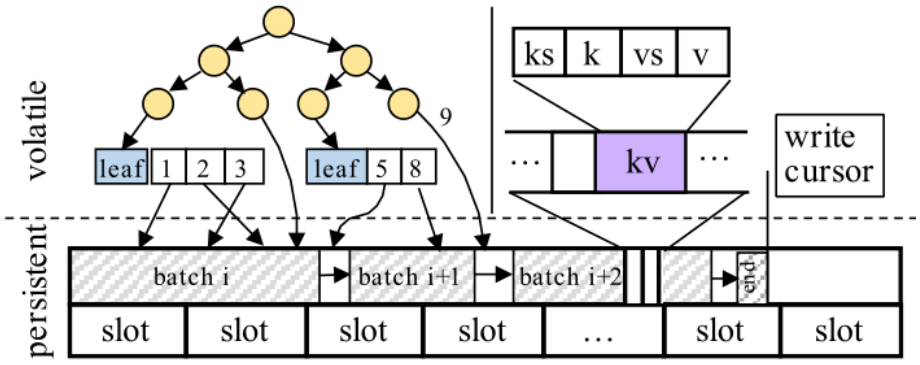

图 4　半持久记忆表的结构

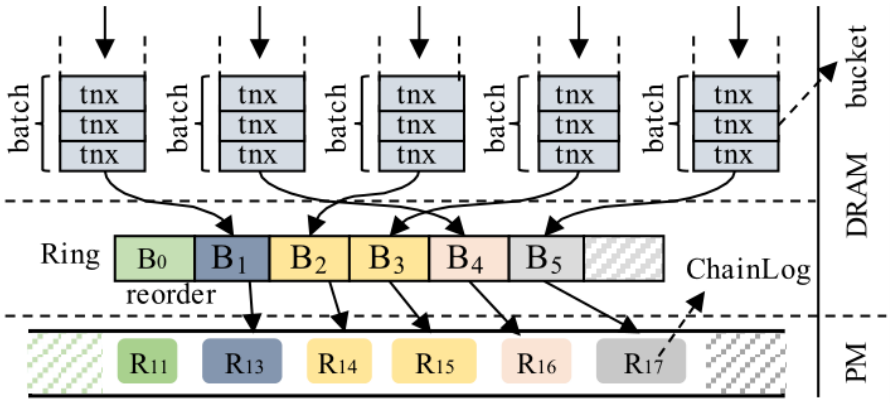

图 5　重新排序环的结构

采用了三个关键技术：ChainLog、批处理和并发环。如图 5 所示，ChainLog 保证无日志原子多字写入 PM。批处理用于将小的事务缓冲区合并为大的事务缓冲区，以避免对 PM 进行小的随机写入。并且基于数组的并发环通过对 ChainLog 项重新排序来实现并发持久性，以提高多核可扩展性。

3.3　全局索引

全局索引 (GI) 是一种索引数据结构，用于在 PM 中为 LSM 树维护全局排序的。为 DRAM-SSD 存储设计的现代基于 LSM 树的 OLTP 引擎通常采用 unordered 以增加读取放大为代价来减少压缩开销并缓解写停顿问题。多亏了字节可寻址的持久内存，我们通过重新设计 PM 中的 L_0 数据结构来解决这个问题。

在我们的实现中，GI 使用与半持久性内存表相同的易失性索引，其中 KV 对持久化在 PM 中。写入记忆表 s 的记录首先通过键粒度内存压缩合并到 GI 中。从记忆表 s 合并的所有记录只涉及指针操作，因此内存压缩不会发生数据复制。当 GI 的内存大小超过空间限制时，会创建它的快照并与 L_1 合并到 SSD 中。然后它被回收，所有 KV 对和 GI 也被删除。请注意，GI 可以使用任何范围易失性索引，因为索引节点在启动时通过扫描数据记录重建。

3.3.1　内存压缩

从记忆表 s 到 GI 的合并操作是通过内存压缩来执行的。与半持久内存表类似，GI 中的键存储在叶节点中，所有多版本值存储在附加到叶节点的排序数组中。在执行内存压缩时，如果键不存在，首先将其插入到 GI 中。然后从 GI 中清除属于该键的陈旧值，同时将新值插入到叶子的排序数组中。由于所有记录都由 Halloc 管理，因此 PM 中的 KV 对的密钥粒度内存释放是不允许的。只有在 GI 快照的压缩完成后才会释放内存。

3.3.2　快照

我们为 GI 设计 snapshot 是为了保证 GI 到 SSD 的 compaction 时 GI 仍然是可写的，这样记忆表 s 到 GI 的合并不会被阻塞。在 GI 中，快照是通过冻结当前 GI 并创建一个新 GI 来实现的。冻结的 GI 中的所有记录都由 Halloc 管理，只有在快照完全压

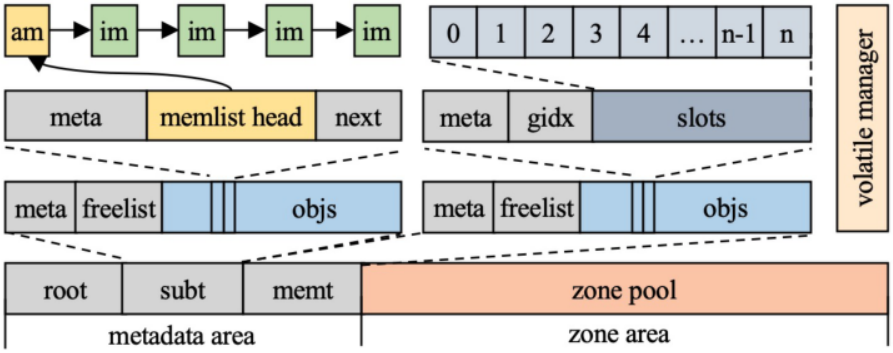

图 6 Halloc 的结构

缩时才会回收，在此期间不允许创建新的 GI。该设计带来了写入的改进，同时由于读取可能会跨越两个索引而导致更多的读取开销。此外，Halloc 通过在牺牲内存管理效率的同时采用批量回收策略来提高内存分配器的性能。可以通过指定支持细粒度内存分配的新 PM 分配器来实现键粒度快照。但是，这种设计会导致更多的持久内存管理成本。我们将具有键粒度快照的高效持久全局索引的提议推迟到未来的工作中。

3.3.3 PM⟶SSD 压缩

由于 GI 是全局排序的，并且在执行 compaction 时 GI 的 snapshot 是不可变的，compaction 不会阻塞其他 GI 的写操作。此外，即使对于 0，GI 的范围也可以方便地拆分以执行并行压缩。

3.3.4 一致性

PM 到 SSD 的 compaction 涉及到数据库状态的变化，应该避免系统崩溃导致的不一致。我们通过在 SSD 中维护清单日志来记录数据库状态来解决这个问题，因为它不在写入的关键路径中。GI 的快照管理来自多个内存表的记录。记忆表 s 的元数据保存在 Halloc 中，直到它完全压缩到 SSD 中。当系统在从 PM 压缩到 SSD 的过程中崩溃时，PM 中的数据记录会通过重放清单日志来删除。PM 中记录的索引节点在启动时重建。

3.4 Halloc

Halloc 通过三个关键设计解决了通用 PM 分配器的缺点：基于池的对象保存、应用程序感知内存管理和统一内存分配。许多通用持久 PM 分配器同时考虑随机分配和解除分配，这会导致非常分散的内存分配和昂贵的缓存行刷新和栅栏。此外，当使用当前的瞬态 PM 分配器时，它们会受到内存空间消耗的影响，这些分配器通常设计有额外的易失性 PM 池。Halloc 不强制执行内存映射的固定位置。对象池的所有对象都由内部 8 字节整数寻址，并且它们的实际内存地址在重新启动时从 DAX 映射文件重新映射。

3.4.1 自定义对象池

如图 6 所示，Halloc 维护了两种自定义的持久对象池，为 LSM-tree 提供元数据持久化：Subtable 池用于持久化列族对象和记忆表池，用于管理记忆表的 PM 内存。Subtable pool 中的一个 Subtable 对象包含一个类似于 log-free freelist 的 memlist 链接的记忆表对象列表，其中第一个记忆表是可变的，其他的都是不可变的，类似于 RocksDB 的规范。并且记忆表池是一个对象池，其中每个记忆表对象由固定数量的 slot 组成，每个 slot 管理一个内存区域，以实现特定于应用程序的 PM 管理。由于 LSM-tree 的记忆表遵循 append-only 内存分配，因此记忆表对象的 gidx 旨在反映分配位置。

3.4.2 区域池

区域池是 Halloc 中的内置池，允许应用程序管理自己的运行时内存。我们设计区域池是因为自定义对象池无法在运行时为可变大小和未知数量的对象执行内存分配。在 LSM-tree 中，KV 对通常具有不同的大小。因此，我们不能为所有 KV 对保留对象池。具体来说，区域池被记忆表用于持久内存管理和易变性管理器用于临时内存管理。为记忆表分配的区域对象遵循 append-only 和批量回收内存分配方案，类似于 SSD 的 ZNS 和 DRAM 的 MSLAB。为易变性管理器分配的区域对象更远。

基于区域池的 volatile 管理器设计的一个限制是易变性对象的大小不能超过区域对象的大小，因为基于池的区域设计保证了一个区域对象中只有一个连续的内存区域。但是，我们可以将大的 volatile 对象分成多个小对象，并为每个对象进行小分配。此外，如果可以预测大型易失性对象的内

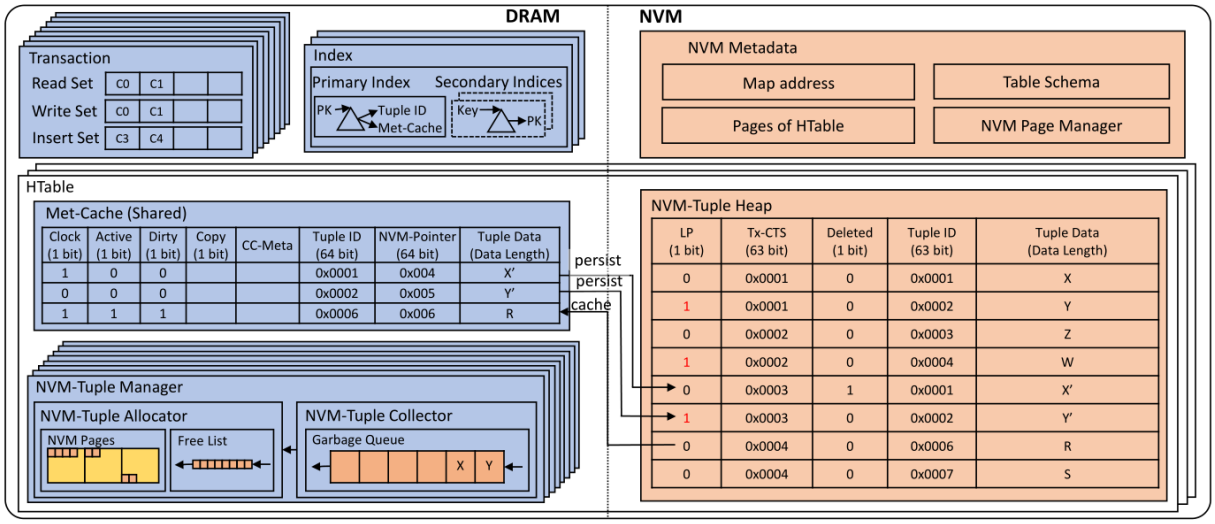

图 7　Zen 的架构

存占用，我们可以将这些对象专门保存在我们的对象池中以用于易失性目的。

3.5　Zen 架构

图 7 概述了 Zen 的架构。每个基表都有一个混合表 (HTable)。它由 NVM 中的元组堆、DRAM 中的 Met-Cache 和每线程 NVM 元组管理器组成。此外，Zen 将表模式和粗粒度分配结构存储在 NVM 元数据中。此外，Zen 将索引和交易私有数据保存在 DRAM 中。

NVM-元组堆。NVM 元组是 NVM 中的持久元组。Zen 将基表中的所有元组作为 NVM 元组存储在 NVMtuple 堆中。堆由固定大小（例如 2MB）的页面组成。每个页面包含固定数量的 NVM-Tuple 插槽 1。一个 NVM 元组由一个 16B 的头和元组数据组成。

NVM 元组堆可能包含多个版本的逻辑元组。元组 ID 和事务提交时间戳 (Tx-CTS) 唯一标识一个元组版本。删除的位显示逻辑元组是否已被删除。最后一个持久化 (LP) 位显示该元组是否是已提交事务中最后一个持久化的元组。LP 位在无日志事务中起着重要作用。请注意，标头不包含特定于特定并发控制方法的字段。NVM 元组插槽与 16B 边界对齐，因此 NVMtuple 标头始终驻留在单个 64B 缓存行中。通过这种方式，使用一个 clwb 指令后跟一个 sfence 来持久化 NVM 元组标头。

3.5.1　元缓存

Met-Cache 为相应的 NVM 元组堆管理 DRAM 中的元组粒度缓存。一个 Met-Cache 条目包含元组数据和七个元数据字段：一个指向 NVM 元组（如果存在）的指针、元组 ID、一个脏位、一个表示该条目可能被活动事务使用的活动位、一个用于支持缓存替换算法的时钟位，用于指示条目是否已被复制的复制位，以及包含特定于正在使用的并发控制方法的附加每元组元数据的 CC-Meta 字段。Zen 支持广泛的并发控制方法（。Zen 使用 Met-Cache 完全在 DRAM 中执行并发控制。

3.5.2　DRAM 中的指数

为 DRAM 中的每个 HTable 维护索引。我们在崩溃恢复时重建索引。一级索引是必需的，二级索引是可选的。对于主索引，索引键是元组的主键。该值指向 (i) Met-Cache 或 (ii) NVMtuple 堆中元组的最新版本。我们使用值的未使用位来区分两种情况 2。对于二级索引，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条目（之前被其他事务修改过）。

3.5.3　交易私有数据

Zen 支持多线程并发处理事务。每个线程为 DRAM 中的事务私有数据保留一个线程本地空间。它记录事务的读、写和插入活动。OCC 和 MVCC 变体将读取、写入和插入集作为单独的数据结构进行维护。2PL 变体以日志条目的形式存储更改。

3.5.4　NVM 空间管理

Zen 使用两级方案来管理 NVM 空间。首先，NVM 页面管理器执行页面级空间管理。它分配和管理 2MB 大小的 NVM 页面。NVM 元数据中的映射地址和 HTable 页面维护从 NVM 页面到 HTable

的映射。其次，NVM 元组管理器执行元组级别的空间管理。每个线程为线程访问的每个 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 中维护一个不相交的空闲 NVM 元组槽子集。空闲槽有两种：新分配页面中的空槽或垃圾回收槽。我们在系统设置时用全 0 初始化 NVM，并使用 Tx-CTS=0 表示空槽。垃圾收集器收集陈旧的 NVM 元组并将它们放入空闲列表中。同一个 HTable 的所有收集器协同工作以回收 NVM-Tuples。

3.6 轻量级 NVM 空间管理

我们的两级 NVM 空间管理设计几乎不会产生 NVM 持久开销。首先，只有页面级管理器持久化元数据。由于我们分配了 2MB NVM 页面，因此在 NVM 中记录页面分配和页面到 HTable 映射的持久化操作并不频繁。其次，元组级管理器完全在 DRAM 中执行垃圾收集和 NVM 元组分配，而无需在正常处理期间访问 NVM。这是可行的，因为写入提交的元组是为了将 NVM 元组槽标记为已占用。我们不需要在 NVM 中记录单独的元组元数据以进行元组分配。在崩溃恢复期间，Zen 扫描 NVM 元组堆，并能够通过检查其标头来确定每个 NVM 元组槽的状态，如第 2.3.3 节所述。因此，Zen 完全消除了元组级 NVM 分配的 NVM 写入和持久操作的成本。

4 总结与展望

在这项工作中，我们研究了如何利用 PM 重新访问基于传统 LSM 树的 OLTP 存储引擎。具体来说，我们 (1) 提出了一种名为 Halloc 的轻量级 PM 分配器，针对 LSM 树进行了优化，(2) 利用 PM 的持久内存写入设计了一个高性能的半持久性记忆表，(3) 设计了并发重新排序用于实现 OLTP 工作负载的无日志事务的环算法和 (4) 将全局索引呈现为具有非阻塞内存压缩的新全局排序持久级别。我们的评估表明，这些关键设计可以释放 PM-SSD 存储架构的力量，显着提高基于 LSM 树的 OLTP 存储引擎的性能。Zen 是一种用于 NVM 的高吞吐量无日志 OLTP 引擎。Zen 采用了三种新技术来减少 NVM 开销，即 Met-Cache、无日志持久事务和轻量级 NVM 空间管理。实验结果表明，对于 YCSB 和 TPCC 基准测试，Zen 在具有 NVM 容量、WBL 和 FOEDUS 的 MMDB 上实现了高达 10.1 倍的改进。总之，我们相信 Zen 是支持 NVM 内存中 OLTP 事务的可行解决方案。

这些技术大多相互正交，结合我们的见解，我们希望它们可以作为设计未来 PM 数据结构的构建块。最后，我们量化了源自持久性 CPU 缓存和类似 DRAM 的 PM 性能的性能提升。我们发现性能主要取决于 PM 的延迟和带宽特性，相比之下，缓存行刷新导致的减速相对有限。

参考文献

[1] YanBaoyue, ChengXuntao, JiangBo, et al. Revisiting the design of LSM-tree Based OLTP storage engine with persistent memory[J]. Proceedings of the VLDB Endowment, 2021.

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

[3] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. 1567–1581.