# 基于NVM的无日志哈希表

王 涛[1] 梁 潇[1] 吴倩倩[1] 王 彭[2] 曹 伟[2] 孙建伶[1]
(浙江大学计算机科学与技术学院 杭州 310012)[1]
(阿里巴巴-浙江大学前沿技术研究中心 杭州 310012)[2]

**摘 要** 新兴的非易失内存正逐步进入人们的视野。由于这类存储技术同时具备了低延迟、持久化、大容量和字节可寻址的特性,数据库系统可以运行在只有NVM的存储架构上。在这种环境下,一些新颖的无日志索引结构应运而生,并被期望在异常故障后能即时地恢复索引能力而无须重建索引。然而,在现有的计算机体系结构中,这些索引结构为了确保NVM上数据的一致性,需要进行大量的同步操作,从而严重影响了正常执行时的系统性能。基于NVM的无日志哈希表利用指针数据的原子修改确保数据结构的一致性。哈希表使用了一种优化的Rehash方法,既减少了正常工作时的同步操作,又确保了异常故障后的即时恢复能力。实验评估表明,相比于已有的持久化索引结构,无日志哈希表在大部分工作负荷下的吞吐率表现良好,而在恢复时间、NVM资源使用量和写磨损方面具备显著的优势。

**关键词** 非易失内存,索引结构,持久化,即时恢复
**中图法分类号** TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.09.008

## Logless Hash Table Based on NVM

WANG Tao[1] LIANG Xiao[1] WU Qian-qian[1] WANG Peng[2] CAO Wei[2] SUN Jian-ling[1]
(College of Computer Science and Technology,Zhejiang University,Hangzhou 310012,China)[1]
(Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies,Hangzhou 310012,China)[2]

**Abstract** Emerging non-volatile memory(NVM) is taking people's attention. Due to the advantages of low latency, persistence,large capacity and byte-addressable,database system can run on the NVM-only storage architecture. In this configuration,some novel logless indexing structures come into being and are expected to recover indexing capability immediately after an system failure. However, under the current computer architecture, these structures need a large amount of synchronizations to ensure data consistency,which leads to a severe performance penalty. NVM-baesd logless hash table leverages the atomic update of the pointer data to ensure the consistency. An optimized rehash procedure was proposed to not only reduce the synchronizations during normal execution,but also ensure the instant recovery after system failures. Performance evaluation shows that,compared with existing persistent indexing structures,logless hash tables perform well under most workloads,and have significant advantages in terms of recovery time,NVM footprint, and write wear.

**Keywords** Non-volatile memory,Indexing structure,Persistence,Instantly recoverable

新兴的非易失内存(Non-Volatile Memory,NVM)[1]正逐步进入人们的视野。由于同时具备了低延迟、持久化、高容量和字节寻址的特性,人们期望数据库系统可以将数据存放在NVM单层存储上,使得不需要日志也能保证系统的持久性和一致性。由于消除了日志,这类系统拥有断电故障后的即时恢复能力和更高效的存储资源使用率。

目前,一些无日志的索引结构[2-3]被陆续提出,作为无日志数据库系统的核心组件。这些无日志索引结构的主要思想是,写操作直接作用在"目的位置"上,并通过对数据的原子修改提交整个操作。然而,现有的计算机体系结构却阻碍了这类系统的高效实现。写操作的数据可能会驻留在CPU的高速缓存中,这些数据在发生断电故障后将会丢失,从而破坏索引结构的一致性。因此,索引结构必须在操作提交前通过同步指令将高速缓存中的数据逐出。同步指令有显著的性能开销,当存在大量的同步操作时,如由于维护数据结构的某种特性而移动数据项的过程,会导致性能大幅下降。已有的索引结构要么承受这样的性能损失[2],要么只对少量的关键数据进行同步,而在故障重启时重建其他数据[3]。然而,重建过程

到稿日期:2018-07-05 返修日期:2018-09-15
王 涛(1992-),男,硕士,主要研究方向为数据库系统设计,E-mail:wangt0907@zju.edu.cn;梁 潇(1995-),男,硕士,主要研究方向为数据库系统设计;吴倩倩(1996-),女,硕士,主要研究方向为数据库系统设计;王 彭(1989-),男,硕士,主要研究方向为数据库系统、分布式存储系统、云计算;曹 伟(1984-),男,硕士,主要研究方向为分布式数据库与存储系统、大规模实时计算、云计算等;孙建伶(1964-),男,博士,教授,主要研究方向为数据库系统、机器学习、金融科技、软件工程等,E-mail:sunjl@zju.edu.cn(通信作者)。

的巨大开销使得索引结构失去了即时恢复的能力。

本文提出了一种基于 NVM 的无日志哈希表。相比于树形索引结构，哈希表的简单结构使得写操作只需要最少的同步操作就能保持整个数据结构的一致性，从而兼具了正常操作时的高性能和断电故障后的即时恢复能力。对于哈希表扩容的 Rehash 过程，本文提出了一种优化的 Rehash 方法，该方法只同步尽可能少的信息来保证数据项不丢失。而对于断电故障可能产生的数据不一致，本文算法使得不一致数据对系统不可见，并将不一致数据的恢复过程推迟到数据真正访问前执行，以确保索引结构的即时恢复能力不会被破坏。

本文的主要贡献如下：

1)提出了一种基于 NVM 的无日志哈希表，利用指针数据的原子修改实现了哈希的原子写操作，确保了哈希表在断电故障后依然保持一致。

2)提出了一种优化的 Rehash 方法，只选择数据项在 Rehash 输出表上的前继数据项的后继指针进行同步，以提高正常操作时的效率，并对异常故障产生的数据不一致设计相应的即时恢复方法。

3)对无日志哈希表和相关工作进行了性能评估，评估结果表明无日志哈希表在大多数工作负荷下的吞吐率表现良好。即时恢复是无日志哈希表最突出的特点，在恢复时间上相比于其他索引结构减小了 3～4 个数量级。此外，无日志哈希表在 NVM 资源使用量和写磨损方面也均好于其他两种数据结构。

本文第 1 节介绍了已有的相关工作；第 2 节概述了索引结构；第 3 节介绍了数据访问接口的设计；第 4 节介绍了优化的渐进式 Rehash 方法；第 5 节介绍了性能评估情况；最后总结全文，并介绍了未来的研究方向。

## 1　相关工作

### 1.1　非易失内存技术

非易失内存是一类同时满足类似内存的低延迟、字节寻址特性和类似磁盘的持久化的存储技术。尽管 NVM 囊括了多种不同的实现技术，如 PCM 技术[4]、STT-RAM 技术[5]和 Memristor 技术[6]，但目前大规模投入工业应用的是由镁光公司和英特尔公司推出的 3D XPoint 技术[7]。这些技术在存储特性上仍存在明显的差别，本文所提到的 NVM 技术主要指代 3D XPoint 技术，所有的性能参数和实验数据也基于此。表 1 列出了 NVM 和其他成熟存储技术的特性[8]。

表 1　NVM 与其他存储技术的特性

Table 1　Characteristics NVM and other storage technologies

| | DRAM | NVM | SSD | HDD |
|---|---|---|---|---|
| 读延迟 | 60 ns | 250 ns | 25 us | 10 ms |
| 写延迟 | 60 ns | 500 ns | 300 us | 10 ms |
| 寻址单元 | 字节 | 字节 | 块 | 块 |
| 易失性 | 易失 | 非易失 | 非易失 | 非易失 |
| 容量 | 1x | 4x | 4x | — |
| 写寿命 | $10^{16}$ | $10^{10}$ | $10^{5}$ | $10^{16}$ |
| 价格 | >150x | >60x | >8x | 1x |

在现有的体系结构下，NVM 通常通过内存总线接入计算机系统，如图 1 所示。这样，写入 NVM 的数据就可能驻留在 CPU 的高速缓存中，如果发生断电故障，这些数据将会丢失。因此，NVM 的写操作仍然需要通过同步操作确保一致性。同步操作通常由 CLWB 和 SFENCE 两条指令完成，以确保相应的缓存条目被依序逐出。然而，同步指令存在严重的性能开销，使得 NVM 的写带宽下降 2～3 个数量级。因此，如何在软件设计中尽量避免同步操作以提升系统性能是一个重要的问题。

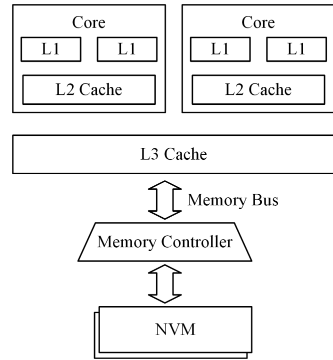


图 1　当前计算机体系结构下 NVM 设备的访问路径

Fig. 1　NVM device access path under current computer architecture

### 1.2　基于 NVM 的无日志索引结构

NVM 细粒度的持久化能力，使得基于 NVM 的无日志持久化机制正逐渐变得可行。这类持久化机制有两个主要特点：1)写操作的数据都直接写到"目的位置"，因此已提交的操作不需要 Redo；2)写操作的数据在提交前对系统的影响是不可见的，因此未提交的操作不需要 Undo。目前，NVM 只支持 8 字节数据写操作的原子性，因此，如何将 8 字节数据作为写操作的提交是设计无日志持久化机制的关键。

CDDS[2]是一种较早提出的无日志 B 树索引，通过多版本机制实现无日志持久化。算法在重启时，通过全局版本号确定所有不一致的数据，并将其忽略。CDDS 的性能瓶颈在于维持 B 树节点的有序性。为了避免数据项排序过程中可能发生的断电故障导致数据不一致的现象，CDDS 需要在每个数据项移动后都进行同步，这一动作占据了整个操作 90% 以上的执行时间[3]。

NV Tree[3]是一种优化的无日志 B 树索引。它采用了 Log Structured 的思想，写操作在每个节点的尾部追加一个新的数据项，并更新当前节点中数据项的个数 $k$，因此数据项的数量信息 $k$ 被视为写操作的提交。在发生断电故障后，算法只将节点中前 $k$ 个数据项视为有效数据。为了减少维持数据一致性的开销，NV Tree 做出了以下设计选择：对于叶节点，不维护数据项的顺序；对于中间节点，维护数据项的顺序，但不进行同步操作。在发生断电故障后，中间节点可能处于不一致的状态，需要通过叶节点进行重建。重建存在较大的开销，因此 NV Tree 不具备即时恢复能力。

### 1.3　渐进式 Rehash

为了保持哈希表的高效访问，当哈希表中的数据项达到一定数目时，需要对哈希表进行扩容，通过 Rehash 将哈希表中的数据项迁移到一个更大的哈希表中。如果哈希表中的数据项较多，Rehash 过程会消耗过长的时间，从而导致工作线程阻塞。为了避免这一现象，渐进式 Rehash[9]被越来越多地

应用到哈希表设计中。渐进式 Rehash 的主要思想是以哈希桶为粒度对整个 Rehash 过程进行划分，每个哈希桶的 Rehash 过程穿插在哈希表的日常工作中进行。因此，哈希表在 Rehash 过程中会存在两个子哈希表，其中，Rehash 的输入表被称为 Primary 表，输出表被称为 Secondary 表，用户请求可能会同时访问这两张表。算法 1 描述了采用渐进式 Rehash 的链式哈希表对一个哈希桶进行 Rehash 的过程。当所有的哈希桶都完成 Rehash 后，算法将 Primary 表替换为 Secondary 表，以完成 Rehash 操作。

**算法 1** 渐进式 Rehash 算法 rehash_bucket

输入：Rehash 输入表 primary，Rehash 输出表 secondary，哈希桶序号 idx

输出：无

```
1. for each entry e in primary. buckets[idx] do
2.    nidx←hash(e. key) mod secondary. size
3.    e. next←secondary. buckets[nidx]
4.    secondary. buckets[nidx]←e
5. end for
```

## 2 索引结构概述

### 2.1 NVM 访问接口和空间管理

本文通过 DAX 文件系统[10]访问 NVM 设备：程序将 NVM 设备映射到虚拟地址空间中，之后由 CPU 通过 Load/Store 指令直接访问 NVM 上的数据。在程序的多次执行中，NVM 会被确保映射到相同的地址空间上，因此指向 NVM 中某一位置的指针在程序重启后依然保持有效。基于此，本文通过一个基于 NVM 的动态分配器来管理 NVM 空间。动态分配器提供了 nvm_alloc/nvm_free 的接口，以满足哈希表动态分配 NVM 空间的需求。动态分配器通过提交操作和垃圾回收机制来确保 NVM 空间在发生断电故障后的持久性和一致性。

### 2.2 无日志哈希表

无日志哈希表采用了链式哈希表设计，而不是性能更好的杜鹃哈希[11]或跳步哈希[12]，因为链式哈希在增删数据时不需要大量的数据移动，可以避免大量的 NVM 同步操作。如图 2 所示，在 NVM 的起始位置上保存了分别指向 Primary 表和 Secondary 表的指针。由于指针长度通常为 8 字节，因此哈希表可以原子地将 Primary 表指针指向 Secondary 表以完成整个 Rehash 操作。在数据结构上，相比于传统链式哈希表，无日志哈希表在每个哈希桶上多了一个额外的 Dirty 标识位，用于指示相应的哈希桶中是否可能存在不一致的数据，如果哈希表一直处于正常运行，标识位将总是处于复位状态；只有在故障重启后，哈希表才会将所有哈希桶的 Dirty 标识位置位。在算法方面，无日志哈希表与传统哈希表的最大区别有两点：1)在恰当的时机对 NVM 上的数据进行同步操作，以确保写操作的持久性；2)通过对哈希表操作算法的调整，确保系统故障产生的数据不一致不会影响系统的继续运行。

## 3 哈希表数据访问接口

### 3.1 插入操作

算法 2 描述了无日志哈希表的插入操作。在插入操作中，数据项会被插入到相应哈希桶的头部，哈希桶的 next 指针的修改被视为提交动作。算法 2 中的第 1－5 行根据当前是否存在未完成的 Rehash 选择合适的子哈希表作为插入操作的目的表，以确保数据项在 Rehash 后不会丢失。第 7 行通过动态分配器分配插入数据项所需的空间。第 8－11 行在数据项中填充必要的信息，包括插入的键、值以及哈希桶当前的表头。第 12 行对数据项进行同步，需要注意的是，这次同步操作必须保证在操作提交前执行，这被称为 Flush-before-Commit 原则；否则，如果提交操作的同步顺序先于数据项修改，断电故障可能会使已提交的数据项处于不一致状态。第 13－14 行将哈希桶的 next 指针指向新插入的数据项，并通过第二次同步操作对插入操作进行提交。如果在第 14 行前发生了断点故障，由于插入操作并未对哈希表的其他部分产生影响，因此不会产生不一致现象。无日志哈希表的布局示意图如图 2 所示。

**算法 2** 无日志哈希表插入操作 insert

输入：无日志哈希表 lht，插入键 key，插入值 val

输出：无

```
1. if an ongoing rehashing exists do
2.    target←lht. secondary
3. else
4.    target←lht. primary
5. end if
6.
7. e←nvm_alloc(sizeof(entry))
8. idx←hash(key) mod target. size
9. e. key←key
10. e. val←val
11. e. next←target. buckets[idx]. next
12. sync(e)
13. target. buckets[idx]. next←e
14. sync(target. buckets[idx]. next)
```

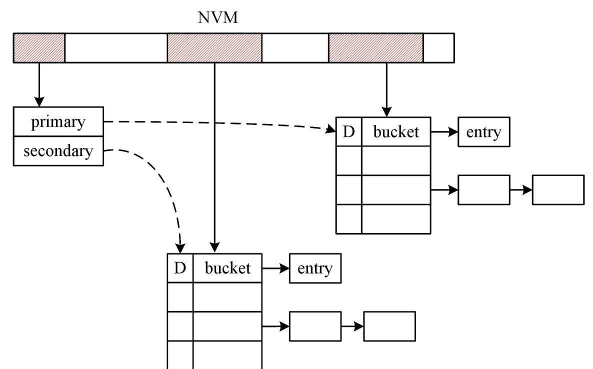


图 2 无日志哈希表的布局示意图

Fig. 2 Layout diagram of logless hash table

### 3.2 查找操作

算法 3 描述了无日志哈希表的查找操作。由于使用了渐进式 Rehash，查找操作最多会访问两个子哈希表。相比于传统哈希表，无日志哈希表的查找操作存在两个特点：1) Secondary 表先于 Primary 表被查找(第 1 行)，这是由于 Rehash 过程可能会产生重复的数据项，在这种情况下，哈希表总是将 Secondary 表中的版本视为有效版本，而忽略 Primary 表中的

版本；2）在遍历哈希桶上的数据项前，算法会对其进行一致性检查（第 4－6 行），一致性检查用于恢复哈希表在断电故障后可能存在的第二种数据不一致——过期指针。为了避免多余的一致性检查产生的开销，每个桶存在一个 Dirty 标识位。当程序一直正常运行时，Dirty 标识位处于置位状态，一致性检查将会被跳过。第 4 节将详细讨论无日志哈希表面临的数据不一致和相应的恢复机制。

**算法 3**　无日志哈希表查找操作 lookup

输入：无日志哈希表 lht，查找键 key

输出：如果存在查找键，则返回相应的数据项，否则返回 NULL

```
1. for target∈{lht. secondary, lht. primary} do
2.   if target≠NULL do
3.     idx←hash(key) mod target. size
4.     if target. buckets[idx]. dirty＝TRUE do
5.       check_consistency(target, idx)
6.     end if
7.     e←target. buckets[idx]. next
8.     while e≠NULL do
9.       if e. key＝key do
10.        return e
11.      end if
12.      e←e. next
13.    end while
14.  end if
15. end for
16. return NULL
```

### 3.3　更新操作

算法 4 描述了无日志哈希表的更新操作。更新操作将包含更新键的数据项的前继数据项的后继指针的修改作为提交动作。由于哈希表的键值长度可能超过 8 个字节，为了使更新操作能够原子地执行，本文使用了写时拷贝的策略。算法 1 中的第 1 行寻找包含更新键的数据项，lookup_prev 的逻辑类似于 lookup，但是返回数据项的前继数据项。接着，算法申请一块新的数据项来存放更新键和相应的更新值。第 8 行同样通过 Flush-before-Commit 原则确保更新操作在提交后的一致性。第 9－10 行将前继数据项的后继指针指向新的数据项并进行提交。第 11 行释放旧数据项所占用的空间。需要注意的是，这一操作不能移动到提交操作之前，否则如果更新操作最终未提交，而旧数据项却被释放，则会引起数据不一致。

**算法 4**　无日志哈希表更新操作 update

输入：无日志哈希表 lht，更新键 key，更新值 val

输出：无

```
1. prev←lookup_prev(lht, key)
2. if prev≠NULL do
3.   olde←prev. next
4.   e←nvm_alloc(sizeof(entry))
5.   e. key←key
6.   e. val←val
7.   e. next←olde. next
8.   sync(e)
9.   prev. next←e
10.  sync(prev. next)
11.  nvm_free(olde)
12. end if
```

### 3.4　删除操作

算法 5 描述了无日志哈希表的删除操作。删除操作类似于查找操作，可能会访问两个子哈希表，但其关键的区别在于，删除操作会先访问 Primary 子表，再访问 Secondary 子表。因此，对于重复数据项，Secondary 子表上的版本被删除才代表删除操作成功执行。如果不遵从这个次序，一次失败的删除操作可能删除了 Secondary 子表上的最新版本而保留了 Primary 子表上的过期版本，从而产生数据不一致现象。

**算法 5**　无日志哈希表删除操作 delete

输入：无日志哈希表 lht，删除键 key

输出：无

```
1. for target∈{lht. primary, lht. secondary } do
2.   if target≠NULL do
3.     idx←hash(key) mod target. size
4.     if target. buckets[idx]. dirty＝TRUE do
5.       check_consistency(target, idx)
6.     end if
7.     prev←target. buckets[idx]
8.     while prev. next≠NULL do
9.       if prev. next. key＝key do
10.        prev. next←prev. next. next
11.        sync(prev. next)
12.        break
13.      end if
14.      prev←prev. next
15.    end while
16.  end if
17. end for
```

## 4　优化的渐进式 Rehash

### 4.1　算法概述

算法 6 和算法 7 描述了无日志哈希表的优化的渐进式 Rehash 操作。查找操作中用于避免不一致数据的方法同样被应用于 Rehash 过程，比如，遍历哈希桶前的一致性检查（算法 6 中的第 2－4 行）和忽略重复数据项在 Primary 表上的版本（算法 7 中的第 8－10 行）。算法的主要目标是确保 Rehash 过程中的断电故障不会使哈希表丢失已有的数据项，并尽可能地减少同步操作。为此，本文在传统的渐进式 Rehash 算法上做出了以下调整。

1）当数据项被 Rehash 到 Secondary 子表中时，将这个数据项插入到相应哈希桶的末尾而非头部（算法 7 中的第 5－14 行），并同步其前继对象的后继指针（算法 7 中的第 15 行）。同步操作确保了数据项可以由 Secondary 子表进行访问，因此，即使 Primary 表上的访问路径被破坏，也不会丢失该数据项。另一方面，将数据项插入到哈希桶的末尾而非头部，减少了同步操作的数量。如果将数据项插入到头部，为了确保哈希桶上所有数据项都能被访问到，需要对哈希桶及数据项分别进行一次同步操作。相比于插入头部，插入末尾需要遍历哈希桶中已有的数据项，从而产生更多的读操作，但由于 NVM 上读操作的性能远远高于同步操作，因此插入末尾仍然能够获得性能提升。

2）在 Primary 子表上，从一个哈希桶的末尾数据项开始

逆序地对每个数据项 Rehash。本文通过递归来实现这种执行策略(算法 7 中的第 1—3 行)。逆序 Rehash 使得数据项被移动到 Secondary 子表上后能够安全地将其后继指针指向 NULL,以截断旧的链表(算法 7 中的第 16 行)。这是因为,该数据项的后继数据项已经在更早的时候被 Rehash 到Secondary 子表上,由于第 1)点已确保了数据项在 Rehash 后一定能被 Secondary 子表访问,因此修改后继指针不会丢失那个数据项。

3)对于旧链表的截断操作(算法 7 中的第 16 行),不施加同步操作。这样做的代价是,断电故障可能使已经完成 Rehash 的数据项的后继指针指向旧的数据项。4.2 节讨论了这类数据的不一致性,并设计了相应的恢复机制。

综合上述几点,优化后的渐进式 Rehash 在保证数据项不丢失的基础上,对每个数据项只进行一次同步操作,有效地减小了数据同步的开销。

**算法 6** 优化的渐进式 Rehash 算法 rehash_bucket_opt

输入:无日志哈希表 lht,哈希桶序号 idx

输出:无

```
1. if lht. primary. buckets[idx]. next≠NULL do
2.    if target. buckets[idx]. dirty=TRUE do
3.       check_consistency(target. buckets[idx])
4.    end if
5.    rehash_entry(lht. primary. buckets[idx]. next)
6.    lht. primary. buckets[idx]. next←NULL
7. end if
```

**算法 7** 单个数据项 Rehash 算法 rehash_entry

输入:无日志哈希表 lht,数据项 e

输出:无

```
1. if e. next≠NULL do
2.    rehash_entry(e. next)
3. end if
4. idx←hash(e. key) mod lht. secondary. size
5. prev←lht. secondary. buckets[idx]
6. while prev. next≠NULL do
7.    if prev. next =e do
8.       return
9.    end if
10.   prev←prev. next
11. end while
12. prev. next←e
13. sync(prev. next)
14. e. next←NULL
```

### 4.2 数据不一致及恢复

优化的渐进式 Rehash 可能会在断电故障后产生不一致数据,原因是:1)单个数据项的 Rehash 过程不能确保被原子地执行;2)对旧链表的截断操作(算法 7 中的第 16 行)没有同步,哈希表可能会面临两类数据不一致。

1)重复数据:数据项同时存在于 Primary 子表和 Secondary 子表中。

2)过期指针:数据项已经迁移到 Secondary 子表,但是其后继指针依然指向了 Primary 子表中的后继对象。

图 3 给出了产生数据不一致的场景。最初,“A” “B”两个数据项处于 Primary 子表上的同一个哈希桶中。接着,两个数据项被依次 Rehash 到 Secondary 子表中。在 Secondary 子表中,它们处于不同的哈希桶中。此时,哈希桶和数据项“A”的后继指针均更新为 NULL,但是修改操作没有同步到 NVM 中。如果发生了断电故障,程序重启后会发现哈希桶和数据项“A”的后继指针回退为 Rehash 前的内容。数据项“A”同时存在于两个子表中,形成了重复数据异常。数据项“A”的后继指针指向了另一个哈希桶中的数据项“B”,形成了过期指针异常。

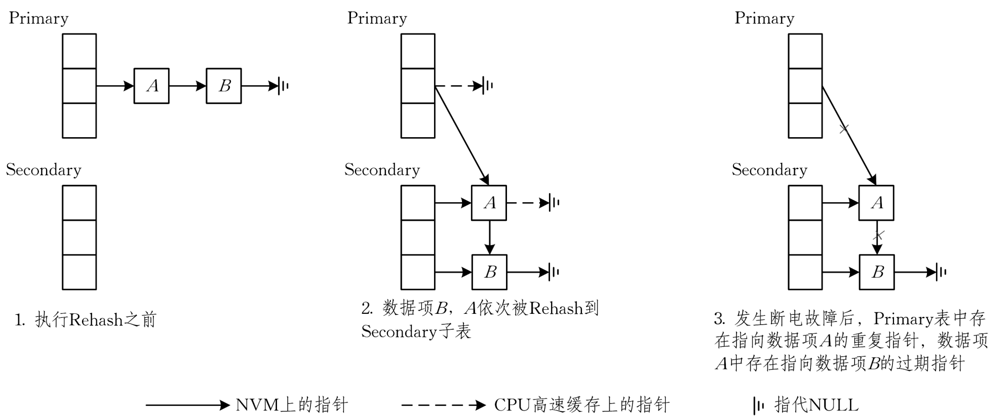


图 3 Rehash 过程在断电故障后可能遇到的数据不一致

Fig. 3 Inconsistent data of Rehash process after power failure

针对上述两种数据不一致的情况,本文设计了相应的恢复方法,并确保哈希表的即时恢复能力不被破坏。对于重复数据,哈希表总是忽略 Primary 子表中的版本,查找操作、删除操作和单个数据项的 Rehash 算法体现了这一原则。对于过期指针,本文在访问可能存在过期指针的哈希桶前对其进行了一致性检查。算法 8 描述了这一过程。算法将遍历哈希桶中的所有数据项,当发现过期指针(后继对象的哈希值不等于当前哈希桶对应的哈希值)时(算法 8 中的第 4 行),将链表截断。为了避免对一个哈希桶重复进行一致性检查,哈希表为每个哈希桶保留一个 Dirty 标识位。一致性检查完成后,算法将会重置该标识位(算法 8 中的第 11 行)。Dirty 标识位处于重置状态的哈希桶在下一次访问时不需要进行一致性检查。如果程序一直处于正常的运行状态,哈希表不会存在数据不一致的情况,因此所有哈希桶的 Dirty 标识位始终处于

重置状态。在断电故障重启后，算法会保守地将所有哈希桶的 Dirty 标识位置位，以避免可能存在过期指针，而过期指针的检查和恢复过程被延迟到每个哈希桶被真正访问前进行。标识位置位是哈希表在故障恢复时的唯一动作，由于该过程通常很快，因此哈希表能够即时地恢复。

**算法 8**　一致性检查 check_consistency

输入：子哈希表 target，哈希桶序号 idx

输出：无

```
1. prev←target. buckets[idx]
2. while prev. next≠NULL do
3.    tidx←hash(prev. next. key) mod target. size
4.    if tidx≠idx do
5.       prev. next←NULL
6.       break
7.    end if
8.    prev←prev. next
9. end while
10.  target. buckets[idx]. dirty←FALSE
```

## 5　实验评估

### 5.1　实验环境

本文对无日志哈希表与另两种持久化数据结构——NV Tree 和基于日志的哈希表进行了实验评估。由于 NV Tree 并未开源，实验使用了自行实现的 NVTree 版本。基于日志的哈希表源自内存数据库 Redis[9] 的开源代码实现，它通过 Write Abead Log(WAL)[13] 保证写操作的持久化，WAL 通过校验码确保单条日志的原子写。首先，本文独立地评估了 3 种数据结构的性能，包括吞吐率、恢复时间、NVM 使用量和写磨损。实验中键值对的键长度均为 8 字节，而值长度通常作为变量来测试数据结构在不同工作负荷下的表现，否则也为 8 字节长。然后，本文将 3 种数据结构集成到内存数据库 Redis[9] 中，并通过 YCSB[14] 端到端地评估了 3 种 Redis 的吞吐率。实验采用了 NVM 单一存储架构，所有的数据结构、日志均放置于 NVM 上，而程序的代码段及堆栈依然放置于 DRAM 中。表 2 列出了实验的软硬件环境。

表 2　实验软硬件环境

Table 2　Experimental software and hardware environment

| 类型 | 参数 |
|---|---|
| 操作系统 | Linux 64 位操作系统(RHEL7) |
| 中央处理器 | Intel(R) Xeon(R) 8163 CPU @ 2.50 GHz |
| 内存信息 | 256 GB DDR4(2 666 MHz) |
| NVM 设备 | Intel 3D XPoint NVDIMM |
| 缓存大小 | 64 KiB/1 MiB/32 MiB(L1/L2/L3) |
| 编译器 | GCC 4.8.5 |

### 5.2　吞吐率

本文评估了 3 种数据结构(即 Set，Get 和 Delete 操作)的吞吐率，并关注了值长度大小对吞吐率的影响，结果如图 4 所示。

对于 Set 操作，无日志哈希表在值长度较短时表现不佳，随着值长度的增长，无日志哈希表的性能下降最慢，当值长度超过 512 字节时吞吐率最高。基于日志的哈希表由于其写放大较高，吞吐率随值长度增加而大幅下降。NV Tree 的吞吐率同样随值长度增加而大幅下降。

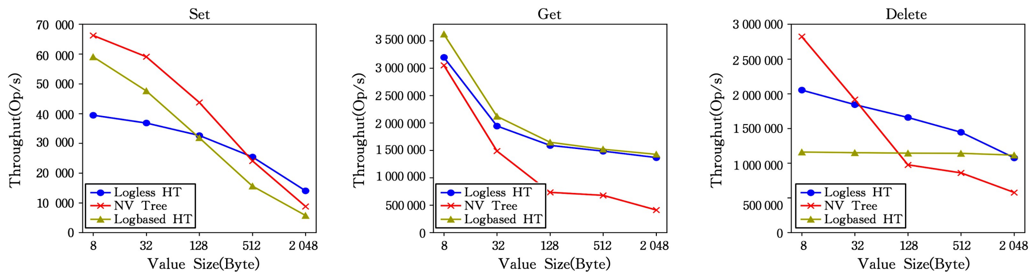


图 4　3 种数据结构下不同操作的吞吐率比较

Fig. 4　Comparison of throughput rates for different operations of three data structures

对于 Get 操作，基于日志的哈希表由于不存在额外开销，吞吐率最高。无日志哈希表的吞吐率整体上与基于日志的哈希表较为接近，并且当值长度大于 128 字节时，两者的差距很小，表明此时无日志哈希表由于维持一致性产生的额外开销几乎可以忽略。NV Tree 的吞吐率随值长度的增加而显著下降，表现较差。

对于 Delete 操作，由于 Delete 日志的代价与值长度无关，因此基于日志的哈希表的吞吐率始终维持在恒定水平。无日志哈希表和 NV Tree 的吞吐率则随值长度的增加而显著下降。

### 5.3　恢复时间

本文评估了 3 种数据结构在不同键值对数量下的恢复时间，结果如表 3 所列。相较于 NV Tree 和基于日志的哈希表，无日志哈希表在恢复时间上降低了 3～4 个数量级。

表 3　3 种数据结构的恢复时间的比较

Table 3　Comparison of Recovery time of three data structures

(单位：s)

| 键值对数量/万 | Logless HT | NV Tree | Logbased HT |
|---|---|---|---|
| 200 | 0.408 | 945.526 | 6 060.299 |
| 400 | 0.511 | 1 371.396 | 13 736.280 |
| 600 | 0.769 | 2 787.171 | 20 939.880 |
| 800 | 1.531 | 3 476.509 | 28 545.740 |
| 1 000 | 2.008 | 4 162.831 | 44 795.650 |

### 5.4　NVM 资源使用量和写磨损

本文在 3 种数据结构中分别插入了 1000 万条键值对，并评估了 NVM 资源使用量和写磨损情况。NVM 资源使用量通过统计动态分配器的空间开销和日志大小得出，而写磨损通过 Intel vTune 放大器[15] 统计插入过程中 NVM 上 Store 指令的数量得出。表 4 列出了评估结果。无日志哈希表的单层结构使其拥有最低的 NVM 资源使用量和最小的写磨损；

基于日志的哈希表则由于日志产生的开销,需要比无日志哈希表多接近 1 倍的 NVM 空间和写磨损;NV Tree 由于中间节点的开销,在资源使用量上较无日志哈希表略多,而写磨损方面则由于节点分裂和合并的开销,表现最差。

表 4 3 种数据结构下 NVM 资源使用量和写磨损的比较

Table 4 Comparison of NVM resource usage and write wear of three data structures

| | Logless HT | NV Tree | Logbased HT |
|---|---|---|---|
| NVM 使用量/MiB | 564 | 574 | 712 |
| Store 指令数 | 182000273 | 1008001512 | 350000525 |

### 5.5 YCSB 评估

实验评估了 3 种 Redis 在不同值长度下吞吐率随读写操作比例的变化情况,以此反映它们在不同工作负荷下的综合表现。实验开启了 YCSB 客户端多线程选项,以最大化吞吐率,而服务器端则由于 Redis 本身的单线程模型无法进一步扩展。

图 5 给出了评估结果。3 种 Redis 在不同工作负荷下的吞吐率相差不大,表明在端到端的环境中,系统的性能主要受到了其他因素(如网络、JVM)的制约。

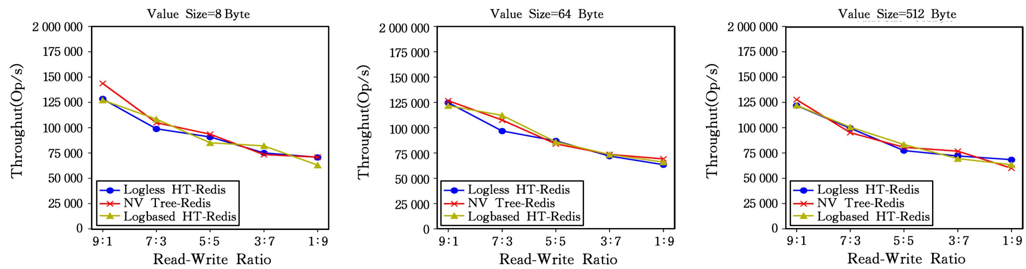


图 5 基于 3 种数据结构的 Redis 在 YCSB 上的吞吐率比较

Fig. 5 Comparison of throughput rate of Redis based on three data structures on YCSB

**结束语** 新兴的非易失内存技术为数据库系统持久化方案提供了新的选择。本文设计了一种基于 NVM 单层存储的无日志哈希表,利用指针数据的原子修改来保证哈希表在异常故障后的一致性。本文同时设计了一种优化的 Rehash 方法,该方法既减少了正常工作时的同步操作,又不影响异常故障后的即时恢复能力。实验评估表明,相对于经典的基于日志的哈希表和较新的无日志数据结构,本文提出的无日志哈希表在大多数工作负载下表现良好,并在即时恢复、资源使用量和写磨损上具备显著优势。

## 参 考 文 献

[1] MEENA J S,SZE S M,CHAND U,et al. Overview of emerging nonvolatile memory technologies[J]. Nanoscale Research Letters,2014,9(1):526.

[2] VENKATARAMAN S,TOLIA N,RANGANATHAN P,et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory[C]∥Proceedings of 9th USENIX Conference on File and Storage Technologies. San Jose,California:USENIX,2011:61-75.

[3] YANG J,WEI Q,CHEN C,et al. NV-Tree:Reducing Consistency Cost for NVM-based Single Level Systems[C]∥Proceedings of 13th USENIX Conference on File and Storage Technologies. San Jose,California:USENIX,2015:167-181.

[4] HUANG C C. Phase change random access memory:U. S. Patent 7,504,652[P]. 2009-3-17.

[5] DRISKILL-SMITH A. Latest advances and future prospects of STT-RAM[C]∥Non-Volatile Memories Workshop. San Diego:University of California,2010:11-13.

[6] STRUKOV D B,SNIDER G S,STEWART D R,et al. The missing memristor found[J]. Nature,2008,453(7191):80-83.

[7] Intel and Micron produce breakthrough memory technology [EB/OL]. http://newsroom. intel. com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[8] ARULRAJ J,PAVLO A,DULLOOR S R. Let's talk about storage & recovery methods for non-volatile memory database systems[C]∥Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. New York:ACM,2015:707-722.

[9] ZAWODNY J. Redis:Lightweight key/value store that goes the extra mile[J/OL]. Linux Magazine. http://www. linux-mag. com/id/7496/.

[10] RUDOFF A. Persistent Memory Programming[J]. Login:The Usenix Magazine,2017,42(2):34-40.

[11] PAGH R,RODLER F F. Cuckoo hashing[J]. Journal of Algorithms,2004,51(2):122-144.

[12] HERLIHY M,SHAVIT N,TZAFRIR M. Hopscotch hashing [C]∥International Symposium on Distributed Computing. Berlin:Springer,2008:350-364.

[13] MOHAN C,HADERLE D,LINDSAY B,et al. ARIES:a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging[J]. ACM Transactions on Database Systems (TODS),1992,17(1):94-162.

[14] COOPER B F,SILBERSTEIN A,TAM E,et al. Benchmarking cloud serving systems with YCSB[C]∥Proceedings of the 1st ACM symposium on Cloud computing. New York:ACM,2010:143-154.

[15] REINDERS J. VTune performance analyzer essentials[M]. Intel Press,2005:112-135.