Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory

Gang Liu, Leying Chen, Shimin Chen

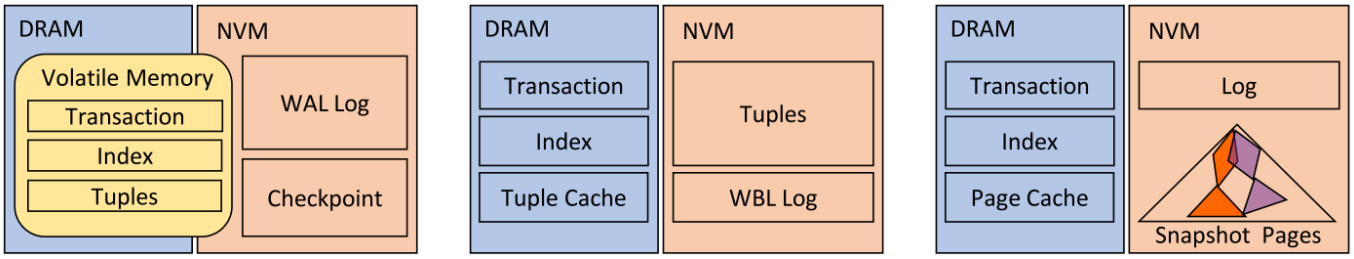
摘要

新兴的非易失性内存 (NVM) 技术，如 3DXPoint，为 OLTP 数据库提供了巨大的性能潜力。然而，事务性数据库需要重新设计，因为非易失性存储比 DRAM 慢数量级且只支持面向阻塞访问的关键假设已经改变。NVM 是字节可寻址的，几乎和 DRAM 一样快。NVM 的容量比 DRAM 大 4-16 倍。这样的 NVM 特性使得完全在 NVM 主存中构建 OLTP 数据库成为可能。本文研究了混合 NVM 和 DRAM 存储器的 OLTP 引擎的结构。为 NVM 设计 OLTP 引擎的三个挑战：元组元数据修改、NVM 写冗余和 NVM 空间管理。提出了一个用于 NVM 的高吞吐量无日志 OLTP 引擎 Zen。Zen 通过三种新颖的技术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。[1]

关键词：无日志；NVM；OLTP；Zen

1 引言

现有的 NVM 技术有类似的特点[2]：(i) NVM 像 DRAM 一样是可字节寻址的；(ii) NVM 比 DRAM 慢一点（2-3 倍），但比 HDD 和 SSD 快几个数量级；(iii) NVM 提供的存储空间可以比 DRAM 大得多（例如，在双插槽服务器中高达 6TB）；(iv) NVM 写入的带宽比读取的带宽低；(v) 为了确保断电后数据在 NVM 中的一致性，需要使用 cache line fush 和 memory fence 指令 (如 clwb 和 sfence) 的特殊持久性操作。将数据从易失性的 CPU 缓存持久化到 NVM，产生的开销明显高于正常的写入；(vi) NVM 单元在有限的次数（千万次）写入后可能会磨损。



(a) MMDB with NVM Capacity (b) WBL (c) FOEDUS

图 1: 基于 NVM 的 OLTP 引擎

如图 1（a）所示，当数据量大于 DRAM 时，MMDB 将部分 NVM 视为较慢的 DRAM。系统在 DRAM 中存储元组和索引，并使用正常的加载和存储指令在 DRAM 中处理事务。为了保证持久性。系统将写前日志（WAL）和检查点放在 NVM 中。崩溃后，DRAM 中的元祖和索引被视为丢失并基于 NVM 中的日志和检查点来进行恢复。这种设计有两个缺点：一个被修改的元组要同时写入 WAL 和检查点，会产生额外的 NVM 写入。如果它被存储在 NVM 上，该元组在 NVM 中被写入三次。随着数据规模的增加，越来越多的元组驻留在 NVM 中。由于元组的元数据经常被修改。这导致读取事务仍然执行大量的 NVM 写入。

如图 1(b) 所示，WBL[3] 在 DRAM 中维护索引和缓存。事务在混存中执行。WBL 在 NVM 中利用元数据（事务 ID，提交时间戳和对该元组上一版本的引用）支持逻辑元组的多个版本。WBL 日志不包含元组数据，日志在一组事务提交后写入。它包含一个持久化的提交时间戳 (cp)，和一个脏提交时间戳 (cd)。任何具有提交时间戳早于的事务一定已经成功地被持久化到 NVM。在崩溃恢复时，系统会检查最后一个日志条目，并撤销任何时间戳在 (cp , cd) 的事务。优点与 MMDB 相比，WBL 减少了日志大小，并且不维护检查点。大大减少了 NVM 的写入次数。缺点 WBL 在 NVM 中的每一个元组都维护每个元组的 metadata。因此，它受到频繁对元组元数据修改的影响。

如图 1(c) 所示，FOEDUS[4] 将元组存储在 NVM 的快照页中，并在 DRAM 中维护了一个页面缓存。DRAM 中的页面索引为一个页面保留了两个指针，一个指向 NVM 快照中的页面的指针，以及一个指向页面缓存中的页面的指针。FOEDUS 在 DRAM 中运行事务。如果包含事务所需元组的页面不在页面缓存中，系统会将该页面加载到页面缓存中并更新页面索引。在提交事务时，系统会写入 NVM 中的 redo 日志。一个后台日志收集器线程会定期收集日志，并运行类似于 map-reduce 的计算，在 NVM 中生成一个新的快照。优点 FOEDUS 完全在 DRAM 中处理事务，从而避免了在 NVM 中进行元数据的写入。缺点缓存的页粒度导致了 NVM 的读取放大。一个元组的读取会带来更大的页读取的开销。复杂的 map-reduce 计算导致许多 NVM 的写入。FOEDUS 的实现使用 I/O 接口来访问 NVM，没有充分利用可由字节编址的 NVM 的优势。

从 NVM 的数据结构和系统的研究中，本文得到了三个共同的设计原则。并将其应用到 OLTP 引擎的设计中。(i) 将经常访问的数据结构放在 DRAM 中（数据结构是瞬时的或者可以重建的）；(ii) 尽可能减少 NVM 的写入。(iii) 尽可能地减少持久性操作。

三个设计挑战：元组元数据的修改。在 MMDB 和 WBL 中，每个元组的元数据与元组一起存储在 NVM 中。一致性控制方法（OCC 变体和 MVCC 变体）可能会修改元数据，甚至对元组的读取。NVM 写冗余。在 MMDB 和 FOEDUS 中，一个被修改的元组被写入元组堆、日志、检查点和/或 NVM 中的页面快照。NVM 的写入放大会对事务性能产生负面影响。NVM 空间管理。WBL 为元组执行细粒度的空间分配。每次元数据持久化到 NVM 需要进行空间分配和释放（有日志记录）。这可能会产生大量的 NVM 持久化开销。

2 相关工作

论文提出了一个用于 NVM 的高吞吐量无日志 OLTP 引擎 ZEN。Zen 支持比 DRAM 大得多的 OLTP 数据库，同时解决了三个设计挑战，以实现事务处理和崩溃恢复的良好性能。图 2 概述了 Zen 的体系结构。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 和每个线程的 NVM 元组管理器组成。此外，Zen 在 NVM 元数据中存储表模式和粗粒度分配结构。此外，Zen 将索引和事务私有数据保存在 DRAM 中。

元组堆。NVM-Tuple 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。堆由固定大小（例如 2MB）的页组成。每个页面包含固定数量的 NVM 元组槽。NVM 元组由 16 位的报头和元组数据组成。NVM 元组堆可以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。请注意，标头不包含特定于特定并发控制方法的字段。NVM 元组时隙与 16 字节边界对齐，使得 NVM 元组报头始终驻留在单个 64 位高速缓存行中。通过这种方式，我们可以使用一个 CLWB 指令后跟一个 sfence 来持久化 NVM 元组头。

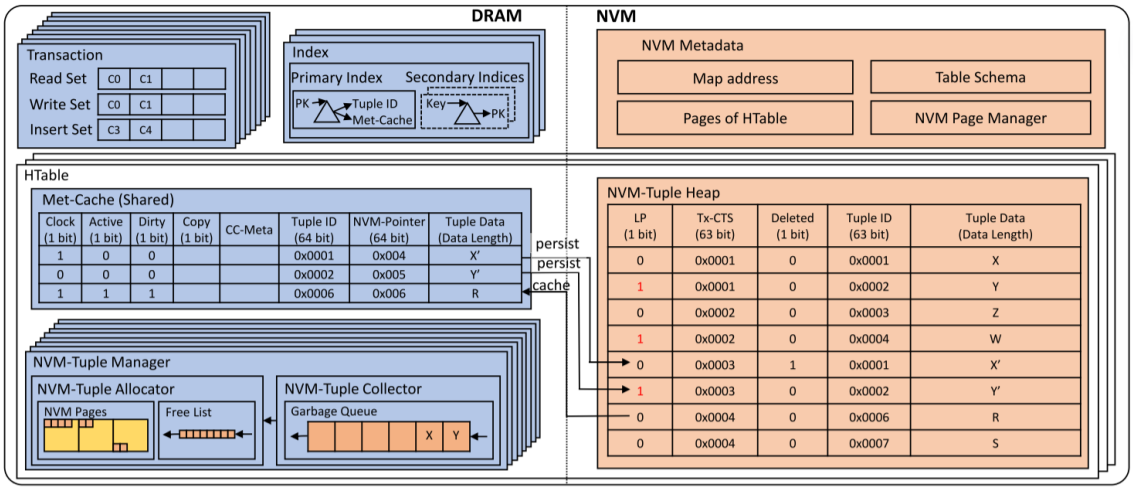


图 2: Zen 的架构

元缓存。MET-cache 为相应的 NVM 元组堆管理 DRAM 中的元组缓存。元缓存条目包含元组数据和七个元数据字段：指向 NVM 元组（如果存在）的指针、元组 ID、脏位、指示该条目可由活动事务使用的活动位、支持高速缓存替换算法的时钟位、指示该条目是否已被复制的复制位，以及包含特定于所使用的并发控制方法的额外每个元组元数据的并发控制字段。使用元缓存，Zen 完全在 DRAM 中执行并发控制。

DRAM 中的索引。我们为 DRAM 中的每个 HTABLE 维护索引。我们在崩溃恢复时重建指数。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) 元缓存或 (ii)NVM 元组堆中元组的最新版本。我们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条目（之前由其他事务修改）。

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本地线程空间。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 页面管理器执行页面级空间管理。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 中维护不相交的子集空 NVM 元组。有两种空闲插槽：新分配页中的空插槽或垃圾回收的插槽。我们在系统设置时使用 0 初始化所有 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 本文方法

3.1 元数据增强元组缓存

对于一个混合表，我们将其元缓存划分为多个大小相等的区域，每个事务处理线程一个区域。NVM 元组堆还被划分为每个线程的区域。线程负责管理其元缓存区域和 NVM 元组堆区域。它可以读取所有区域中的元组，但只能写入自己的区域。对于缓存命中，线程可以读取任何区域中的元缓存条目。如果线程想要修改另一个元缓存区域中的元组，它必须在修改它之前将该条目复制到自己的元缓存条目中。它设置原始元缓存条目的复制位。对于缓存丢失，线程可以将一个 NVM 元组带进它自己的元缓存区域。如果在元缓存区域中没有空条目，线程必须从其区域中挑选并驱逐一个受害元组，以便为缓存丢失的 NVM 元组腾出空间。这种设计消除了管理元缓存条目的线程争用，并支持将 DRAM 和 NVM 地址范围绑定到特定处理器内核。我们采用时钟算法来替换元缓存。该算法选择活动位和时钟位都为 0 的第一个遇到的条目作为受害者。如果设置为活动状态，则该条目正由活动事务访问。算法跳过这样一个条目，这样它就不会替换其他正在运行的事务使用的元缓存条目。如果设置了时钟，则该条目最近被使用过。我们希望在缓存中保留这样的条目。使用原子比较交换指令修改活动位和时钟位。

Zen 在 NVM 元组中没有与并发控制方法相关的每元组元数据。当从 NVM 获取 NVM 元组到元缓存时，元缓存条目中的 CC-Meta 对其进行了增强。CC-META 包含特定于正在使用的并发控制方法的每元组元数据。之后，Zen 可以完全在 DRAM 中运行并发控制方法，因为活动事务访问的所有元组都在元缓存中。这种设计有以下好处。(i) 它将细粒度的每元组元数据读写从 NVM 转移到 DRAM。因此，Zen 享受快速的每元组元数据访问。(ii) 元组读取永远不会导致 NVM 元组上的 NVM 写入。(iii) 中止的事务不会引起 NVM 元组写开销。(iv) 内存并发控制减少事务在关键代码区域中花费的时间，无论是获取关键资源还是执行一致性验证。因此，可以降低总体事务中止率。

3.2 无日志持久事务

执行：事务在开始时获得时间戳。对于事务请求的每个元组，事务在主索引中查找其位置。如果元组在 NVM 中，事务通过缓存替换算法在元缓存中找到（受害者）条目，通过读取所请求的 NVM 元组并使用特定于所使用的并发控制方法的每个元组 CC-Meta 增强它来构建元缓存条目，并用元缓存条目位置更新索引。请注意，出于以下原因，Zen 不需要将受害者条目写入 NVM。首先，如果条目只被以前的事务读取，那么它不会被更改，可以被丢弃。其次，如果条目是由以前提交的事务生成/修改的，那么它必须已经在提交时保存到 NVM。第三，如果条目是由中止的事务修改的，则它是无效的，应该被丢弃。Zen 在元缓存的帮助下完全在 DRAM 中运行并发控制。如果没有冲突并且事务可以提交，则 Zen 将事务移到持久处理中。如果事务必须中止，ZEN 检查事务访问的元缓存条目是否是脏的。对于脏条目，Zen 从 NVM 元组指针指向的 NVM 元组恢复条目，以便事务的重试将在元缓存中找到该条目。

持久化：Zen 将生成和修改的事务元组保存到 NVM，没有日志。挑战是在不写重做日志记录和提交日志记录的情况下持久化多个元组。我们解决方案的基本思路如下。首先，我们将一个元组保存到一个空闲的 NVM 元组槽中。这样，元组的前一个版本在持久化处理期间是完好无损的。Zen 可以在崩溃的情况下回到以前的版本。这个想法已经在 WBL 中被证明是成功的。其次，我们使用 NVM 原

子写将最后一个元组的 LP 位标记为在事务中持久存在。我们确保在保留 LP 位之前保留所有元组。这样，LP 位就起到了与提交日志记录相同的作用。在恢复期间，如果 LP 位存在，则事务已提交。事务生成/修改的所有元组必须已成功保存到 NVM。否则，崩溃发生在持久化事务的中途。因此，Zen 丢弃事务写入的任何 NVM 元组。

维护：为了减少争用，每个线程都有其专用的 NVM 元组分配器和垃圾队列。当线程发现存在一个较新的版本时，它将垃圾收集一个 NVM 元组版本。垃圾收集决策是在两种情况下做出的。首先，当提交覆盖元组的事务时，线程垃圾收集旧的 NVM 元组版本，除非元缓存条目是从另一个区域复制的。其次，在它从其元缓存区域中驱逐一个条目之前，线程垃圾收集要确定元组的复制位是否被设置。

3.3 轻量级 NVM 空间管理

我们的两级 NVM 空间管理设计几乎没有 NVM 持久开销。首先，只有页面级管理器保存元数据。由于我们分配了 2MB 的 NVM 页面，所以在 NVM 中记录页面分配和页面到混合表映射的持久操作很少。第二，元组级管理器完全在 DRAM 中执行垃圾收集和 NVM 元组分配，在正常处理过程中不访问 NVM。这是可行的，因为写入提交元组的目的是将 NVM 元组槽标记为已占用。我们不需要在 NVM 中为元组分配记录单独的每个元组元数据。在崩溃恢复期间，ZEN 扫描 NVM 元组堆，并能够通过检查每个 NVM 元组插槽的头部来确定其状态。因此，ZEN 完全消除了元组级 NVM 分配的 NVM 写入和持久化操作的成本。为了减少线程争用，我们设计了分散的 NVM 元组管理器。每个线程管理自己的 NVM 元组堆区域。它从其区域中分配 NVM 元组时隙。它收集垃圾并释放其区域中的 NVM 元组插槽。当空闲列表为空，并且存在元组分配请求时，NVM 元组管理器要求 NVM 页面管理器分配一个新的 2MB 的 NVM 页面。它将新分配的页面划分为空槽，并将它们放入空闲列表中。空插槽事务提交时间戳为 0，因为 NVM 空间在设置时初始化为 0。两个实现细节有助于减少垃圾收集对单个事务延迟的影响。(i) 在 DRAM 中实现了每个线程的垃圾队列和空闲列表，没有任何 NVM 开销。垃圾回收没有线程争用。(ii) 除非 NVM 空间用完，否则我们限制每个事务在垃圾队列中扫描的项数。这限制了垃圾队列扫描对单个事务延迟的影响。此外，Zen 将一个元组保存到不同于 NVM 中以前版本的位置。这有助于热元组的磨损均衡，因为 Zen 减少了 NVM 中的热点写入。

4 复现细节

4.1 与已有开源代码对比

参考了 https://github.com/efficient/cicada-exp-sigmod2017 和 https://github.com/liugang869/zen-code，基于前者进行修改，并参考后者的实现细节。复现了无日志的 OLTP 引擎，并与后写日志 WBL 引擎进行了实验对比。由于 cicada 引擎是一个内存数据库引擎，所以在复现的时候通过调用 libpmem.h 库来使用持久内存，并将数据保存到持久内存。核心的代码增量为 500 多行，其中包括对无日志的实现，元缓存的实现，将数据写入持久内存实现持久化以及构建其他适应持久内存的类和函数。

4.2 实验环境搭建

首先，需要安装 DBx1000，silo 和 cicada 的环境依赖，然后安装 pmdk，再分别编译 cicada，silo，DBx1000，最后通过 DBx1000 当中的 rundb 就可运行起来了。

4.3 界面分析与使用说明

```
ls@server:~/zen/zen-cicada/cicada-engine/build$ cmake .. make -j
```

图 3: 编译 cicada

```
ls@server:~/zen/zen-cicada/silo$ make -j
```

图 4: 编译 silo

```
ls@server:~/zen/zen-cicada/DBx1000$ make -j
```

图 5: 编译 DBx1000

```
ls@server:~/zen/zen-cicada/DBx1000$ ./rundb
```

图 6: 运行 DBx1000

```
page_used:                    853 pages
memory_size:                    1.789 GB
aep_page_used:              11196 pages
aep_memory_size:             23.480 GB
aep_clwb_cnt:               117.585 M
aep_sfence_cnt:              19.192 M
aep_persist_size:             5.660 GB
elapsed:                      1.776 sec

transactions:               4690671 (  2.641 M/sec)
committed:                  1409146 (  0.793 M/sec;  30.04%
aborted: get_row:                 0 (  0.000 M/sec;   0.00%
aborted: pre_validation:          0 (  0.000 M/sec;   0.00%
aborted: d_version_insert:        0 (  0.000 M/sec;   0.00%
aborted: main_validation:   3281525 (  1.848 M/sec;  69.96%
aborted: logging:                 0 (  0.000 M/sec;   0.00%
aborted: application:             0 (  0.000 M/sec;   0.00%
```

图 7: 部分运行结果

如上图进入到相应文件夹进行编译，最后再运行 DBx1000 即可得到实验结果

5 实验结果分析

图 8 和图 9 的实验结果基本上和论文当中差不多，不过也存在部分差异。在实验的过程中存在 zen 的性能不如 wbl 的情况，考虑主要是实验环境的差异，服务器的负载等情况对结果产生的影响，所以在图中保留了部分不同于论文的实验结果。总的来说，不论是在 clwb 和 sfence 的数量上，还是在吞吐量上，zen 的性能都比 wbl 好。

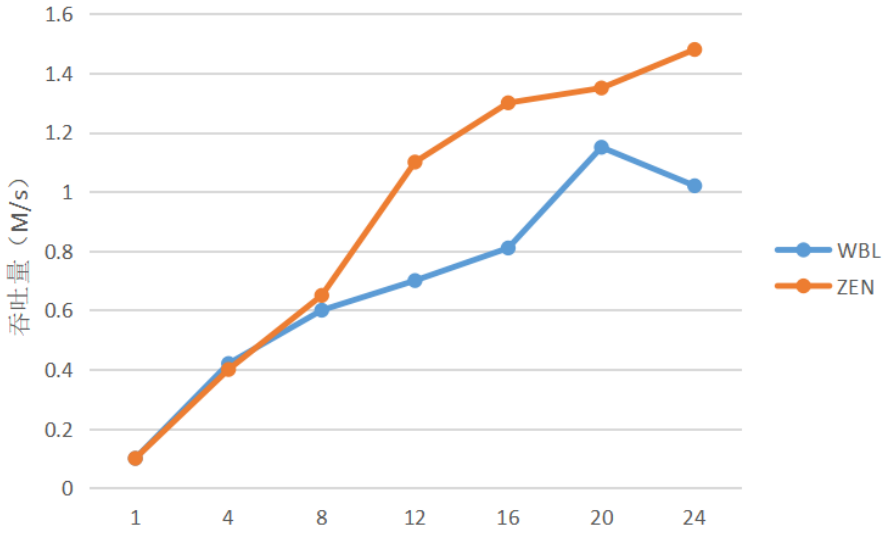
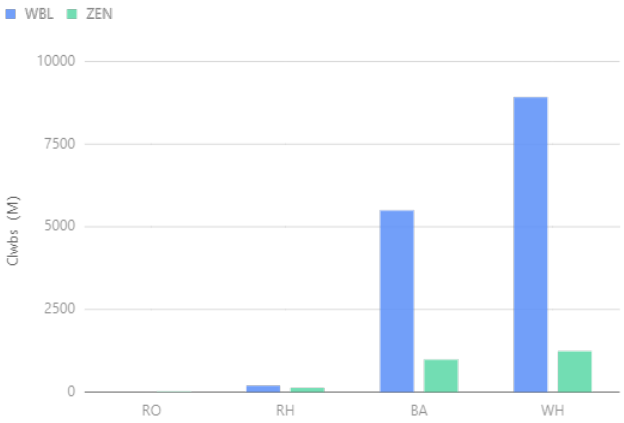
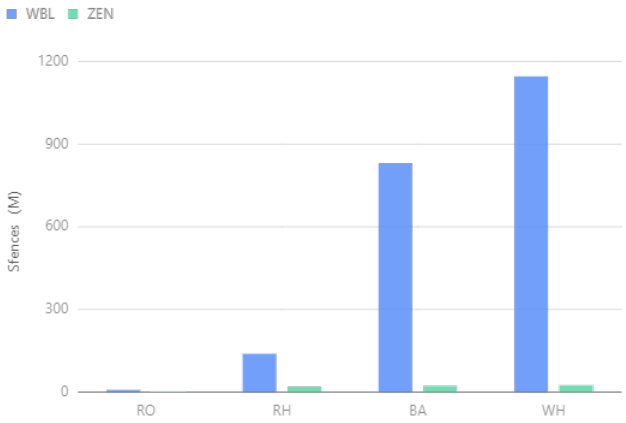


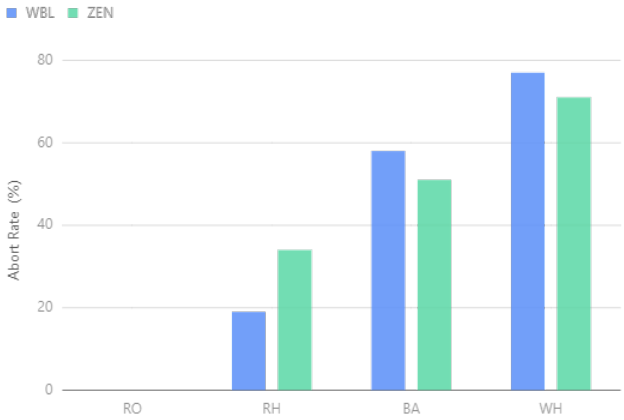
图 8: 不同线程对应的吞吐量



(a) Clwb counts



(b) Sfence counts



(c) Abort rate

图 9: 高倾斜下，NVM:DRAM=4，线程数为 16 的实验结果统计

6 总结与展望

文档的前半部分是对论文进行概况，介绍了论文面临的挑战，相关工作和实现方法。其中的相关工作介绍了系统的组成元素，有元缓存，索引，NVM 空间管理，元组堆。实现方法介绍了元数据增强元组，如何通过无日志实现持久事务以及对 NVM 进行轻量级空间管理分别用来解决前言里面提到的挑战。因为是基于已有的存储引擎进行的修改，原来的存储引擎是一个内存数据库，并没有利用到持久内存，所以在代码中对相应的功能进行了适当的更改。不足之处在于由于代码量过于庞大，所以

参考了很多 zen 的源码。未来的研究方向考虑不使用 DRAM，仅仅利用持久内存来实现数据库，这样可以保持一个强一致性，不过还得考虑对处理数据的速率进行优化，毕竟持久内存的处理速度还是和 DRAM 差了比较多。同时还可以参考本篇论文，采用无日志或者轻量级的日志。

参考文献

[1] 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.

[2] Intel Optane DC Persistent Memory Architecture and Technology.https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html[J]., 2019.

[3] ARULRAJ J, PERRON M, PAVLO A. Write-behind logging[J]. Proceedings of the VLDB Endowment, 2016, 10(4): 337-348.

[4] KIMURA H. FOEDUS: OLTP engine for a thousand cores and NVRAM[C]/ /Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015: 691-706.