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

摘要

随着新兴的非易失性内存 (NVM) 技术的出现, 使用 NVM 作为主要存储来显著提高需要持久性的系统 (如 OLTP 数据库) 的性能成为一个有前景的设计选择。但是传统的事务数据库需要重新设计, 因为许多关键假设发生了改变。NVM 的三大特征 (接近 DRAM 速度、大容量、非易失性) 使构建完全基于 NVM 主内存的 OLTP 数据库成为可能。本报告基于论文《Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory》来学习 NVM 和 DRAM 混合内存的 OLTP 引擎结构。可以观察到为 NVM 设计 OLTP 引擎存在三个挑战: 元组元数据修改、NVM 写冗余和 NVM 空间管理。Zen 是一种用于 NVM 环境下高吞吐量的无日志 OLTP 引擎。Zen 通过三种新技术解决了三个设计挑战: 元数据增强的元组缓存、无日志持久化事务和轻量级 NVM 空间管理。实验结果显示, 与现有解决方案相比,Zen 在运行 YCSB 和 TPCC 基准测试时分别获得了高达 10.1 倍的性能改进, 并且几乎实现了即时恢复。同时, 实验室的 NVM 硬件进行复现实验也一定程度验证了论文中的结论。

关键词: 数据库；非易失性存储器；OLTP

1 引言

1986 年东芝公司舛冈富士雄博士发明了 NAND Flash, 从而使得日本半导体业蓬勃发展 [18]。此后的 30 年，使用 NAND 技术的 SSD 不断发展，到 2020 年其销量已经超过传统移动硬盘（HDD）的销量。每次硬件的巨大发展都会对整个计算机业界带来惊天动地的变化。所以如图 1非易失性内存 (NVM) 技术一经出现, 构建基于 NVM 的数据库系统立即就成为一个热点方向。NVM 兼具近似 DRAM 的速度、超大容量和数据持久性的优点, 使用 NVM 作为 OLTP 数据库的主要存储可以大幅提高性能。但是 NVM 仍存在一些因其特性带来的问题, 其中写性能较差且持久化操作开销大, 让传统 DBMS 设计面临挑战。目前 NVM 数据库研究主要集中在存储管理与空间管理方面, 对于事务处理部分的优 jj 化研究还不足。而事务吞吐量直接关联到 OLTP 应用的整体性。因此, 设计适用于 NVM 的高性能事务处理引擎迫在眉睫。该论文针对 NVM 的特性提出了元数据增强缓存、无日志持久化事务和轻量级空间管理等技术, 大幅减少了元数据更新、写放大等对 NVM 性能的影响。该研究对指导 NVM 数据库事务子系统的优化设计具有重要意义。

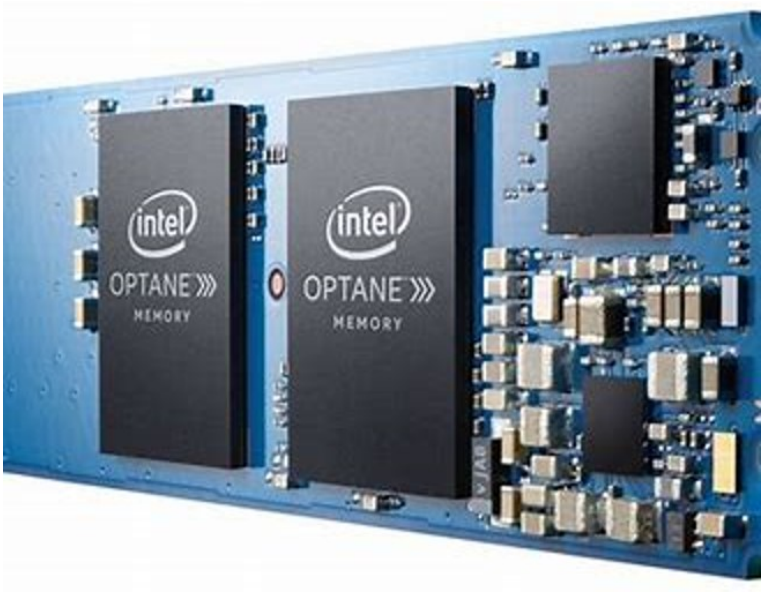

图 1. 英特尔公司非易失性存储产品 Optane

2 相关工作

2.1 非易失性存储特性和先前研究

非易失性存储器是一类存储其的总称，目前常见的非易失性存储技术包括 PCM [15]、STT-RAM [21]、Memristor [1] 和 3DXPoint [6]。它们具有的相似特性可以总结为 6 点：

1. 非易失性存储器像内存一样是可字节寻址的 [5];
2. 非易失性存储器比内存稍慢 2-3 倍，但比机械硬盘和固态硬盘快几个数量级，在图 2计算机存储层次结构中从上到下的第 3 与 4 层之间;
3. 非易失性主存储器的容量可以比主存大得多;
4. 非易失性存储器写入的带宽低于读取 [11];
5. 为了确保数据在电源故障时在非易失性存储器中是一致的，需要使用 cache line 刷新和内存栅栏指令 (如 clwb [14] 和 sfence [20]) 的特殊持久操作需要将数据从 CPU 缓存持久化到非易失性存储器，导致开销明显高于正常写入;
6. 与 NAND 的固态硬盘相同，非易失性存储器单元可能在有限数量的写入之后磨损 [22]。

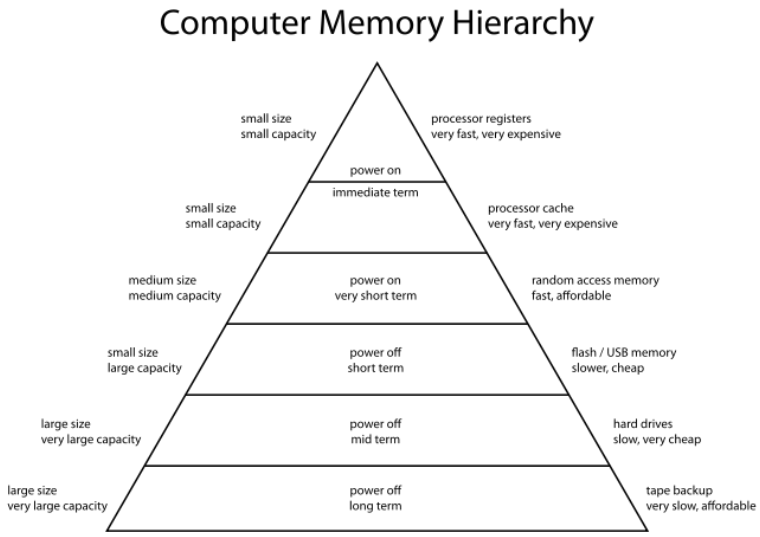

图 2. 计算机存储层次结构图

2.2 内存 OLTP 数据库

内存 OLTP 系统是为非易失性存储器设计 OLTP 引擎的起点，其考虑并发控制和崩溃恢复机制来实现 ACID 事务支持。主流内存 OLTP 设计没有使用传统面向磁盘数据库的两阶段锁 (2PL)，而是利用乐观并发控制 (OCC) 和多版本并发控制 (MVCC) 来获得更高的性能。Silo [16] 通过基于 epoch 的批量时间戳生成并分组事务的提交从而增强乐观并发控制。MOCC [17] 也是一种基于乐观并发控制的方法，它利用锁定机制来处理被频繁访问元组的高冲突问题。Tictoc [23] 消除了乐观并发控制中集中式时间戳分配的瓶颈，并在事务提交时延迟计算事务时间戳。Hekaton [4] 采用 latch-free 数据结构和多版本控制来处理内存中的事务。Hyper [12] 通过执行就地更新并将前映像增量存储在 undo 缓冲区中，改进了列存储中读取密集型事务的多版本控制。Cicada [10] 通过多个松弛同步时钟来生成时间戳并将其尽力内联以减少缓存未命中的情况以及优化多版本验证，从而减少了多版本控制的开销和争用。上述方法的一个共同特征是，它们使用元数据扩展每个元组或元组的每个版本。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，内存数据库（MMDB）将日志和检查点存储在传统的持久二级存储介质上，以实现持久性。主要区别在所有数据都保存在内存中，因此只有提交状态的事务和重做日志需要写入磁盘。当数据库崩溃时，主存数据库通过将最新的检查点从那些持久二级存储加载到内存中来恢复，然后读取并应用重做日志直至崩溃点。

2.3 现有针对非易失性存储介质的 OLTP 引擎设计

目前新型非易失性存储器并没有完全代替传统内存而是让计算机系统同时包含非易失性存储器和传统内存，二者被映射到软件虚拟内存中的不同地址范围。如图 3(a) 所示，当 OLTP 数据库大于内存容量时，主存数据库可以通过将非易失性存储器的一部分视为较慢的易失性内存使用，即仅仅利用了非易失性存储的巨大容量，并没发挥其硬件特性。与现有的内存数据库设计一样，系统将元组和索引存储在传统内存中，并使用正常负载和存储指令完全处理传统内存中的事务。而系统预写日志 (WAL) 和 checkpoint 保存在非易失性存储器中。它发

出特殊的持久性指令来持久化日志条目和 checkpoint。崩溃后，传统内存中的元组和索引就会完全丢失，系统会恢复 NVM 中的日志和检查点。

如图 3(b) 所示，后写日志 (WBL) [2] 维护传统内存中的索引和元组缓存。元组被提取到元组缓存中以进行事务处理。后写日志通过使用元组元数据来支持非易失性存储中逻辑元组的多个版本。提交事务通过在非易失性存储器中创建元组的新版本，在元组缓存中持久化修改后的元组。通过这种方式，如果崩溃发生在提交时间，则元组的先前版本可用。与预写日志不同，后写日志不包含修改后的元组数据。日志条目是在一组事务提交之后编写的。所以需要包含一个持久提交时间戳和一个脏提交时间戳。由于持久操作发出内存栅栏指令，因此该日志条目的存在表明任何提交时间戳早于的事务都必须成功持久化到非易失性存储器中。在崩溃恢复后时，系统会检查最后一个日志条目并撤销两个时间戳中的每个事务。

如图 3(c) 所示，FOEDUS [8] 将元组数据存储在非易失性存储器的快照（snapshot）页面中，并在传统内存中进行缓存。传统内存中的页面索引维护页面的双重指针，并在传统内存中运行事务。如果包含事务所需的元组的页面不在页面缓存中，系统将页面加载到页面缓存中并更新页面索引。在提交时，系统写入非易失性存储器中的 redo 日志。系统还会额外产生一个后台日志线程定期收集日志并运行类似 map-reduce 的计算来生成非易失性存储器中的新快照。FEDUS 完全处理传统内存中的事务，从而避免非易失性内存中的元组元数据写入。

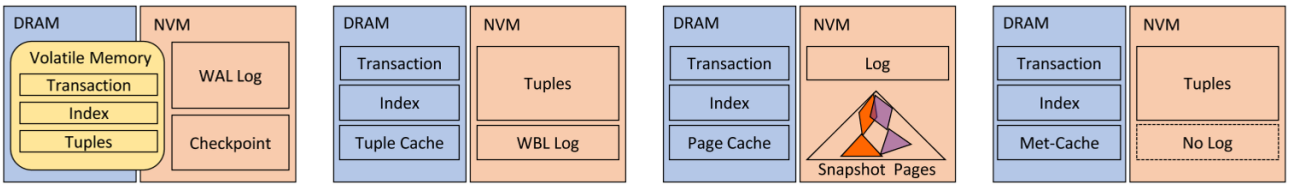

图 3. 几种现有针对非易失性存储介质的 OLTP 引擎设计

3 本文方法

从整体来看，Zen 遵循着两点最主要的的设计原则。如果传统中频繁访问的数据结构要么是瞬态的，也可以在恢复时重构。以及尽可能地减少 NVM 写入和持久性操作。图 3(d) 所示，Zen 在传统内存中维护元数据增强元组缓存 (Met-Cache)。与上述的 FFOEDUS 页面缓存不同，Met-Cache 缩小了缓存的粒度，让缓存的数据更加精确且高效。并且在并发控制时，仅需要在元组缓存中修改每个元组的元数据即可。其次，Zen 设计成了无日志形式，从而消除了非易失性存储的写放大问题。最后，Zen 提出了一种轻量级的空间分配设计，使其没有非易失性存储器的持久操作用于元组分配和空闲。

深入内存细节如图 4所示，每个基本表都被称为混合表（HTable）。它由一个非易失性存储器中的元组堆、传统内存中的元素缓存和每个线程的非易失性存储元组管理器组成。Zen 将混合表中的所有元组存储为元组堆中的非易失性存储元组，堆由固定大小的 2MB 页面（page）组成。非易失性存储元组由大小为 16 字节的头部（header）和元组数据组成。非易失性存储元组堆可能包含逻辑元组的几个版本，元组 ID 和事务提交时间戳 (Tx-CTS) 唯一地标识元组版本。1 bit 的删除位（Deleted）表示逻辑元组是否已被删除，最后持久化 (LP) 位表示元组是否是提交事务中持久化的最后一个元组。

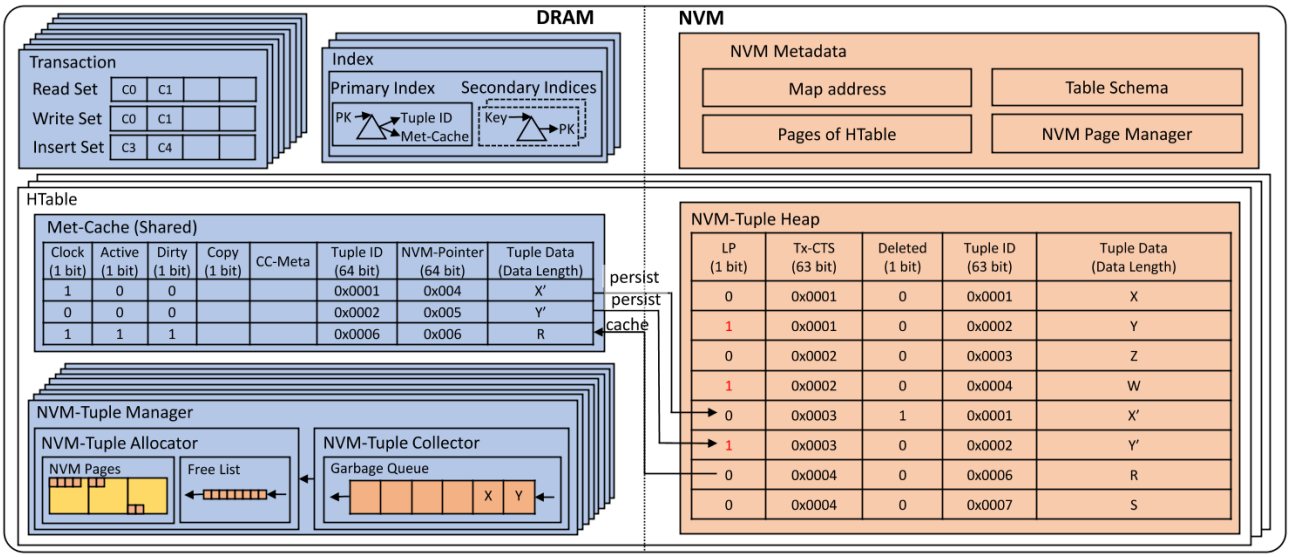

图 4. Zen 架构的细节设计图

Met-Cache 条目（entry）包含元组数据和七个元数据字段：元组 ID（Tuple ID）、脏位（Dirty）、正在活跃事务中的活动位（Active）、支持缓存替换算法的时钟位（Clock）、是否被复制位（Copy）以及用于事务并发控制的 CC-Meta 字段。在主索引中，值（key）指向元组缓存或非易失性存储元组堆中元组的最新版本。由于内存中的大小是有限的，所以 Zen 中次索引（Secondary Indices）的大小和数量是可选的。Zen 支持同时处理事务的多个线程，每个线程为内存中的事务私有数据保留线程本地空间来记事务读、写和插入操作。

Zen 同时结合了页面级和元组级来管理非易失性存储器的存储空间。非易失性存储页面管理器执行页面级空间管理，其分配和管理 2MB 大小的非易失性存储器页面。非易失性存储元数据中的映射地址和混合表页面维护从非易失性存储页面到混合表的映射。另一方面，非易失性存储元组（NVM-Tuple）管理器执行元组级空间管理。每个线程为每个混合表拥有线程本地非易失性存储元组管理器进行访问。每个非易失性存储元组管理器由一个分配器（Allocator）和收集器（Collector）组成，分配器维护着混合表中空闲元组的相关信息，而收集器则是不断接收从垃圾回收（garbage collection）中收集元素并将其放入到空闲列表（free list）中。

3.1 无日志的持久事务

Zen 中的事务处理由三个部分组成：

1. 执行：Zen 在内存中执行事务处理；
2. 持久化：Zen 将新编写的元组持久化到非易失性存储器；
3. 维护：Zen 垃圾收集陈旧的元组。

图 5描述了交易的生命周期，假设该表为客户保留帐户余额。最初，X 有 500 美元，Y 有 100 美元，Z 有 100 美元。交易将 100 美元从 X 转移到 Y，将 100 美元从 X 转移到 Z。图的上半部分显示了事务之前的系统状态 NVM 元组堆包含五个元组，其中 $R : d$ 已删除并收集垃圾，$Q : 300$ 缓存在元组缓存中，而索引则跟踪着有效元组的位置。如果存在，则指向对应元组缓存中的条目，分配器记录三个空的非易失性存储元组槽。图的下半部分显示了事务后的系统

状态。在执行处理中，Zen 将事务请求的三个元组（即 X、Y、Z）带入到元素缓存中。索引也相应地进行更新，并且事务在元素缓存中将 X 修改为 300、Y 修改为 200 和 Z 到 200。

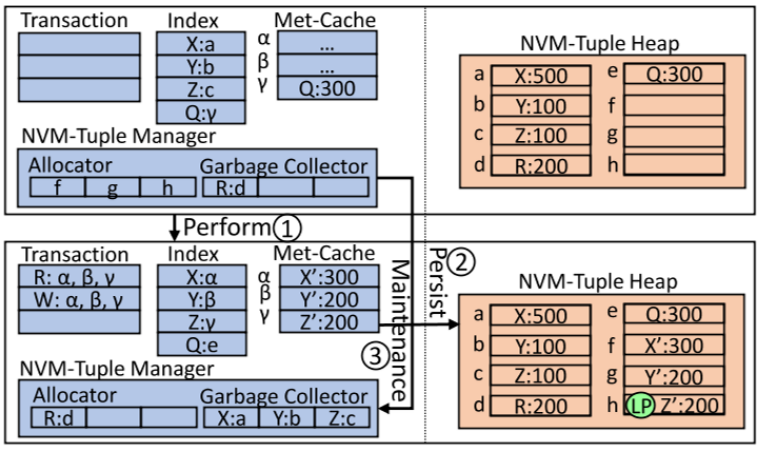

图 5. Zen 中事务的生命周期示意图

对于无日志持久事务的具体实现，首先 Zen 将元组持久化到一个空闲的非易失性存储器元组槽中。通过这种方式，元组的前一个版本在持久处理期间保持不变。在崩溃的情况下，Zen 可以回退到前一个版本。其次，再通过非易失性存储的原子写入标记最后一个元组的 LP 位以持久化在事务中。这样就可以确保所有元组在持久化 LP 位之前都持久化。在恢复过程中，如果 LP 位存在，则说明事务已经提交。由事务生成/修改的所有元组都必须成功持久化到非易失性存储器中。否则，崩溃发生在持久化事务的中间时，Zen 会丢弃事务写入的任何非易失性存储器元组。算法 1提供了这个元组持久化的整个过程。

Algorithm 1 持久化过程

Input: 事务中修改的元组 $changed-tuples$

```
for each tuple in changed − tuples do
    for p = tuple.start; p < tuple.end; p+ = 64 do
        clwb(p);
    end for
end for
last_tuple = changed_tuples [changed_tuples.size − 1];
for p = last_tuple.start; p < last_tuple.end; p+ = 64 do
    clwb(p);
end for
sfence();
last_tuple.LP = 1;
clwb(last_tuple.start);
```

3.2 灵活的并发控制方法

并发控制方法可以分为两类：单版本方法和多版本方法。而非易失性存储元组堆支持的是多版本方法来消除 redo 日记从而实现无日志机制。算法 2使用 ts-commit 来计算迄今为止

看到的最大提交时间戳。Zen 在遇到带有 LP 集的元组时更新 tscommit。如果一个元组的时间戳小于等于 ts-commit，Zen 考虑关联的事务已提交。如果索引包含元组的一个版本，Zen 将当前元组与索引中的版本进行比较。Zen 保留索引中的新版本，并将旧版本（如果存在）放入自由列表中。当元组的时间戳大于 ts-commit 时，Zen 此时无法判断相关事务的状态，并将将元组放入挂起列表中。

在在扫描区域后，ts-commit 是该区域的最大提交时间戳。然后，Zen 重新审视待处理列表中的元组。如果一个元组的时间戳小于等于 ts-commit，那么 Zen 使用元组更新索引，并可能垃圾收集索引中元组的旧版本。如果一个元组的时间戳大于 ts-commit，当关联的事务被持久化时，就会发生崩溃。由于线程只能写入自己的区域，因此事务的所有元组写入都指向同一个区域。这意味着扫描已经看到了事务写入的所有元组，但它们都没有 LP 集。因此，事务尚未完成，Zen 通过标记元组空闲槽并将其放入垃圾队列中来。

4 复现细节

4.1 与已有开源代码对比

论文中并未给出开源的代码，所以复现基于现有的 DBx1000 数据库管理系统（DBMS）进行开发和测试。DBx1000 是一款实验性单节点 OLTP 数据库管理系统，旨在评估未来 1000 核处理器上的并发控制协议，所有事务都作为嵌入式存储过程执行 [13]。

4.2 实验环境搭建

复现实验是在一台具有 nvm 实体存储介质的服务器上进行的。该服务服务器的处理器为英特尔 Intel(R) 志强 Xeon(R) 金牌处理器 Gold 6326，其主频为 2.90GHz。服务器的主存的大小是 128GB。搭配的非易失性存储是英特尔的 Optane 傲腾系列产品，其大小为 1TB。而软件方面，由于目前对于非易失性存储的系统级支持还不是特别全面，所以选择的是 Linux 发行版中的 Ubuntu Server 版。内核版本号为 5.4.0-165-generic。

4.3 软硬件使用说明

在 Linux 中，万物皆为文件，非易失性存储器与其他存储介质一样，需要挂载到文件系统中，并以文件形式进行访问。在本次复现实验中，如图所示，非易失性存储器被挂载在根目录中的 *mnt* 目录下。

```
Filesystem      1K-blocks       Used Available Use% Mounted on
/dev/pmem0     1027053960 134219776 840588988  14% /mnt
```

图 6. 复现实验中非易失性存储器的挂载示意图

对于复现实验的代码，是使用 C++ 语言编写的。在实验文件下，共分为 *benchmarks*、*concurrency_control*、*lib*、*storage* 和 *system* 文件夹以及其根目录下的其他若干文件。

1. *benchmarks* 文件夹存储的是对于实验数据的处理文件。在复现实验中，支持 YCSB [3] 和 TPC-C [19] 两种 benchmark。这两种 benchmark 是目前对于 OLTP 数据库最主流的选择，所以使用二者的实验结果具有一定的说服力；

2. *concurrency_control* 文件夹存储的是 Zen 关于并发控制的相关代码。包括对于死锁(deadlock)、乐观并发控制和多版本并发控制等内容;

3. *libs* 文件夹中是对现有的关于空间分配静态库的引用;

4. *storage* 中则是对于论文中 Zen 复杂架构的实现，包括混合表、树型索引和元组信息等内容;

5. *system* 文件夹中则是关于整个 Zen 引擎系统的一些组件，如对于非易失性存储元组缓存的管理机制、线程的相关管理和最重要的内存以及非易失性存储器的空间分配管理。需要说明的是在其中的 *nvm_memory.cpp* 等相关文件中需要引用 *libpmem.h* 文件（即 $\#include < libpmem.h >$）。这个头文件是英特尔公司专门为操作其非易失性存储器 Optane 产品所推出的开发工具，需要在系统级安装 *pmdk* 依赖;

6. 在实验文件的目录下还有一些其他的零散文件，包括 $Makefille$ 文件，config 相关文件。config 相关文件记录了实验中所有的参数信息并可供选择和切换，如 benchmark 的相关配置，线程个数和页面大小等大量信息。

编译实验代码时，需要在实验文件目录的终端下执行 *make* 命令。此时，实验文件目录下就会生成一个名为 rundb 的可执行文件。再在终端下执行 *./rundb* 命令即可运行相关的实验。

4.4 创新点

论文中提出了一种名为 Zen 的面向非易失性存储介质的 OLTP 高效存储引擎。但是存储引擎需要配合数据库管理系统才能真正地发挥其作用。本复现实验着重于实现 DBx1000 中关于 Zen 存储引擎的原型实现，虽然最终无法完全复现出论文中谁提出的所有创新点，但是大体上的框架已经搭建完成。并且通过进行实验也感受到了新型非易失性存储介质的特种特性以及所带来的问题。新型硬件的发现总是伴随着一些争议和困难，在执笔期前，学术界出现了一篇质疑非易失性存储介质实用性的文章 [9]。文章分析了不同持久或非易失性内存 (PMEM) 模式下不同数据库引擎的性能，表明内存模式下的 PMEM 没有明显的性能优势，使用 PMEM 作为永久存储可以加快查询执行速度，但有一些局限性。并且在进行该复现实验的过程中，也一定程度上地证实了这篇论文的结论。

5 实验结果分析

复现实验分别从存储介质种类、线程数和读写比例对代码进行验证。每张实验结果图可分为两部分，上部分为每个线程的运行信息，线程号从 0 开始，*tid* 的个数就表示实验的线程数。下半部分则是关于实验汇总信息的呈现，有事务成功还是拒绝各自的个数统计、运行时间、每秒处理事务的个数、索引时间和上下文切换次数等统计信息。

5.1 Zen 引擎在内存与非易失性存储介质的性能比较

本实验在相同的 8 线程下使用传统内存和非易失性存储器的 Zen 引擎性能对比。如图 7和图 8中的实验结果显示，使用非易失性存储器比使用传统内存的情况下，每秒钟的事务处理

个数可以提高 38% 以上。并且着还是在非易失性存储器容量大，导致大量上下文切换的情况下的实验结果。

```
[tid=0] txn_cnt=87,abort_cnt=1
[tid=1] txn_cnt=100,abort_cnt=1
[tid=2] txn_cnt=57,abort_cnt=0
[tid=3] txn_cnt=89,abort_cnt=1
[tid=4] txn_cnt=43,abort_cnt=0
[tid=5] txn_cnt=89,abort_cnt=0
[tid=6] txn_cnt=63,abort_cnt=0
[tid=7] txn_cnt=87,abort_cnt=0
[summary] txn_cnt=615, abort_cnt=3, run_time=0.016825, TPS=36552.205762, time_wait=0.000000, time_ts_alloc=0.000000, time_man=0.002669, time_index=0.002640, time_abort=0.000117, time_cleanup=0.001688, latency=0.000027, deadlock_cnt=0, cycle_detect=0, dl_detect_time=0.000000, dl_wait_time=0.000000, time_query=0.000059, debug1=0.000000, debug2=0.000000, debug3=0.000000, debug4=0.000000, debug5=0.000000, switch_cnt=4984
```

图 7. 在 8 线程下非易失性存储器的实验结果

```
[tid=0] txn_cnt=61,abort_cnt=0
[tid=1] txn_cnt=40,abort_cnt=1
[tid=2] txn_cnt=100,abort_cnt=0
[tid=3] txn_cnt=95,abort_cnt=0
[tid=4] txn_cnt=41,abort_cnt=0
[tid=5] txn_cnt=40,abort_cnt=0
[tid=6] txn_cnt=57,abort_cnt=1
[tid=7] txn_cnt=40,abort_cnt=0
[summary] txn_cnt=474, abort_cnt=2, run_time=0.018031, TPS=26288.485178, time_wait=0.000000, time_ts_alloc=0.000000, time_man=0.003256, time_index=0.002562, time_abort=0.000089, time_cleanup=0.001917, latency=0.000038, deadlock_cnt=0, cycle_detect=0, dl_detect_time=0.000000, dl_wait_time=0.000000, time_query=0.000058, debug1=0.000000, debug2=0.000000, debug3=0.000000, debug4=0.000000, debug5=0.000000, switch_cnt=3778
```

图 8. 在 8 现成下传统内存的实验结果

5.2 Zen 引擎在不同线程下的实验

本实验在不同线程下对比 Zen 引擎的性能。如图 9和图 10中的实验结果显示，使用单线程不会有事务间的冲突，所以事务失败的个数为 0。而在 8 个线程同时运行时，也仅有 0.7% 的事务失败概率，虽然每秒处理事务的个数不及单线程，但是能证明 Zen 对于处理并发事务上的机制是完全可行的，并且该复现也基本成功。

```
[tid=0] txn_cnt=100,abort_cnt=0
[summary] txn_cnt=100, abort_cnt=0, run_time=0.003041, TPS=32881.065241, time_wait=0.000000, time_ts_alloc=0.000000, time_man=0.000673, time_index=0.000437, time_abort=0.000000, time_cleanup=0.000445, latency=0.000030, deadlock_cnt=0, cycle_detect=0, dl_detect_time=0.000000, dl_wait_time=0.000000, time_query=0.000012, debug1=0.000000, debug2=0.000000, debug3=0.000000, debug4=0.000000, debug5=0.000000, switch_cnt=0
```

图 9. 在单线程下 Zen 引擎在 DBx1000 中的实验结果

```
[tid=0] txn_cnt=82,abort_cnt=1
[tid=1] txn_cnt=79,abort_cnt=0
[tid=2] txn_cnt=55,abort_cnt=0
[tid=3] txn_cnt=55,abort_cnt=1
[tid=4] txn_cnt=80,abort_cnt=1
[tid=5] txn_cnt=55,abort_cnt=0
[tid=6] txn_cnt=54,abort_cnt=1
[tid=7] txn_cnt=100,abort_cnt=0
[summary] txn_cnt=560, abort_cnt=4, run_time=0.024529, TPS=22830.290708, time_wait=0.000000, time_ts_alloc=0.000000, time_man=0.004692, time_index=0.003305, time_abort=0.000270, time_cleanup=0.002878, latency=0.000044, deadlock_cnt=0, cycle_detect=0, dl_detect_time=0.000000, dl_wait_time=0.000000, time_query=0.000097, debug1=0.000000, debug2=0.000000, debug3=0.000000, debug4=0.000000, debug5=0.000000, switch_cnt=4662
```

图 10. 在 8 线程下 Zen 引擎在 DBx1000 中的实验结果

5.3 Zen 引擎在不同读写比例下的实验

本实验在不同读写比例下对比 Zen 引擎的性能。如图 11和图 12中的实验结果显示，使用读操作较多时每秒处理事务个数就越多，这符合计算机中“读性能优于写性能”的直觉。并

且身为针对 OLTP 的存储引擎，在大量写操作的情况下，系统仍然可以又一个较高的事务处理性能。进一步证明了复现实验符合论文的描述和计算机直觉。

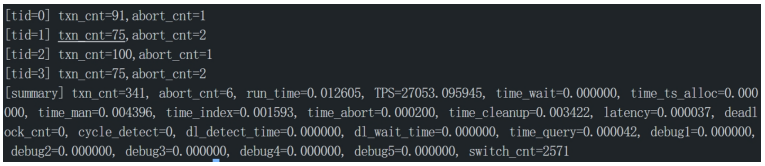

图 11. 在 benchmark 中读写比例各为 50% 下 Zen 引擎在 DBx1000 中的实验结果

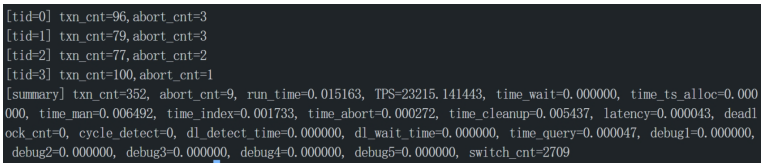

图 12. 在 benchmark 中写比例为 90% 下 Zen 引擎在 DBx1000 中的实验结果

5.4 结果分析

从实验结果来看，Zen 存储引擎是一个有效优化 OLTP 性能的存储引擎。但是复现实验结果并没有论文中那么好，原因可能有一下几点。

1. 存储引擎在不同数据库管理系统中的表现也不同；
2. 论文作者有更高的变成水平，并且该复现并没有完全实现所有特性；
3. 硬件所处的环境和系统对于整体性能表现有一定的影响；
4. 不同工作负载下的存储引擎的表现也不尽相同。

6 总结与展望

虽然 2022 年 9 月底，Intel 宣布彻底终结 Optane 傲腾业务 [7]。但是这包括 SSD 固态硬盘、PMem 持久内存两大产品线，并且其只是 NVM 的一种产品。并不能代表对于持久性存储介质发展和研究的结束。另一方面，学术界的相关论文数量也收到了一定程度的影响，但是在相关领域的顶会中要是不断有相关方向的论文发表。所以综上所述对于持久性存储等硬件的研究仍然会技术同软件算法的研究并行而一直发展下去。

参考文献

[1] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–35, 2013.

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, page 337–348, Nov 2016.

[3] brianfrankcooper. Ycsb. https://github.com/brianfrankcooper/YCSB.

[4] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[5] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1221–1231. IEEE, 2011.

[6] D. H. Graham. Intel optane technology products - what's available and what's coming soon. https://software.intel.com/en-us/articles/3dxpointtechnology-products.

[7] In Intel. 已停产英特尔 ® 傲腾 ™ 固态盘和模块的客户支持选项. https://www.intel.cn/content/www/cn/zh/support/articles/000024320/memory-and-storage.html.

[8] Hideaki Kimura. Foedus. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, May 2015.

[9] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. Nvm: Is it not very meaningful for databases? *Proceedings of the VLDB Endowment*, 16(10):2444–2457, 2023.

[10] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[11] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: Optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[12] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[13] Database of Databases. Dbx1000. https://dbdb.io/db/dbx1000.

[14] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[15] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[16] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[17] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[18] In Wikipedia. Flash memory. https://en.wikipedia.org/wiki/Flash_memory.

[19] In Wikipedia. Tpc-c. https://en.wikipedia.org/wiki/TPC-C.

[20] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. {HiKV}: a hybrid index {Key-Value} store for {DRAM-NVM} memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[21] J Joshua Yang and R Stanley Williams. Memristive devices in computing system: Promises and challenges. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):1–20, 2013.

[22] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. {NV-Tree}: reducing consistency cost for {NVM-based} single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[23] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

Algorithm 2 扫描非易失性存储元组堆

```
1: procedure 更新垃圾回收索引
2:     ptup = searchIndex(ntup)
3:     if ptup == NULL then
4:         insertIndex(ntup); return ;
5:     end if
6:     if ntup.Tx − CTS < ptup.Tx − CTS then
7:         insertIndex(ntup); return ;
8:     end if
9: end procedure
10: procedure 扫描对应区域
11:     ts − commit = 0; pending − list = ;
12:     for each ntup ∈ region do
13:         if ntup.LP then
14:             ts-commit= max(ts-commit, ntup.Tx-CTS);
15:         end if
16:         if ntup.Deletedorntup.Tx − CTS == 0 then
17:             putIntoFreeList(ntup);
18:             continue;
19:         end if
20:         if (ntup.Tx − CTS ts − commit then
21:             updateIndexGC(ntup);
22:         else
23:             put ntup into pending-list;
24:             continue;
25:         end if
26:     end for
27:     for each ntup ∈ pendingIist do
28:         if ntup.Tx − CTS ts − commit then
29:             updateIndexGC(ntup);
30:         else
31:             putIntoFreeList(ntup);
32:             ntup.Tx-CTS=0;
33:             clwb(ntup);
34:         end if
35:     end for
36:     sfence();
37: end procedure
```