避免键值存储中的线程停顿与切换：全新的无闩锁技术及更多内容

David Lomet
美国华盛顿州雷德蒙德
美国华盛顿州雷德蒙德
lomet@microsoft.com

Rui Wang
微软公司
wanrui@microsoft.com

摘要

键值存储实现高性能的一大障碍是线程切换或停顿的高昂成本。这种情况有许多成因，其中一个主要原因是资源争用。随着负载增加，冲突操作更频繁地尝试并发访问数据，这一成本也随之上升。传统的基于闩锁的方法通常通过阻塞一个或多个争用线程来处理这些情况。无闩锁技术可以避免这种行为。但如果无闩锁技术需要执行无用工作，其收益可能有限。本文展示了无闩锁技术如何利用增量记录更新，并通过一种新的无闩锁方法——通知机制，大幅减少无用工作。本文阐释了通知机制的工作原理，以及它如何解决B树索引维护问题，同时避免线程切换或停顿。本文还讨论了其他避免线程切换或停顿的可能途径。

CCS概念

• 信息系统 → 数据访问方法

关键词

无闩锁更新、线程切换、停顿、键值存储、索引树结构修改操作

ACM参考文献格式：

David Lomet 和 Rui Wang，2026。避免线程停顿和切换：新的无闩锁技术及其他。在 第*16*届创新数据系统研究年会论文集（*CIDR'26*）。*ACM*，纽约，纽约州，美国，*6*页。 https://doi.org/10.1145/1234567890

1 引言

1.1 概述

我们希望避免数据库内核中的线程停顿和线程切换，因为它们会对性能产生不利影响。通过对缓存页进行增量更新，即将一个增量（更新后的记录）前置到之前的页面状态，我们可以确保更新缓存中的页时无需切换线程。当

允许出于个人使用或课堂教学目的，免费制作本作品部分或全部内容的数字版或纸质版副本，但前提是不得为营利或商业利益制作或分发副本，且副本首页须附有此声明和完整引文。本作品中第三方组件的版权必须得到尊重。如有其他用途，请联系所有者/作者。https://doi.org/10.1145/1234567890

通过比较并交换（CAS）指令完成前置操作后，这就成为一种无闩锁更新，可与对同一页的读取和更新并发执行。我们在微软采用了这种方法：在 SQL Server 的 Hekaton 主内存数据库中使用 Bw-tree [1]，并在 Cosmos DB [9] 中将其用于实时索引文档，也就是说，文档一经添加就会被索引。结合适当的无闩锁基础设施，在页上进行并发操作时，读取和更新都无需等待。

使用闩锁在更新期间保护页面状态，会阻止并发更新和并发读取。如果这些阻塞持续时间较短，那么损失不大，因为短暂的闩锁自旋影响有限。但在负载较重时，阻止读取和更新的闩锁会严重影响性能。在这种情况下，通常会使用等待线程队列。此时，线程会让出底层处理器，转而执行另一项任务。线程切换会增加额外指令开销，并导致缓存局部性丢失。无闩锁技术通过让单个线程持续掌控执行，缩短代码路径长度，并通过沿用现有且频繁驻留在缓存中的指令路径来提高处理器效率。

即使页面不在页面缓存中，增量更新也可以避免一次 I/O，因为更新（盲更新）不需要缓存中存在先前的页面状态。增量更新可以附加到部分页面状态之后，甚至可以附加到完全不在页面缓存中的页面状态之后。不过，读取缓存未命中的记录时可能需要该状态。此时就需要一次读取输入/输出。增量更新可以延迟这次读取，并将其摊销到多次更新中。最终，由多个部分页面组成的页面需要合并，但可以延迟进行。

更新后的页面需要不时地从缓存中逐出，并写入辅助存储（我们假设为闪存）。我们可以采用日志结构化存储策略 [8]，将页面批量写入大型缓冲区，从而减少写入输入/输出的次数。这也会降低线程切换和输入/输出路径执行的频率，因为大多数页面"写入"都针对一个大型的多页 I/O缓冲区。读取输入/输出则更难避免。如果记录不在缓存中，就必须从存储中读取其所在的页面。

1.2 无闩锁难题

使用比较并交换（CAS）将一条记录增量更新前置到缓存页中非常简单，性能也很高，而且对其他操作的性能影响极小，即使这些操作访问的是同一页也是如此。然而，随着越来越多的记录增量不断更新该页，通常更为常见的页面读取操作性能会逐渐下降，因为搜索线性列表远比搜索经过组织、便于查找的

页要慢，例如通过二分查找或页内索引进行查找 [4]。因此，最终我们仍会面临这一常见读取操作的性能下降问题；而避免热点资源上的线程切换又会妨碍可扩展性。这是一种权衡，因为系统既需要高性能，也需要可扩展性。

那么，我们该如何重新组织更新，才能在采用无闩锁方法的同时，让系统保持高性能和高可扩展性呢？我们将在第 2 节介绍这一主题。重要的是，我们将介绍 NOTICE，它能大幅降低重新组织的成本。第 3 节和第 4 节讨论如何将 NOTICE 的理念应用于常见 B树形式的结构修改操作（SMO [7]）。第 5 节则以更抽象的方式概述如何更普遍地使用 NOTICE。后续各节将介绍更多系统性能技术，并探讨其他相关工作。最后，我们将总结全文。

2. 页面更新的无闩锁并发

2.1 闩锁

使用闩锁通常能让多线程编程更容易正确实现，因为它可以在容易出问题的临界区排除多线程操作，避免竞态条件破坏状态。例如，如果某个线程正在更新 B树节点，设置一个闩锁（测试并设置）以阻止其他线程访问临界区，就能确保其他访问者（读者或写入者）不会看到处于过渡状态的节点。遗憾的是，随着线程数量增加，这种基于闩锁的临界区方法会越来越妨碍线程的高效利用。基本上，线程要么空闲等待（通常是自旋等待），直到闩锁释放；要么切换到其他可执行的任务。但线程切换代价很高，既会增加指令数量，也会影响处理器缓存命中率。这两种策略会以不同组合使用，但都会妨碍线程计算能力得到最大程度的发挥。在高争用工作负载下，随着线程不断遇到被闩锁保护的节点，闩锁机制会严重限制多线程可扩展性。

2.2 无闩锁基础

无闩锁技术通常会频繁使用比较并交换（CAS）指令，从而避免闩锁所带来的线程干扰，并有望更充分地发挥线程处理能力。由于访问数据不会受到阻碍，因此在更改状态时必须格外谨慎。在简单情况下，会先离线创建一个新状态，然后通过 CAS 原子地将指针从旧状态切换到新状态，以安装新状态。至少在一段时间内，新旧状态都必须可访问；只有在任何活动线程都无法再访问旧状态后，才能将其丢弃。这就需要构建相应的基础设施来处理这一问题，例如使用纪元机制跟踪活动线程是否仍在旧状态的某个部分上执行。要以高性能正确实现这一机制并非易事。但完成后，[1], 它对线程执行成本的影响就微乎其微。

2.3 无闩锁增量更新

Bw-tree 节点的增量更新展示了无闩锁方法可以多么简单高效，同时也揭示了其中的复杂之处。所有节点都通过映射表访问，表索引用于标识节点，而表项则指向节点状态。对节点的引用始终通过映射表进行。

更新节点需要以原子方式将旧节点替换为新节点。这可能代价高昂。但对于简单的增量更新，这很容易实现。我们在离线阶段构造一个更新“增量”，其中通常包含一条记录，以及要应用于该页、涉及这条记录的操作。该增量指向旧状态。当安装这个新增量时，更新便会发生：通过一次比较并交换，将映射表中该节点的旧状态指针替换为指向更新增量的指针。如前所述，新的更新增量随后会指向先前的节点状态。从逻辑上说，这会将节点状态更改为包含该更新增量的效果。如果比较并交换成功，这次增量更新就会立即生效并引发状态变化。

然而，另一个线程发起的竞争更新有时可能抢先一步，先安装自己的更新。用于在节点映射表条目中安装新状态的比较并交换操作，会决定哪个线程赢得竞争。失败的线程随后会重复这一过程，这次使用获胜线程操作后产生的更新状态。这种方法的巨大优势在于，新状态与旧状态共享内容，从而最大限度地降低执行更新的线程所承担的开销。事实上，这比对页加闩锁，然后在页内移动记录以容纳新更新，要高效得多。此外，没有线程会被阻塞。

2.4 较大的节点状态变更

不过，线程访问经过增量更新的节点时会产生额外开销。现在，除了搜索我们最初采用的读优化状态（称为“基础状态”或“基础节点”）之外，还需要搜索增量列表。随着增量更新列表不断增长，这项开销也会随之增加。

因此，我们需要定期将增量更新列表并入节点的读优化基础状态。我们最初的尝试如下：在一旁构建一个新状态，将基础状态与增量更新的影响合并，从而生成新的优化基础状态。然后，通过比较并交换，将指向这个合并状态的指针安装到该节点在映射表中的条目里。这里的逻辑很简单：无需闩锁，我们就能转换节点状态，让访问效率更高。不过，这样做会带来性能损失。如果多个线程尝试安装合并状态，最终只有一个线程会通过比较并交换获胜，这正是我们想要的结果。但失败的线程也要付出合并的开销，而这笔开销可能相当可观。我们希望避免这项额外开销。

2.5. 避免浪费节点合并

这里我们介绍 Rui Wang [10] 提出的一种思路：让线程竞相安装一个增量。获胜线程安装的增量会声明由该线程负责合并状态。因此，比较并交换的竞争是为了安装增量，而不是安装合并状态。这意味着落败的线程只会损失生成并尝试安装这个“合并”增量所付出的工作。赢得比较并交换竞争并安装该增量的线程，随后会独自

构建并安装合并状态。我们将这种增量称为通知。在 Bw-tree 的其他改动中，我们也会使用通知。线程将增量与基础状态合并时，会先在节点的状态前置入一个 cNOTICE。“cNOTICE”中的“c”表示该通知的用途是将更新增量与基础页合并，从而生成新的基础页。我们也会对其他通知采用这种命名约定。

构造并前置一个 cNOTICE，与生成合并状态相比轻而易举。一旦发布，它就能让其他线程免于承担生成合并状态的开销。只有抢先发布 cNOTICE 的线程才需要承担合并的开销。未能抢先的线程只需承担尝试发布 cNOTICE 但失败的开销。由于这些其他线程通常正在执行更新，它们可以通过将自己的更新前置到 cNOTICE 来继续执行。只有 cNOTICE 通知竞争中的“赢家”才需要承担节点合并的开销，而且合并所针对的状态保证不会被其他线程更改。

我们在图1中展示了节点合并过程中涉及的状态序列。在图1(a)中，cNOTICE 已发布，保护先前的状态（此处包括旧基础页和一些更新增量）不再发生变化。在图1(b)中，发布 cNOTICE 的线程生成了一个新的基础页，其中现在包含先前发布的更新增量。在此次合并进行期间，其他线程仍可继续发布其更新增量。



图1：使用 cNOTICE 合并一个页。安装 cNOTICE 的线程是唯一执行合并的线程，其他线程则继续读取和更新节点。

为确保胜出者的更新先于落败者的更新发布，胜出者可以构建一个双增量列表，其中包含 cNOTICE 和其更新，并将其前置到节点状态中。这样，cNOTICE 便能保护包含胜出者更新的节点状态，并最终将该更新合并到新的基础页中。

2.6 结构修改操作

规模较大的多节点变更需要在执行比较并交换之前付出一些成本。这些多节点变更就是结构修改操作（SMO）[7]。我们的目标是尽量减少比较并交换前的工作，因为对于比较并交换失败的线程来说，这些工作会白费。不过，我们希望这些比较并交换失败的线程能够顺畅地继续执行。因此，竞争中的线程需要准备好比较并交换后的状态，以便失败的线程能够继续高效执行。失败线程所失去的，正是这条供其继续执行的比较并交换后延续路径。

不过，与合并一样，结构修改操作中开销最大的部分是替换由 NOTICE 保护的状态，而这项工作只有比较并交换成功的线程才会执行。

3. 无闩锁节点分裂

在最初的 Bw-tree [1] 中，我们采用了 B-link 树风格的节点分裂。这使节点分裂可以通过两个独立的原子操作完成：一个用于分裂节点，另一个用于为新节点发布索引项。对于 Bw-tree，分裂同样分为两个原子操作完成，这与我们之前执行合并的方式相同。我们用旧节点中一半的条目构建新节点的状态，将这些条目从旧节点复制到新节点，然后尝试使用比较并交换安装一个分裂增量。多个线程可能会尝试执行分裂，这意味着比较并交换操作的落败者也执行了构建新节点的工作。随后，按照 B-link 的第二个原子操作，使用第 2 节所述的无闩锁节点更新，在父索引节点中为新分裂出的节点发布一个索引项。

和之前一样，我们希望利用通知来减少线程在比较并交换之前所执行的工作。在本例中，我们利用分裂通知（sNOTICE）来分裂页。赢得比较并交换的线程负责完成大部分节点分裂工作。但为了确保比较并交换失败的线程能够继续执行，仍有一些工作必须完成，通常还需要这些线程执行额外的更新。我们的目标是在确保 sNOTICE 之前的旧节点状态不会被比较并交换胜出线程以外的线程修改的同时，尽量减少比较并交换之前的工作量。图2对此进行了说明。这项工作使失败线程甚至能够在半分裂完成之前继续执行。我们不包括 B-link树分裂中的父索引节点更新部分；该部分通过对父索引节点执行增量更新来完成，并且可以在节点分裂之后执行。

我们还需要考虑一个与存储相关的问题。我们希望在分裂变得持久化时，新节点 N 已填入其应有的记录，并且在旧节点 O 以未将记录移至 N 的稳定状态出现之前完成这一点。否则，如果系统崩溃，O 中的记录可能会在稳定状态下丢失。下面将介绍 Deuteronomy [1] 如何利用其日志结构化缓冲区来实现这一目标。

1. 对于新节点 N，我们只在映射表中创建一个槽位。我们会在用于将数据发送到持久化存储的 I/O 缓冲区中分配空间。该空间将容纳 N 和 O 的数据状态。但不会移动任何数据。在 I/O 缓冲区中分配这块空间，可确保系统崩溃时，此次分裂要么完全持久化，要么完全不持久化。直到分裂器释放对缓冲区的占用后，缓冲区才会写入存储。此外，在 O 或 N 收到后续增量更新之前，O 分裂前的状态（现已与 N 分开）将位于日志结构化存储缓冲区中。

2. sNOTICE 包含用于定义分裂的键、N 和 O 两个映射表条目的指针，以及 I/O 缓冲区中为 O 和 N 分配的空间位置。sNOTICE 会在不发生竞争的情况下发布到 N 的映射表条目中。用于定义分裂的键将成为 O 的新侧链接键。

3. 我们使用比较并交换（CAS）将此 sNOTICE 安装到 O 上。图2展示了这种情况。如果 CAS 失败，我们只会损失上述步骤 1 和 2 中的工作。数据尚未移动，而数据移动是分裂执行成本的主要部分。此时，O 和 N 上都已设置 sNOTICE，因此 O 中位于 sNOTICE 以下的数据可以由两者共享。后续对 O 的访问会遇到 sNOTICE，并根据 sNOTICE 的分裂键将访问导向 O 或 N。对 N 或 O 的访问都会“正常”执行，但在 N 完全实例化且 O 中现已属于 N 的条目被清除之前，二者会共享受 sNOTICE 保护的状态。

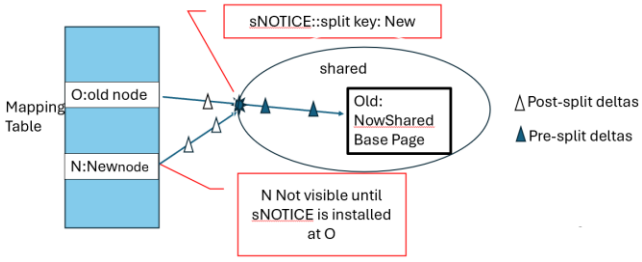


图2：步骤2中的节点分裂——简化示意。圈出的部分是尚未分裂的节点（原来的旧节点），现在由一个 sNOTICE 保护。它是只读的，并由新旧节点共享。根据键值，更新可以继续在 O 节点或 N 节点上进行。

4. sNOTICE 比较并交换的胜出者将 N 的数据（键大于 sNOTICE 分裂键的数据）从 O 移至 N 的分配的缓冲区空间。接着，将 N 的前置增量连接到缓冲区中的 N 状态，从而结束共享安排。N 的侧链接指针被设置为 O 的侧链接指针。

5. 共享状态中小于分裂键的记录现在被移至缓冲区中的新空间。现在，我们将 O 中的 sNOTICE 替换为指向 O 新形成版本的指针。sNOTICE 中的分裂键成为 O 的侧链接指针。至此，我们完成了所谓的半分裂，即将数据分到两个节点中。

6. 最后一步是更新父索引节点，加入一个索引项，根据分裂键将搜索从父节点引导到其下方相应的节点。这是 B-link 树分裂的第二步。如果系统在完成此步骤之前崩溃，那么可以在恢复期间或之后完成这部分分裂。

4. 无闩锁节点合并

我们最初发表的 Bw-tree 论文并未正确描述节点合并。通知的存在使合并过程更简单、更易于理解。这里进一步略微限制可合并节点的范围，合并过程就更加简单了。我们只合并那些未通过父索引节点中最低值的索引项引用的节点。合并数据节点会从父索引节点中删除一个索引项。我们并不禁止父索引节点分裂。这会导致一个问题：如果分裂过程选择在节点合并发生的区域分裂父索引节点，就可能出现这种情况。若真如此，分裂后产生的两个索引节点都可能需要访问合并后数据节点所覆盖的部分键空间。合并步骤如下。

1. 向数据节点 M（合并后形成的节点）和 D（将合并到 M 中的节点）的父索引节点 P 发布一条 pNOTICE。它指出相邻索引项所引用的数据节点（M 和 D）对应的键空间，从而也标识出要移除的节点（键顺序较高的节点 D）。通过比较并交换将其发布到 P，以确保 P 不会在新合并的子节点键空间内发生分裂。

2. 现在，父节点在逻辑上不再包含指向 D 的索引项，所有对其数据的搜索都会经由键顺序较低的节点 M 进行。M 包含一个指向 D 的侧指针；如果需要读取或更新 D 中的数据，搜索就会沿着该侧指针继续。与节点分裂的情况正好相反：分裂时，一个节点包含原本分属两个节点的数据；而在这里，两个节点包含原本属于一个节点的数据。接着，我们在 D 处前置一条 dNotice，以防止向 D 前置更多更新；也就是说，确保在 pNOTICE 发布之前、但在 dNotice 前置之后已经流入的、针对 D 的更新不会被发布到 D；相反，这些更新器会被重定向回 M，并可在 M 上发布增量更新。

3. 最后，如图3所示，我们在 M 处发布一个 mNotice，使 M 和 D 的内容都无法再更改，从而可以合并它们的内容。mNotice 与 cNotice 类似，不同之处在于它同时保护 M 和 D。

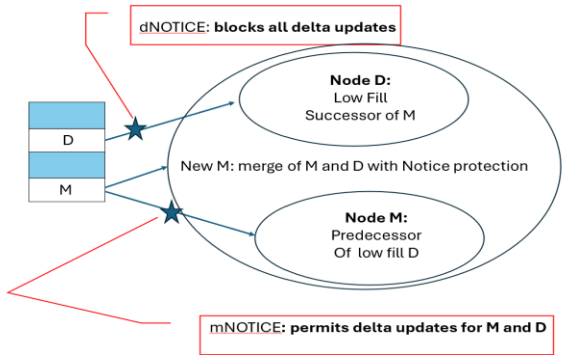


图 3：步骤 3 中的节点合并——简化图。大圆圈所示部分是新的合并节点 M，其中包含发布 mNotice 时 M 和 D 中的所有数据。M 处仍可继续进行更新，但 D 处不能。后续针对 D 的更新也会发布到 M 处。

4. 现在我们可以执行节点合并，也就是对 M 和 D 进行“合并”。这会移除 D 和 dNotice，并用新的 M 替换旧的 M；新的 M 合并了旧的 M 和 D。随后可以通过合并从 P 中移除 pNotice。mNotice 也可以这样移除。

节点合并中最繁重的部分，是将 M 和 D 中的条目整合到一个优化的连续存储区中。和之前一样，这项操作会在放置必要的通知以保护

状态不在合并期间发生变化之后进行。在父节点处竞争发布 pNOTICE 失败的线程可以继续执行读取或更新，并确信获胜的 pNOTICE 线程会执行节点合并。任何正在进行的更新也都会受到通知的保护，或被通知重定向。

5. NOTICE 无锁范式

在我们如何使用通知，在竞争条件下为共享数据提供非阻塞并发读写访问的过程中，贯穿着一个共同的线程（请原谅这个双关语）。

1. 增量更新至关重要，因为这样才能安全地读取不变的增量前状态，并在读取和更新都并发进行的同时更新该状态。

2. 当线程希望执行转换性更新，而不是基于增量的更新时，通知有助于进行轻量级冲突解决。通知以与增量更新相同的方式添加到前端，从而允许并发读取和基于增量的更新。 3. 通知保护状态的一部分，该部分的转换（更新）无法通过增量更新完成。添加通知的状态，沿着放置该通知的访问路径接近该状态的任何其他线程都无法对其进行更改。 4. 必须在通往正在转换的状态部分的所有路径上添加通知。这样可以确保，受通知保护的状态部分只能由发布通知的线程更改。 5. 与通知竞争的线程必须确保：如果它赢得通知竞争，失败的线程仍能成功继续其操作，而无需更改受通知保护的状态，例如通过执行增量更新。 6. 正在转换的通知保护状态会被设为只读。它通过替换来完成转换，并在一次原子操作中将其附加到通知保护路径上涉及的状态。 7. 通知在完成其并发控制目的后，可以使用“void”标记在逻辑上禁用。通常也可以将其移除，不过这需要更多工作。 。

8 需要具备无闩锁基础设施，才能在安全时执行垃圾回收。Deuteronomy 使用了纪元方法 [3]。

与其他冲突解决方法（例如使用闩锁）一样，必须谨慎处理。例如，如果闩锁之间的顺序安排不当，就可能发生死锁。对于通知及其位置和顺序，也必须谨慎处理，以确保正确性。

6. 其他无锁注意事项

6.1 列表表示

存储管理可能代价高昂。此外，遍历链表时，可能需要先访问指针元素，再访问与该指针元素相关的数据。这些指针可能需要单独管理，包括其分配和垃圾回收，如图4(a)所示。

我们建议在此准备一个足够大的列表存储元素，以同时容纳一个列表指针及其关联的列表元素，如图 4(b) 所示。这并非新想法，但可能对性能产生显著影响。首先，它能将每个列表元素所需的存储分配数量减半。其次，与列表指针相邻的数据（图中的 Next）更有可能与该指针一同位于处理器缓存中，从而提升线程执行性能。



图 4：两种形式的列表格式。左侧形式所需的分配次数是右侧形式的两倍，而且访问“Data”时更有可能触发处理器缓存故障。

6.2 线程故障

CAS 时进行的“一次性”状态替换虽然执行开销更大，但有一个优点：如果 CAS 胜出线程因某种原因在安装新状态之前终止，总会有另一个线程接手完成这项工作。不过，我们尚未说明，发布 NOTICE 的线程在发布 NOTICE 后、安装更新后的状态前终止时会发生什么。

NOTICE 不仅可以标识胜出线程，还可以设置超时；超时后，NOTICE 将不再阻止其他线程执行先前分配给发布 NOTICE 的胜出线程的任务。为此，可以使用时钟时间，但纪元编号可能更有效。发布 NOTICE 的线程可以在 NOTICE 中注明其执行时所处的纪元。如果经过两三个纪元后工作仍未完成，另一个线程便可以接手。CAS 可以通过更改另一个 CAS 所见的 NOTICE，防止重复胜出者安装更改。

6.3 更新不在缓存中的页面

Cosmos DB（最初名为 Document DB） [9]，希望文档一录入文档数据库，就能立即按关键词建立索引，这称为“实时索引”。如果必须读入新文档中所有关键词对应的页，才能更新这些页以纳入新文档，那么实时索引就无法实现。
Document DB 选择使用 Bw-树为文档建立索引，不仅因为它在大多数情况下表现出色，也因为它支持这种实时索引。

Bw-tree 的每个页在基于向量的映射表中都有一个条目。映射表会指向内存中或辅助存储上的页。映射表利用日志结构化存储（见 6.4），从而允许重新定位页。由于增量更新，
Bw-tree 具备实时索引能力。如果待更新的页不在缓存中，其映射表条目将包含该页在闪存上的位置。这使得该页可以被

进行增量更新，因为增量更新无需访问该页的先前状态。先前状态会与新的增量更新关联起来，因此之后读取该页时，可以还原出该页的完整状态。

6.4. 利用日志结构化存储

资源争用并非线程停顿或线程切换的唯一来源。由于闪存延迟，I/O 操作可能导致需要切换线程。日志结构化是一种将页面写入批次化后写入辅助存储的技术 [10]。Deuteronomy 使用日志结构化来批次化写入。这样一来，不再是每页执行一次 I/O 操作，而是每批次执行一次 I/O。批次中的页面放入日志缓冲区时，不会发生线程切换。并且，使用取值并递增（FAI）指令即可在缓冲区中分配页面，因此缓冲区操作无需闩锁，也不会造成线程停顿。只有缓冲区已满时才需要进行更多操作，而 Bw-tree 也能以无闩锁的方式完成这一过程。

这需要通过虚拟化辅助存储来实现，但根据我们的经验，这总能大幅减少所需的 I/O 写入次数 [8]。遗憾的是，我们不知道有什么方法能在处理页面读取时同样减少 I/O 操作。

7. 其他工作

据我们所知，由日志结构化存储支持的 Bw-tree（作为 Deuteronomy 的数据组件（DC））及其后续版本 Bwᵉ-tree[10], , 是仅有的采用无闩锁技术的键值存储。竞争对手的存储引擎大多采用传统的闩锁机制。我们将自己的 DC 与其他键值存储进行了比较。在早期的基准测试中，我们使用了 Berkley DB [12]，其性能远不及 Bw-tree。RocksDB [13] 是一款应用非常广泛的存储引擎，部分原因在于它是开源的。它可以在不从辅助存储读取数据的情况下执行记录更新，这一特性也为 Deuteronomy 所具备。在我们的基准测试中，Deuteronomy 的表现更好[10]。但 Deuteronomy 并非开源，因此仅在少数地方使用[2,9,10]。

有一篇论文将 Bw-tree 与纯内存访问方法进行了比较，并且不出所料地发现，内存方法速度更快[11]。其他主存方法也展现出更高的性能，但这些论文并未评估成本效能。我们将 MassTree [6], 这一速度极快的主存系统纳入成本效能比较[5]，发现 MassTree 速度更快，但它使用的主存多得多。因此我们得出结论：只有在不计成本、追求极致性能时，才会选择 MassTree 而非 Deuteronomy 及其 Bw-tree；后者通过提高内存利用效率，并将数据从主存缓存移至辅助存储来降低成本。

通常，需要放在主存缓存中的热数据，远少于应留在辅助存储中、直到被访问并变为热数据的冷数据。这正是基于缓存的系统平衡成本与性能的方式，也是它们如此广泛使用的原因。所有大型商业数据库系统都是基于缓存的系统，原因正在于此。客户希望降低“销货成本”（商品和服务成本），只要性能足够好即可。

8. 结论

本文重新审视了如何利用通知大幅改进无闩锁技术：避免冗余工作，并使 B树型访问方法中常见的大规模状态变更变得更简单、更高效。免受线程中断的影响，让无闩锁方法在性能和可扩展性方面拥有显著优势，前提是其余代码路径都很短。通知通过先确定竞态条件中的胜出者，再执行开销较大的代码路径，从而缩短了其他路径。

减少线程切换是提升性能最可靠的方法之一。线程切换会“破坏”处理器缓存；如果切换涉及操作系统，还可能造成非常显著且往往难以察觉的开销。忽略竞争时，无闩锁技术的代码路径并不总是最短的。但它为竞争中的失败者提供了一条路径，使其在竞态条件中落败后仍能继续执行有用工作，从而避免了线程切换带来的巨大开销。

这种通用方法也可以扩展到事务冲突，我们已在 [3] 中采用了这一方法，并将在完整论文的附录中加以说明。

致谢

感谢 Justin Levandoski、Sudipta Sengupta 和 Ryan Stutsman，他们参与了 Deuteronomy 的实现，并贡献了其中的多项创新。

参考文献

[1] Justin Levandoski、David B. Lomet、Sudipta Sengupta：面向新硬件平台的B树：Bw-tree。*ICDE* *2013* 302 313 ：- Justin J. [2] Levandoski、David B. Lomet、Sudipta Sengupta、Adrian Birka、Cristian Diaconu：现代硬件上的索引：Hekaton 及其他。*SIGMOD Conference* *2014* 717 720 ：- [3] Justin J. Levandoski 、 David B. Lomet 、 Sudipta Sengupta 、 Ryan Stutsman 、 Rui Wang：Deuteronomy 中的高性能事务。 *CIDR* *2015* [4] David B. Lomet：有效B树的演进：页面组织与技术：个人回顾。SIGMOD Rec. 30(3)：64-69（2001）[5] David B. Lomet：现代数据存储系统中的成本/性能：数据缓存系统如何取得成功。 *DaMoN* *2018* *9 1 9 10* ：：-：Y. Mao、E. Kohler、R. T. Morris。快速 [6] 多核键值存储的缓存优化技巧。载于 EuroSys，2012 年，第 183-196 页。[7] C. Mohan、Don Haderle、Bruce G. Lindsay、Hamid Pirahesh、Peter M. Schwarz：ARIES：一种支持细粒度锁定和使用预写日志进行部分回滚的事务恢复方法。ACM Trans. Database Syst. 17(1)：94-162（1992）[8] Mendel Rosenblum、John K. Ousterhout：日志结构文件系统的设计与实现。SOSP 1991：1-15.....[9] Dharma Shukla、Shireesh Thota、Karthik Raman、Madhan Gajendran、Ankur Shah、Sergii Ziuzin、Krishnan Sundaram、Miguel Gonzalez Guajardo、Anna Wawrzyniak、Samer Boshra、Renato Ferreira、Mohamed Nassar、Michael Koltachev、Ji Huang、Sudipta Sengupta、Justin J. Levandoski、David B. Lomet：使用 Azure DocumentDB 进行模式无关索引。 *Proc. VLDB Endow. 8(12)*：1668-1679（2015）[10] Rui Wang、Xinjun Yang、Feifei Li、David B. Lomet、Xin Liu、Panfeng Zhou、Yongxiang Chen、David Zhang、Jingren Zhou、Jiesheng Wu：Bwe-tree：快速存储上的 Bw-tree 演进。*ICDE* *2024* 5266 5279 ：- Ziqi Wang、Andrew [11] Pavlo、Hyeontaek Lim、Viktor Leis、Huanchen Zhang、Michael Kaminsky、David G. Andersen：构建 Bw-tree 不仅仅需要流行术语。 *SIGMOD Conference 2018*：473-488。[12] 维基百科：https://en.wikipedia.org>wiki>Berkeley_DB [13] 维基百科：https://en.wikipedia.org › wiki › RocksDB