针对持久内存的动态哈希方案综述

邹亦舸 [1)]

[1)](华中科技大学 武汉光电国家研究中心, 武汉 430074)

摘 要 具有大容量、非易失性、DRAM 级读写性能的持久内存（PM）有望在新一代存储系统中取代传统的 DRAM 内存。动态哈希索引得益于其扁平化的结构，具有很高的查询效率，并且其动态调整大小的特性可以满足更多使用场景，因此成为广泛使用的索引结构。然而，由于 PM 的特性，以及其读写性能与 DRAM 相比仍有劣势[1]，在 PM 上实现动态哈希面临着许多挑战。本文调研了多个针对 PM 的动态哈希方案，它们通过精心设计的哈希索引结构尝试克服 PM 的性能瓶颈。Zuo [2]等人提出了 Level Hashing，设计了一种两层结构的哈希方案。Chen [3]等人在 Level Hashing 的基础上提出 Clevel Hashing，设计了一种无锁并发控制机制，提升了并发能力。CCEH [4]是可扩展哈希（EH）在 PM 上的改进版本，能支持更小粒度的扩容，并且设计上考虑了 CPU Cache 块，以提升索引结构的性能。Dash [5]则改进了 CCEH，以实现更好的高并发性能。Wang [6]等人认为现有的动态哈希方案都无法实现性能效率与性能可预测性的两全，且性能可扩展性有限，因此他们提出了性能可扩展的、高效的、可预测的哈希（SEPH）。

关键词 持久内存；索引；哈希方案；哈希表；重哈希；并发控制

A Review of Hashing Schemes on Persistent Memory

Yige Zou[1)]

[1)](Wuhan National Laboratory for Optoelectronic, Huazhong University of Science and Technology, Wuhan 430074)

Abstract Persistent memory (PM) with large-capacity, non-volatile, DRAM-level R/W performance is expected to replace traditional DRAM memory in next-generation storage systems. Dynamic hash index benefits from its flat structure with high search efficiency and their dynamic resizing feature to meet more usage scenarios, thus becoming a widely used index structure. However, due to the characteristics of PM and the fact that its R/W performance is still inferior compared to DRAM [1], implementing dynamic hashing on PM faces many challenges. In this paper, we investigate several dynamic hashing schemes for PM, which try to overcome the performance bottleneck of PM by means of well-designed hash index structures. Zuo [2] et al. propose Level Hashing, which designs a two-level structured hash table. Chen [3] et al. propose Clevel Hashing on the basis of Level Hashing, which designs a lock-free concurrency control mechanism to improve the concurrency capability. CCEH [4] is an improved version of Extendible Hashing (EH) on PM, which can support smaller granularity resizing and is designed to improve the performance of operations by taking into account the CPU Cacheline. Dash [5] improves CCEH for better high concurrency performance. Wang [6] et al. argue that all of the existing dynamic hashing schemes are unable to achieve both performance efficiency and performance predictability and have limited performance scalability, so they propose Scalable, Efficient, and Predictable Hashing (SEPH).

Key words Persistent Memory; Index; Hashing Scheme; Hash Table; Rehashing; Concurrency Control

1 引言

1.1 持久内存

数据中心要求内存具有大容量、低延迟、易恢复的特性，传统 DRAM 内存已很难满足现代数据中心的需求。持久内存（PM），例如 ReRAM、PCM 以及 STT-RAM，具有高密度、非易失性以及 DRAM 级的读写带宽和延迟，有望在新一代存储系统中取代传统的 DRAM 内存。2019 年，Intel 推出了首款商业化的 PM 产品——傲腾 DCPMM，然而，其读写性能与 DRAM 相比仍有差距。表 1 展示了 DRAM 与傲腾 DCPMM 的读写性能比较。其中，PM 的随机读延迟（DRAM 的 3 倍）和写带宽（DRAM 的 1/6）问题尤为突出，成为潜在的性能瓶颈。

表 1 DRAM 与 PM（傲腾 DCPMM）的读写性能比较[1]

	DRAM	PM	PM / DRAM
连续读延迟（ns）	81	169	208.64%
随机读延迟（ns）	101	305	301.98%
写延迟（ns）	57	62	108.77%
读带宽（GB/s）	105.6	37.6	35.61%
写带宽（GB/S）	76.8	12.5	16.28%

1.2 针对持久内存的哈希索引

索引结构是存储系统中实现高效查询的关键。由于内存架构和特性的显著变化，传统 DRAM 索引结构因忽略了 PM 的设备属性和一致性要求而无法直接适用于 PM 设备。为了更好激发 PM 的性能潜力，已有许多研究为 PM 设备开发了高性能的索引结构，其中包括了许多基于树的索引，例如 CDDS B-tree，NV-Tree，wB+-Tree，FP-Tree，WORT 以及 FAST&FAIR。基于树的索引结构一般具有 O(log(N))规模的查询复杂度，其中 N 为数据结构的大小。

而得益于扁平化的结构，基于哈希的索引可以提供常数级别的查询复杂度，因而被更广泛地使用在内存系统中。哈希索引一般可分为两类：静态哈希和动态哈希。静态哈希会预估并一次性分配足够的内存空间，但在许多应用场景下，哈希表所需的内存空间无法精确预估，导致其存在哈希冲突、溢出或利用率低的问题。动态哈希可以在维护哈希表的过程中动态调整大小，即在负载因子（桶利用率）变高时扩容，负载因子降低时缩小，以保证查询性能和空间利用率的最优。在扩容的过程中，动态哈希会新建一个合适大小的哈希表，将原哈希表中的 KV 项重哈希到新的表中。

然而，在持久内存中维护动态哈希索引存在着诸多挑战，例如扩容过程中的重哈希导致的性能劣化以及一致性保证带来的高开支。本文将介绍多个针对 PM 的动态哈希方案，并阐释它们如何通过精心设计的哈希索引结构克服 PM 的性能瓶颈。

2 相关研究

2.1 基于层次的哈希方案

2.1.1 Level Hashing

为克服 PM 的性能瓶颈、解决扩容导致的性能劣化，Zuo [2]等人提出了 Level Hashing。如图 1，该哈希方案引入了一种基于共享的双层结构，上层中的桶数量是下层桶数量的两倍，且两个相邻的上层桶共享一个下层桶，下层桶作为上层桶的备用桶，即当某个上层桶的槽被使用完后，KV 项会被存储到该上层桶拥有的下层桶中。Level Hashing 维护两个哈希函数，$Hash_1(x)$和 $Hash_2(x)$，可将一个 KV 项关联至两个不同的上层桶，因此每个 KV 项在上层和下层中各有两个候选桶。在该哈希表结构中，每次插入/删除/更新/查询操作都仅需最多探测 4 个桶，因而在最坏和平均情况下都具有常数级别的复杂度。

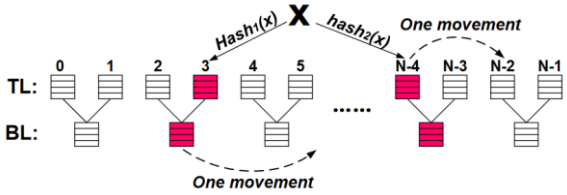
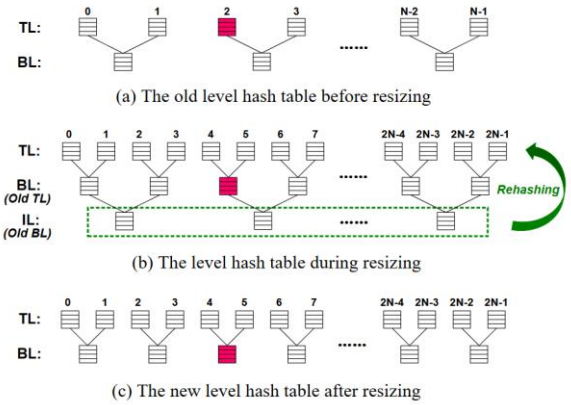


图 1 Level Hashing 的哈希表结构（图中 TL 代表上层，BL 代表下层）

当插入一个 KV 项，且该 KV 项的所有候选桶都已满时，Level Hashing 会进行哈希表扩容。Level Hashing 会生成一个新的上层，其桶数量为原来上层桶数量的两倍，将原下层中各桶中的 KV 项重哈希到新上层中，就完成了一次扩容。此时新上层和原上层形成了新的哈希表，原上层成为了新哈希表的下层。因此，Level Hashing 每次扩容只需重哈希下层，即 1/3 个哈希表，无需将整个哈希表重哈希，因此减少了重哈希带来的写操作，使哈希表的扩容拥有更高的效率。



(a) The old level hash table before resizing

(b) The level hash table during resizing

(c) The new level hash table after resizing

图 2　Level Hashing 的扩容过程

2.1.2　Clevel Hashing

虽然 Level Hashing 实现了高效率的扩容操作，但是它在扩容时需要锁定整个哈希表，并阻塞其他线程的对哈希表的正常访问和操作。为解决这个问题，Clevel Hashing [3]在 Level Hashing 的基础上做出改进，提出了一个崩溃一致的无锁并发方案。在 Clevel Hashing 中，触发扩容操作的插入线程仅生成新的顶层，以在新分配的层中插入 KV 项，而剩余的重哈希操作被延迟并交给一个专用的后台线程完成，从而在无需加锁的情况写实现扩容时的并发访问。

2.2　基于可扩展哈希（EH）的哈希方案

2.2.1　CCEH

考虑 Cache 块的可扩展哈希（CCEH）[4]是可扩展哈希（一种用于传统 DRAM 内存的哈希方案）的变体，它提出了一种针对 PM 的可扩展的哈希表结构（如图 3）。为提升读写性能，CCEH 将桶大小设置为 CPU Cache 块的大小，并且引入了一种称为段的中间粒度，每个段由固定数量的桶组成。CCEH 维护一个目录，相同段内的桶会被索引至相同的目录项，这样目录会被极大地压缩，有更大的可能存在于 CPU Cache 中。

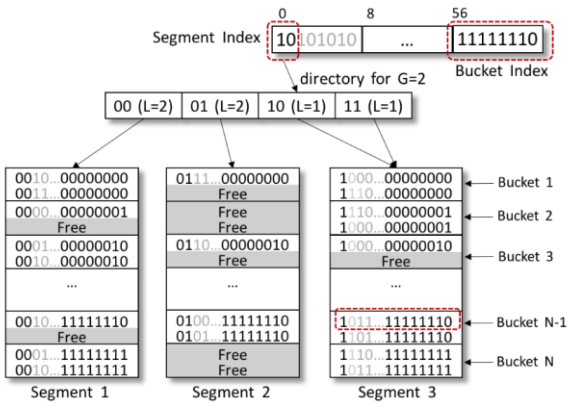
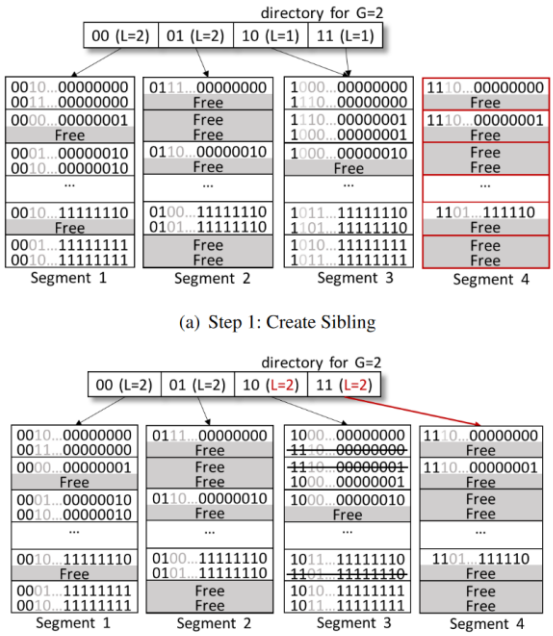


图 3　CCEH 的哈希表结构

CCEH 取键哈希值的最高若干（取决于哈希表中段的数量）位作为段索引，取最低若干（取决于段中桶的数量）位作为桶索引。为进一步提升负载因子，CCEH 采用线性探测法处理哈希冲突，使 KV 项可以放在它所属的桶（由桶索引得出）的后面连续的几个桶中。

当新插入的 KV 项所有的候选桶（索引桶以及可以被线性探测到的所有桶）全满时，CCEH 会以分裂冲突段的方式进行扩容。如图 4，CCEH 首先生成一个新段，根据冲突段中 KV 项的段索引保留新段所需的 KV 项，并根据桶索引将这些 KV 项重哈希到新段相应的桶中。接着更新段索引的目录，冲突段便完成了分裂。为了防止引入多余的 PM 写，冲突段中被重哈希至新段的 KV 项不会被立即删除，而是在后续的插入操作中被覆盖。



(a) Step 1: Create Sibling

(b) Step 2: Split and Lazy Deletion

图 4　CCEH 的段分裂式扩容

CCEH 的段分裂允许哈希表实现以段为粒度的增量扩容，而无需以整个哈希表为粒度扩容，因此避免了对全表的重哈希，减少了单次扩容的开销。

2.2.2　Dash-EH

为进一步提升 PM 哈希索引的性能可扩展性（即高并发请求下的性能），Lu [5]等人提出了动态且性能可扩展的哈希（Dash），其对 DRAM 上的两种经典散列方案（可扩展散列（EH）和线性散列（LH））进行了多项改进，其中针对 EH 的 Dash 继承了 CCEH 的大部分设计，但将存储桶大小设置为傲腾 DCPMM 的 XPLine 大小（即 256 字节），

以获得更好的定局部性。如图 5，Dash-EH 将每个桶的内部空间分为记录区（224 字节）和元数据区（32 字节），前者保存指向 KV 项的指针，以支持可变长度的键和值，后者则专门用于优化哈希表的读写性能和空间利用率。

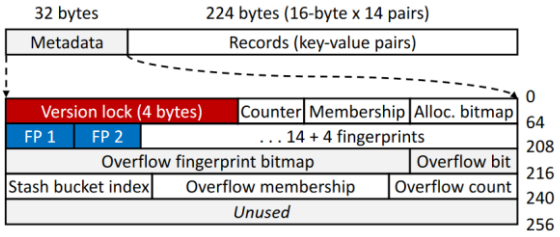


图 5 Dash-EH 的桶内部结构

为减少线性探测和唯一性检查所需的指针解引，对于每个 KV 项，Dash-EH 都会在元数据区中保留键哈希值的第二低字节作为指纹；此外，Dash-EH 还采用了乐观并发控制，以避免在搜索 KV 项时锁定整个段，使得哈希表在应对高并发请求时具有更好的性能。另一方面，Dash-EH 结合了多种技术来提高负载因子，如多探测一个桶、平衡候选桶的负载因子、允许 KV 项在索引桶和线性探测桶之间移动一次，以及在每个段中添加几个隐藏桶来容纳冲突的 KV 项。

2.3 SEPH

Wang [6]等人认为，现有的两类 PM 哈希方案，即基于层次的哈希和基于 EH 的哈希都无法同时满足高性能效率（即平均性能）和高性能可预测性（即高百分位性能），并且都不具有良好的性能可扩展性（高并发性能），因此他们提出了性能可扩展的、高效的、可预测的哈希（SEPH）。SEPH 结合了两类哈希方案的优点，提出了一种同时具有层次和段的哈希表结构，如图 6。

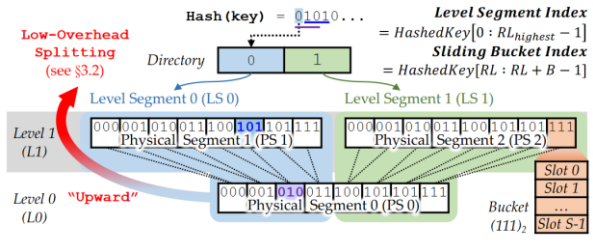


图 6 SEPH 的哈希表结构

SEPH 构造的哈希表中有两种数据结构，物理段 PS 和层次段 LS。物理段是 PM 中固定大小的存储空间，每个物理段包含固定数量（N 个）的桶，每个桶中又包含固定数量的槽。物理段之间具有层次结构，半个低层物理段和一个高层物理段组成一个层次段。与 Level Hashing 一样，高层物理段中的两个相邻的桶共享一个低层桶，但不同的是，KV 项会优先存储在低层候选桶中，低层桶满了才会存储在高层候选桶中。每个物理段所在的层级称为驻留等级 RL，最低层的 RL 为 0。SEPH 维护一个哈希函数 Hash(x)，类似于 CCEH，SEPH 可通过 Hash(key)直接获得段索引和桶索引。具体地说，取 Hash(key)[0 : $RL_{highest}$ − 1]为层次段索引，取 Hash(key)[RL : RL + B - 1]为桶索引，其中 B = $\log_2$N，N 为物理段中的桶数量。

SEPH 设计了一种名为 1/3 分裂的扩容方案，每次扩容仅需重哈希 1/3 个层次段。如图 7，在扩容时，SEPH 首先在发生冲突的层次段上方分配两个新的物理段，将冲突层次段中的低层桶内的 KV 项重哈希至新分配的两个物理段中，此时新的两个物理段和原来的高层物理段便组成了两个新的层次段。SEPH 利用段结构减小了扩容的粒度，利用层次结构减少扩容时的重哈希操作，因此极大地减少了扩容开销。

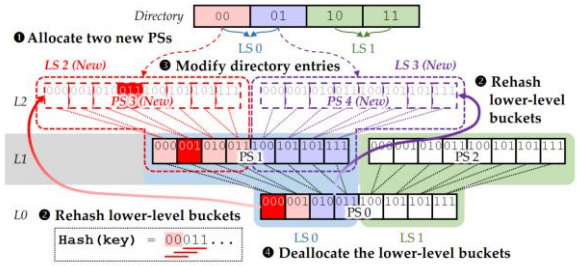


图 7 SEPH 的扩容过程

哈希索引通常在槽中储存指向 KV 项的指针以存储可变长度的 KV 项，重哈希时需要 key，因此涉及到指针解引，会造成大量的随机读，加重扩容的开销。SEPH 设计了一种无需指针解引的重哈希机制，称为桶索引预测器。如图 8，SEPH 将每个槽设置为 64bit，其中最高的 16 位作为桶索引预测器，它维护着 Hash(key)中未来会被用作桶索引的 bit。每次层次段分裂时，KV 项被重哈希至高两层的物理段中，因此仅需从桶索引预测器中向后取 2 个 bit 即可得到新的桶索引（这样就完成了一次重哈希）。

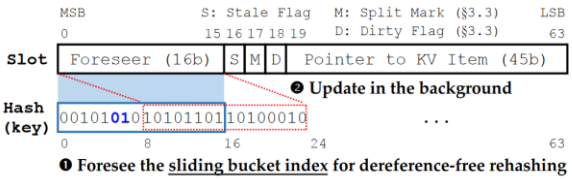


图 8 SEPH 的槽结构以及桶索引预测器

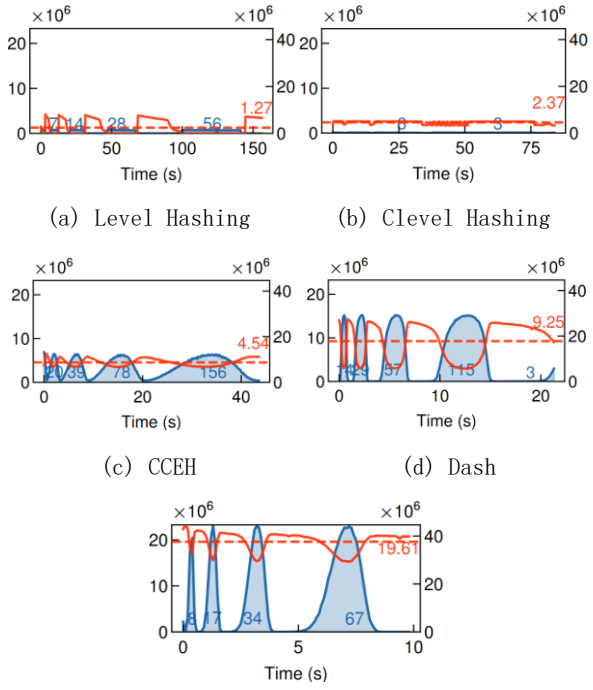
为了获得更好的性能可扩展性，SEPH 采取了一

种半无锁并发机制，即对频繁的操作（插入、删除、更新、查询）采用无锁机制以避免锁机制导致的写放大，对不频繁的操作（扩容）采用锁机制以保证正确性。

3 方案评估

3.1 平均性能与高百分位性能

在混合负载（50%查找，50%插入）下，对第 2 节中的 5 种哈希方案测试实时吞吐量和扩容开支。如图 9，红色曲线为实时吞吐量，红色虚线为平均吞吐量，蓝色曲线为重哈希次数。在处理冲突时，基于 EH 的哈希采用连续读，基于层次的哈希采用随机读，由于 PM 连续读比随机读具有更低的延迟[1]，因此基于 EH 的哈希具有更高的平均性能。对比 c 和 d，Dash 的平均性能高于 CCEH，但由于其更高的扩容开销，高百分位性能比 CCEH 差。此外，SEPH 借助其低开销的扩容方案，以及无指针解引的重哈希，实现了最高的平均性能和高百分位性能。



(a) Level Hashing　(b) Clevel Hashing

(c) CCEH　(d) Dash

(e) SEPH

图 9　混合负载下的实时吞吐量和重哈希次数

3.2 高并发性能

在混合负载（50%查找，50%插入）下，逐渐增加并发线程量，测试各哈希方案的平均吞吐量以及总数据写入量。如图 10 所示，当线程数量超过 24 时，Level Hashing、Clevel Hashing、CCEH、Dash 的插入、更新、删除操作的平均吞吐量都提升较少，而 SEPH 却依然具有较大的提升，这是因为前四种哈希方案的并发控制机制都存在严重的写放大问题，额外引入的大量 PM 写成为了高并发态下的性能瓶颈；而 SEPH 的半无锁并发控制机制极大地减少了一致性保证所需的 PM 写。对查询操作，除 Level Hashing 以外的哈希方案都能在并发线程量增长的过程中有着较好的性能提升，因为查询操作是非互斥操作，同时发生的查询不会导致数据不一致，因此不需要引入额外 PM 写来保证一致性。

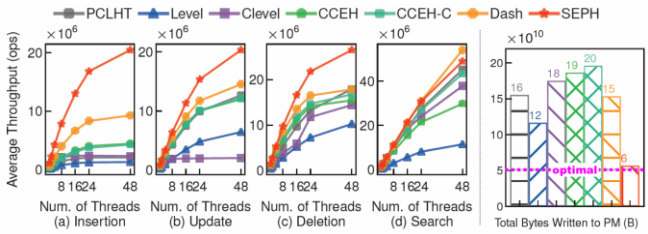


图 10　不同并发线程量下的平均吞吐量以及总数据写入量

4 总结

由于 PM 与 DRAM 读写性能的差距，维护哈希表过程中产生的额外读操作和写操作会成为 PM 哈希索引的瓶颈，主要体现在 1）扩容时大量的重哈希导致的高开销 2）并发控制需求以及其带来的写放大问题。

为提升 PM 哈希的性能，当前的研究成果在以下方面做出了许多努力：1）优化哈希表结构，使哈希表的基本操作保持常数规模的复杂度 2）优化发生冲突时的探测过程，减少探测所需的 PM 读开销 3）设计低开销的扩容方案，减小扩容的粒度，减少扩容时产生的重哈希次数 4）使用精心设计的并发控制机制，减少保证一致性所需的额外 PM 写，以提升性能可扩展性。

参 考 文 献

[1] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In 18th USENIX Conference on File and Storage Technologies (FAST 20), Santa Clara, CA, USA, 2020: 169–182.

[2] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 461–476, Carlsbad, CA, October

2018. USENIX Association.

[3] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 799–812. USENIX Association, July 2020.

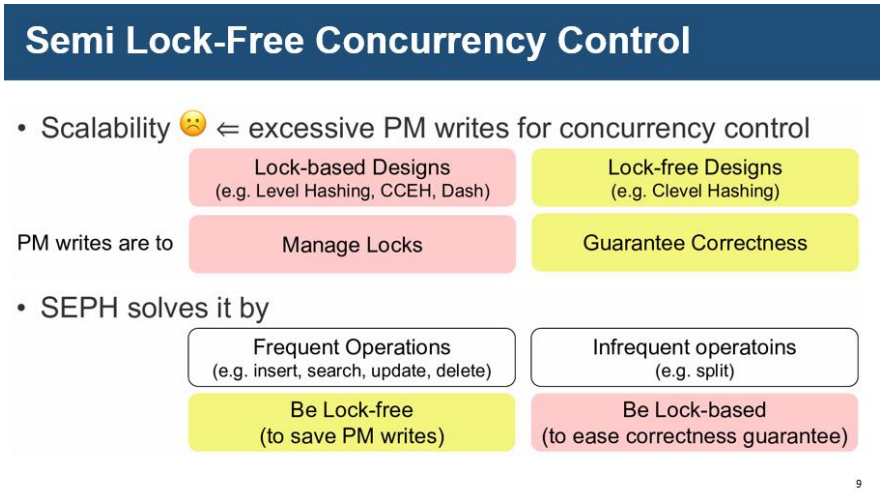
[4] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In 17th USENIX Conference on File and Storage Technologies (FAST 19), pages 31–44, Boston, MA, February 2019. USENIX Association.

[5] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. Proc. VLDB Endow., 13(8):1147–1161, April 2020.

[6] Chao Wang, Juniang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory. Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). Boston, MA, USA, 2023: 479-495

附录. Presentation时提出的问题

针对slides中的这张图，在YCSB中如何体现frequent和infrequent？是CRUD操作占的多还是split占的多？



回答：YCSB是一种数据库和索引性能测试工具，其提供多种工作负载模板，例如：

读写均衡：50%查询 50%更新，适用于会话存储等场景。

读多写少:，95%查询 5%更新，适用于照片标签等场景。

只读：100%查询，适用于用户资料缓存等场景。

读取最新数据:，95%查询 5%插入，适用于用户状态更新等场景。

此外，测试者还可以根据适用的场景自定义工作负载。在SEPH中，作者在多种工作负载下都进行了测试。

作者将CRUD四种基本操作定义为frequent，将split定义为infrequent，操作是否高频与操作在YCSB负载中的占比无关。split不会主动发生，而是由插入操作引起的，仅有极少数的插入会导致split，因此无论使用何种YCSB负载，split相比于CRUD的总数来说都是极少的。如下图所示，在 200,000,000 个CRUD操作(一半插入一半查询)的负载下，仅有少于 5 次split/resizing(图中一个蓝色峰代表一次split/resizing)。