Noname 手稿编号（将由编辑插入）

更新密集型基于 LSM 的 R 树索引

Jaewoo Shin · Jianguo Wang · Walid G. Aref

arXiv:2305.01087v1 [cs.DS] 1 May 2023

收稿日期：日期 / 录用日期：日期

摘要 许多应用需要对空间对象执行更新密集型工作负载，例如跟踪移动对象的社交网络服务和共享出行服务。通过在内存中缓冲插入和删除操作，日志结构合并树（LSM）凭借处理写密集型工作负载的能力，已广泛应用于各种系统。虽然对 LSM 的研究重点一直在键值存储及其优化上，但现代异构数据需要使用二级索引，因此有必要研究如何高效支持基于 LSM 的二级索引（例如基于位置的索引）。本文研究如何将一种基于主内存的备忘录结构扩展为 LSM 二级索引结构，以高效处理更新密集型工作负载。我们在基于 R 树的二级索引环境中开展这项研究。具体而言，我们引入了 LSM RUM 树，展示了如何在基于 LSM 的 R 树中使用更新备忘录，以提升 R 树的插入、删除、更新和搜索操作的性能。LSM RUM 树引入了新的策略来控制更新备忘录的大小，确保其始终适合存放在内存中，从而实现高性能。更新备忘录是一种轻量级的内存结构，适合处理更新密集型工作负载，且不会引入显著的开销。使用真实空间数据进行的实验结果表明，LSM RUM 树与现有的 LSM R 树实现相比，更新操作的速度最高可提升 9.6 倍，查询处理的速度最高可提升 2400 倍。

关键词 基于 LSM 的索引 · 二级索引 ·查询处理 · R 树 · 大数据 · 空间数据库

J. Shin 普渡大学，西拉斐特，印第安纳州 电子邮箱：shin152@purdue.edu J. Wang 普渡大学，西拉斐特，印第安纳州 电子邮箱：csjgwang@purdue.edu W. G. Aref 普渡大学，西拉斐特，印第安纳州 电子邮箱：aref@purdue.edu

1 引言

近年来，移动设备、社交媒体和共享出行服务持续产生海量位置数据。随着设备或对象在空间中移动，它们会更新自身的位置，并期望获得响应迅速的服务（例如获取紧急天气信息和位置定向广告）。从系统角度来看，如何高效处理更新密集型位置工作负载并以低延迟响应查询，是一项挑战。

日志结构合并树（简称 LSM）是应对写入密集型工作负载的一种广泛采用的方法 [19]。LSM 的核心思想是在内存中缓冲数据摄取，然后定期将缓冲区刷新到磁盘。对于写入负载较重的工作负载，这可以将随机 I/O 转换为顺序 I/O。LSM R 树已被提出，用于处理写入密集型空间工作负载 [6]。位置或坐标并非主键，而是二级键。原因是，位置数据无法像对象标识符那样唯一标识一个对象。此外，同一位置在不同时间可能被多个对象共享。

在写入密集型（更新密集型）工作负载中，以较低的开销确定对象的最新状态（在本例中即对象的当前位置）是一项具有挑战性的任务。例如，考虑一个维护当前位置的移动对象空间数据库应用。

空间中移动的对象的位置。由于对象持续移动，这一应用具有更新密集型的特点。用于存储位置数据的类LSM 结构将是理想的选择。LSM R树 [6,16]已被引入，用于在空间中接收对象的移动，同时能够回答面向位置的查询，例如范围查询。在 LSM R树中，在空间中移动的对象会持续将其位置更新到内存 R 树中。每个对象只有一个“当前”位置，而该对象之前的所有位置都会变为过时位置。当分配给内存 R 树的内存耗尽时，该树会被转存到磁盘，并启动一个新的内存 R 树。在查询 R 树时，例如执行范围搜索查询，需要探测所有基于内存和磁盘的 R 树，因此结果可能包含对象的过时位置。如果处理不当，这可能导致范围查询返回错误的答案。因此，LSM R树采取了额外的措施，以确保结果的正确性 [6,16]。

Alsubaiee等人 [6]通过采用*Eager* 策略解决了这一问题：将一种额外的数据结构，即已删除键 B^+-树，与内存 R 树关联起来，用于存储已删除对象的键。这表示对象的旧版本已被删除，并插入了一个新版本。已删除键 B^+-树也会与相应的 R 树一起刷写到磁盘。由于需要维护和访问已删除键 B^+-树，该方案会产生额外开销，从而影响更新性能和查询性能。

解决这一问题的另一种方案是采用*Validation* 策略 [16]，移除与每棵 R 树关联的已删除键 B^+-树。取而代之的是，使用带时间戳的主键索引（即存储主键的 B^+-树），在基于 LSM 的二级索引执行合并操作时，延迟清理过时对象。与时间戳关联的独立主键索引有助于避免额外的维护成本，并提升更新性能。然而，为了验证对象的最新状态，该方法会在搜索操作中产生额外开销。尽管提升了更新性能，与 Eager 策略相比，该方法会降低搜索性能。

RUM 树 [23, 24] 已被提出：它通过在更新备忘录（UM）中维护对象最近一次更新的时间戳，利用内存结构处理频繁更新的位置数据。UM 已应用于 R 树 [23,24,26], 、网格文件 [25], 、有限轨迹索引 [17], , 以及移动对象当前和近期位置的索引 [9]。由于 UM 会在内存中保留每次更新，因此其大小至关重要。在 RUM 树 [23, 24] 中，通过探查磁盘上的 R 树节点并惰性地执行垃圾清理，可以控制（设定上限）UM 的大小。然而，原始 RUM 树及其变体中的垃圾清理策略无法直接应用于基于 LSM 的 R 树索引，因为在 LSM 中，多个 R 树作为磁盘组件存在，其中存储着一些已移动对象的过时位置。在这种情况下，若不针对 LSM 环境进行调整就执行垃圾清理，成本会很高。要从 R 树中清除这些过时对象并非易事，因此在基于 LSM 的索引中部署时，更难为 UM 的大小设定上限。此外，原始更新备忘录对插入命令很敏感，即在配套索引执行插入操作时就会更新。这对于插入密集型工作负载尤其不实用。即使在内存层维护多个删除键 B^+-树 [6] 或主键索引 [16]，也不足以应对更新密集型工作负载。原因在于，维护多个删除键 B^+-树需要在每棵删除键 B^+-树中进行多路径搜索，而且与 LSM R 树协调所需的维护成本依然存在。同样，在内存中维护主键索引也无法扩展。原因是它可能超出内存预算，因为主键索引的大小与主键总数成正比，而这并不能保证能放入内存。

我们提出了一种新方案，可同时提升更新和搜索性能。我们对更新备忘录在 LSM 二级索引（主要是 R 树）中的应用进行了调整和研究。我们将启用新型更新备忘录的索引称为 LSM RUM 树。

本文的主要贡献如下：

– 我们介绍了 LSM RUM 树，这是一种基于 LSM 的 R 树，利用驻留在内存中的更新备忘录（UM），在不牺牲搜索性能的前提下支持更新密集型空间工作负载。LSM RUM 树在磁盘上维护多个 R 树，但其仅驻留在内存中的 UM 组件大小有上限（不会随系统中的对象数量成比例增长）。UM 保证不会超出分配给它的内存。–我们在地理位置数据的背景下介绍 LSM RUM 树。不过，基于 LSM 的更新备忘录（UM）也可应用于其他更新密集型二级索引，例如 B^+ 树或倒排索引（第3节）。– 我们介绍了四种清理策略，用于限制内存中 UM 的大小以及磁盘上 R 树的数量。这

不仅减少了内存空间的消耗，还提升了整体性能（第4节）。– 我们使用真实数据集进行了大量实验，以复现更新密集型工作负载。实验结果表明，LSM RUM 树提升了基于 LSM 的 R 树在删除、更新和搜索操作上的性能。与当前最先进的 LSM R 树实现相比，LSM RUM 树在更新操作上最高可实现 9.6 倍加速，在搜索性能上最高可实现 2400 倍加速。

本文是介绍 LSM RUM 树 [22] 的短论文的扩展版本。本文在以下方面扩展了短论文 [22] 。

– 我们扩展了对 LSM 感知的更新备忘录清理策略的分析，以展示各项更新备忘录清理策略缩减更新备忘录大小的效果，以及它们带来的性能提升（第 4 节）。

– 我们为更新备忘录引入了并发控制支持。这对于支持针对 LSM RUM 树的并发活动至关重要，例如在多线程/多核环境中。我们引入原子操作比较并在较小时交换(CILS)，以支持多线程环境中针对更新备忘录的并发活动（第5节）。实验表明，与现有实现和基线实现相比，并发更新操作的速度提升了 4.5 倍。– 我们对 LSM RUM 树进行了全面分析，旨在展示相较于现有实现，性能提升体现在哪些方面。除了原短文中的实验外，我们还新增了 LSM RUM 树实验，并讨论了实验结果（第6节）。

本文其余部分安排如下。第2节介绍背景知识并讨论相关工作。第3节介绍 LSM RUM 树及其所有支持的索引操作，并展示如何调整更新备忘录（UM），使其能够在 LSM RUM 树内部使用。第 4 节介绍清理策略如何限制 UM 的大小，以及如何提高 LSM RUM 树的性能。第5节讨论我们如何支持 UM 中的并发操作。第6节通过大量实验，将 LSM RUM 树与以往的方法进行比较。第7节给出总结性结论。

2 背景与相关工作

本节概述 LSM 树 [19] ，并介绍两种用于处理 LSM R 树中空间工作负载的已有方法。由于本文的目标是通过使用更新备忘录来提升 LSMR 树的性能，因此，了解 LSM R 树如何执行各项操作，以及更新备忘录如何加速 R 树的更新过程，十分重要。

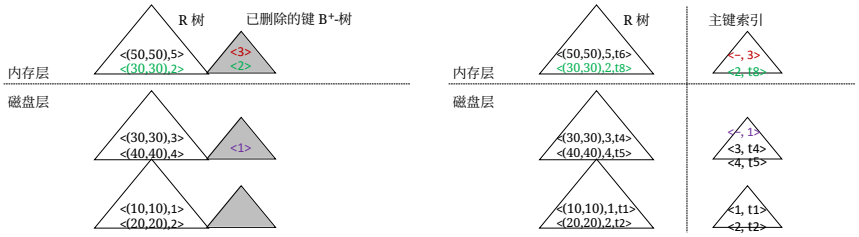
2.1 日志结构合并树

LSM树 [19] 通过在内存层中缓冲数据摄取来优化写入密集型工作负载。一旦内存层中的数据大小超过某个阈值，LSM树就会使用顺序I/O将内存数据刷新到磁盘层。随着刷写组件在磁盘层中累积，它们会根据合并策略进行合并，从而使LSM树维持有限数量的磁盘组件。由于LSM树能够高效地缓冲处理写入密集型工作负载，因此它已广泛应用于各种系统中，例如 AsterixDB [6,16], Cassandra [2], LevelDB [3], 和 RocksDB [5]。

为便于说明，本文沿用先前研究中的以下术语。主索引 是一种 B$^+$-树，其中键为主键（例如， O_{id}），值为描述对象的数据。主键索引 和已删除键 B^+-树 也都是 B$^+$-树，但其中仅存储主键，而非整个对象。LSM 二级索引（例如 LSM R 树）包含一个二级键（例如位置），其值为对象的主键。请注意，对于相同的二级键，可能存在多个对象（例如，不同对象可能位于同一位置）。

2.2 LSM R树

传统的基于磁盘的空间索引，例如 R 树 [12], ，采用“原地更新”策略。这会导致大量随机 I/O。鉴于空间数据领域中移动对象的高更新率，以及对这些移动对象进行索引的需求，采用 LSM 来处理这些更新密集型空间工作负载并不令人意外。LSM R 树 [6] 是一种用于高效处理位置数据的空间索引，并受益于 LSM 机制。通过应用一种通用的二级索引*LSM* 化 [6], 框架，LSM R 树成为针对写入密集型空间数据工作负载优化的二级索引。为处理 LSM R 树索引中的频繁更新，已有两种策略被提出，分别称为 Eager [6] 和 Validation [16],。本文着重于优化 LSM R 树的性能。传统上，



ts	操作。	对象
t1	插入	⟨(10，10) 1⟩
t2	插入	⟨(20，20) 2⟩
t3	删除	⟨1⟩
t4	插入	⟨(30，30) 3⟩
t5	插入	⟨(40, 40) 4⟩
t6	插入	⟨(50，50) 5⟩
t7	删除	⟨3⟩
t8	更新	⟨(30，30) 2⟩

(a) 急切策略下的 LSM R树 (b) 验证策略下的 LSMR树　　(c) 对象及其操作

图 1：运行示例

更新二级键值（例如，更新对象的位置）需要同时维护 LSM 主和二级索引，以确保两个索引之间的一致性。在主索引侧，对象标识符（键）需要保存对象的新位置；而在二级索引侧，对象的新位置（键）应指向该对象的标识符。维护主索引很简单，只需将更新缓冲到内存层的组件中。然而，维护 LSM 二级索引则较为复杂，因为同一个二级键可能对应多个对象，且同一个对象可能有多个版本（即最新版本或过时版本）。此外，先前的研究 [6, 16] 要求在维护二级索引期间对主索引执行点查询。因此，为了简化二级索引的说明，我们只重点介绍 LSM R 树的维护。

2.2.1 *LSM* 二级索引的急切策略　　*es*

Alsubaiee等人 [6] 为每个 R 树在 LSM 中引入一个额外的已删除键 B$^+$-树，以确保 LSM R 树的一致性。已删除键 B$^+$ 树存储已删除/已更新对象的主键（对象标识符），以便根据查询验证对象的状态。内存 R 树及其对应的已删除键 B$^+$-树紧密耦合，并作为一个组件一起刷写到磁盘。尽管已删除键 B$^+$-树会在内存层缓冲删除操作，但搜索操作期间额外的维护成本和磁盘 I/O 会降低整体搜索性能。

假设我们有一个对象，例如 $o = \langle Loc, O_{id}, ... \rangle$，其中 Loc 表示 o的位置，而O_{id} 是 o的对象标识符。由于 R 树对位置建立索引，Loc 是 R 树索引的二级键，而 O_{id} 是一个外键，指向主索引中该对象的主键；主索引可能还包含用于描述 o的其他属性。要将 o 插入 LSM R树，我们将 o 插入内存 R 树，无需访问已删除键 B$^+$树。

要从 LSM R 树中删除条目 $o = \langle Loc, O_{id} \rangle$，我们执行以下步骤：(1) 如果内存 R 树索引中存在 $o = \langle Loc, O_{id} \rangle$，则将其移除；(2) 通过将 O_{id} 添加到内存中的已删除键 B$^+$ 树，使磁盘组件中的过时 o 失效。对于 *Eager* 策略，步骤 1 对于确保 LSM R 树的一致性是必要的。例如，对于某个对象的一系列更新，如果不在内存组件中执行步骤 1，即插入：$\langle Loc_1, O_i \rangle$、删除：$O_i$，再插入：$\langle Loc_2, O_i \rangle$，就无法确定条目 $\langle Loc_1, O_i \rangle$ 和 $\langle Loc_2, O_i \rangle$ 中哪一个是最近的位置。如 [6]，所述，已删除键 B$^+$ 树中的条目有助于验证磁盘组件中的对象。

要更新 LSM R 树中的空间对象 o，例如，由于 O_{id} 将其位置从 Loc_{old} 更改为 Loc_{new}，我们执行以下操作：删除旧对象 $o = \langle Loc_{old}, O_{id} \rangle$（按照上述删除过程），然后将新对象 $o = \langle Loc_{new}, O_{id} \rangle$ 插入 LSM R 树中（按照上述插入过程）。

要搜索 LSM R 树（例如，查找 $Loc = (2, 2)$ 周围的所有对象），需要搜索所有 R 树，无论它们位于内存中还是磁盘上，以查找候选查询结果，因为所有 R 树中都可能包含符合条件的对象。请注意，二级索引，即 R 树，对于给定的二级键（即 Loc）可能包含多条记录，因此使用该二级键进行搜索的结果可能位于任意组件中。然后，需要针对所有 现有的已删除键 B$^+$-树搜索每个候选项的O_{id}；仅当 O_{id} 不存在于任何比候选项所在组件更新的组件的已删除键 B$^+$-树中时，才会将其作为输出返回。请注意，由于内存中的已删除键 B$^+$-树会与其对应的 R 树一起刷新到磁盘，因此会存在多个已删除键 B$^+$-树，并且我们会在内存中与对应的新内存 R 树一起启动一个新的已删除键 B$^+$-树。

运行示例：请参阅表1c和图1a以了解示例。假设插入和删除事件按照表中所示的顺序到达，其中每个对象用 $\langle Loc, O_{id}\rangle$ 表示。图1a展示了 *Eager*策略如何处理 LSM R 树中的插入、删除和更新。每次插入操作都会将一个对象添加到 R 树中。为便于说明，假设 R 树的容量为 2，即当 R 树中有两个对象时，就会将其刷新到磁盘。在时间 t3 时，对象 1 和 2 已经插入内存 R 树中，并且由于其容量已达到上限，内存 R 树会与其对应的空已删除键 B$^+$-树一起刷新到磁盘（两者均显示在图的底行中）。在时间 t3 时，DELETE $\langle 1\rangle$ 会使现有对象 $\langle$**(10，10)** $1\rangle$ 失效。因此， O_{id} $\langle 1\rangle$ 会被插入已删除键 B$^+$-树中（该已删除键 B$^+$-树显示在图的第 2 行中）。因此，在任何未来时间搜索 R 树并返回对象 $\langle$**(10，10)** $1\rangle$ 时，该输出会因已删除键 B$^+$-树中存在的键 $\langle 1\rangle$ 而失效。请注意，*Eager* 策略需要额外的开销来维护多个已删除键 B$^+$-树。

2.2.2 LSM 二级索引的验证策略

Validation 策略 [16]解决了 Eager 策略的更新开销。它避免使用已删除键的 B$^+$ 树，而是使用 主键索引 验证查询结果。Validation 策略会在主索引和次级索引的每个对象中添加时间戳。图1b 展示了 Validation 策略如何处理上述同一运行示例中的数据操作。对于插入操作，它会将一个对象作为 $\langle Loc, O_{id}, ts\rangle$ 插入 R 树，其中 ts 是使用本地挂钟时间记录的插入时间戳。此外，对于插入操作，主键索引 会将一个对象存储为$\langle O_{id}, ts\rangle$。对于 Time t3 时的删除操作，Validation 策略只会在主键索引中插入一个控制条目（例如， $\langle -, 1\rangle$，表示对象 1 已删除）。因此，相较于 *Eager* 策略，它简化了删除或更新过程。不过，由于需要通过主索引进行直接验证或通过主键索引进行时间戳验证， *Validation* 策略的查询性能仍会受到所需验证步骤的影响。

总而言之，*Eager* 和 *Validation* 策略的更新和搜索性能都不理想。对于更新，*Eager* 策略会遍历内存 R 树，删除旧对象条目，并向已删除键的 B$^+$-树中插入一个控制条目，然后插入新对象。因此，Eager 策略的更新性能会下降。*Validation*策略在更新时不会修改 R 树，而是将带有时间戳的新对象插入主键索引和 R 树。与 *Eager* 策略相比， *Validation*策略的更新速度更快。然而， *Validation* 策略的搜索性能会下降，因为二级索引会产生误报结果，还需要将结果与主键索引进行比对以验证。

2.3 RUM 树：带有更新备忘录的 R 树

RUM 树 [23, 24] 是处理更新密集型空间工作负载的另一种方法。它为 R 树增添了一种驻留内存的更新备忘录（UM）结构。RUM 树是一种基于磁盘的 R 树索引，并在内存中维护一个 UM。它维护一个全局时间戳计数器，并为每个对象标记，以表示对象之间的时间关系。每个 UM 条目的形式为： $\langle O_{id}, ts, cnt\rangle$，其中 ts 是 RUM 树中对 O_{id} 最近一次更新的时间戳，而 cnt 是索引中对象 O_{id} 的过时版本数。通过在 UM 中处理插入和更新操作，RUM 树显著降低了更新成本，同时不会对搜索造成太大影响。UM 采用了多种清理策略，用于从 R 树和 UM 中移除过时条目，以限制 UM 的大小。因此，与传统 R 树相比，RUM 树在更新方面性能更佳 [9, 23, 24]。不过，无法在 LSM 环境中直接照搬使用 RUM 树，因为其 UM 是针对传统 R 树优化的；若不修改相关机制就直接使用，其 UM 维护过程会导致 UM 大小过大。本文进一步拓展了更新备忘录的思路，并提出新的策略，使其适用于基于 LSM 的 R 树二级索引。在以下各节中，我们将说明如何为此用途调整 UM，并介绍基于 LSM 的 R 树的插入、删除、更新和搜索算法，展示这些算法如何利用 UM 并控制其大小，从而确保 UM 始终能够放入内存。

3 LSM RUM 树

我们介绍了新型 LSM RUM 树，即一种增添了更新备忘录的基于 LSM 的 R 树。LSM RUM 树旨在高效处理更新密集型空间工作负载，并提升搜索性能。根据第2节第2.2小节的内容，现有的 LSM 二级索引策略会降低更新和搜索性能。为解决这一问题，我们在基于 LSM 的 R 树中引入更新备忘录，

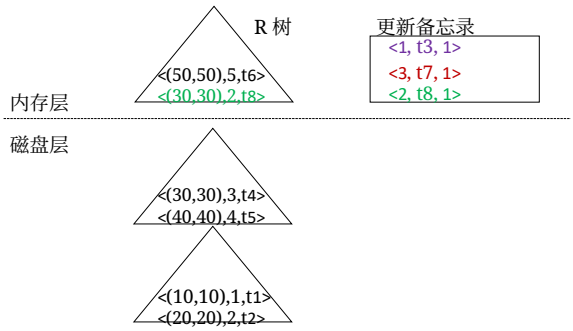


图 2：LSM RUM 树概览

以简化内存层中的删除和更新处理，并降低基于磁盘的搜索成本。

3.1 更新备忘录结构 (更新备忘录)

示意见图 2。在 LSM RUM 树中，更新备忘录（UM）基于驻留在内存中的哈希映射。UM 的键是对象标识符，值是一对最近的时间戳和过时对象计数器。UM 有多种作用。它会跟踪写入 LSM RUM 树的删除和更新。由于它仅驻留在内存层中，更新 UM 条目只需常数时间。此外，UM 还会验证搜索操作产生的候选对象。如第2.2节所述，以往的策略需要执行额外工作才能正确回答查询。*Validation* 策略需要通过将输出候选对象与主键索引进行比较来验证搜索的输出候选对象，其方式与*Eager* 策略类似。由于 UM 轻量且完全驻留在内存中，我们预计其搜索性能将优于以往方法。

如图 2 所示，LSM RUM 树中的每个对象都表示为一个三元组 $\langle Loc, O_{id}, ts\rangle$，其中 Loc 是对象的位置，O_{id} 是对象标识符，而ts 是全局时间戳计数器——一个整数值，每执行一次插入、删除或更新操作就递增 1。 ts的值越大，对象在索引中的时效性就越高。UM 中的条目 e 表示为一个三元组 $\langle O_{id}, ts, cnt\rangle$，其中 cnt 是 LSM RUM 树中具有相同 O_{id} 的过时对象数量计数器。默认情况下，删除或更新一个在 UM 中没有现有条目的对象时，会先将条目 $e = \langle O_{id}, ts, 1\rangle$ 插入 UM。否则，如果 UM 中先前某次删除或更新所对应的条目（例如 e）中已存在 O_{id}，则将 $e.cnt$ 递增 1。当在 LSM RUM 树中找到一个过时对象并将其移除时，我们将 UM 中该对象的 $e.cnt$ 递减 1。当 $e.cnt = 0$时，条目 e会从 UM 中移除；也就是说，当前 LSM RUM 树中不存在 O_{id} 的过时条目。

3.2 更新备忘录中的惰性维护

更新备忘录在内存中缓冲删除和更新操作。下面，我们将讨论 LSMRUM 树中的删除、更新和搜索操作如何利用更新备忘录。

3.2.1 插入

算法 1： 插入操作

输入： Loc：对象的位置（二级键）
O_{id}：对象 ID（主键）

```
1 ts ← 时间戳 计数器++;
2 O_new = ⟨Loc, O_id, ts⟩;
3 insertLSMRtree(O_new);
```

参见算法 1。在插入过程中，首先将全局时间戳计数器加 1（第 1 行）。请注意，时间戳计数器是一个整数，每执行一次插入、删除或更新操作，该值就会递增。读取全局时间戳计数器后，假设其值为 ts，便将一个新对象 $o_{new} = \langle Loc, O_{id}, ts\rangle$ 添加到内存 R 树中（第 2–3 行）。UM 仅跟踪对象的删除和更新，不对插入执行任何特殊操作。LSM RUM 树的目标是支持更新密集型和插入密集型的空间工作负载。如果 UM 像原始 RUM 树 [23,24]，那样处理插入操作，那么对于插入密集型工作负载，UM 的大小将随着对象数量的增加而线性增长。这将导致显著的内存空间开销。在 LSM RUM 树中，通过避免在插入对象时维护 UM，可以防止 UM 大小出现不必要的增长。在第3节.3 中，我们将详细讨论如何验证在 UM 中没有对应条目的对象。

3.2.2 删除

算法 2： 删除操作

输入： O_{id}：对象 ID（主键）

```
1 ts ← 时间戳 计数器++;
2 如果 entry e for O_id exist in UM 则
3     e.ts ← ts;
4     e.cnt++;
5 否则
6     将 e_new = ⟨O_id, ts, 1⟩ 放入 UM;
```

在 LSM RUM 树中，我们尽量减少验证结果所需的处理工作。要删除一个对象，我们只需添加或修改该对象对应的 UM 条目。如算法 2 所示，如果给定的 O_{id} 有一个 UM 条目（例如 e），我们就将 Field $e.ts$ 设为全局时间戳（第 3 行），并将 $e.cnt$ 加 1（第 4 行）。如果在 UM 中没有这样的 e，我们就将 $\langle O_{id}, ts, 1\rangle$ 插入 UM。请注意， $e.cnt$ 的值对应于 R 树中给定 O_{id} 的过时副本数量。自然，每次删除操作都会在 R 树中新增一个过时对象。通过跟踪 ts 以判断新旧，以及跟踪 cnt 以统计 O_{id} 的过时对象数量，我们不仅能清楚地判断索引中给定对象副本是否为最新，还能准确掌握 LSM RUM 树中该对象的过时副本数量。在图 2 的示例中，“t3：删除⟨1⟩”和“t7：删除 ⟨3⟩”分别向 UM 中添加了 $\langle 1, t3, 1\rangle$ 和 $\langle 3, t7, 1\rangle$。

3.2.3 更新

为了处理一次更新，我们检查 UM 中是否包含一个具有相同 O_{id} 的条目，例如 e。如果 e 在 UM 中存在，我们将 $e.ts$ 更新为当前时间戳ts，并将 $e.cnt$ 增加 1。如果 e 不存在，我们则向 UM 中添加一个新条目 $e_{new} = \langle O_{id}, ts, 1\rangle$。然后，我们将新的对象条目 $\langle Loc, O_{id}, ts\rangle$ 添加到内存 R 树中。请注意，更新同时包含删除和插入。因此，我们遵循算法 2 的第 1–6 行以及算法 1 的第 2–3 行。在图 2 的示例中，“t8 : 更新 ⟨(30,30) 2⟩” 将条目 $\langle 2, t8, 1\rangle$ 添加到 UM 中，并将新对象 ⟨(30,30) 2, $t8$⟩ 插入内存 R 树中。请注意，如果在 t9 时对同一对象执行另一次更新，我们会将 UM 条目从 $\langle 2, t8, 1\rangle$ 更新为 $\langle 2, t9, 2\rangle$，并将带有时间戳 $t9$ 的新对象插入 R 树中。

3.3 搜索

算法 3： 使用更新备忘录进行验证

输入： $candidates$：a 的候选项列表
搜索
输出： $results$：搜索的结果列表

```
1 对于 O_cand ← candidates 执行
2     O_id ← O_cand.O_id;
3     如果 entry e for O_id exist in UM 则
4         如果 O_cand.ts == e.ts 则
5             results.插入(O_cand);
6     else
7         results.插入(O_cand);
8 返回 results
```

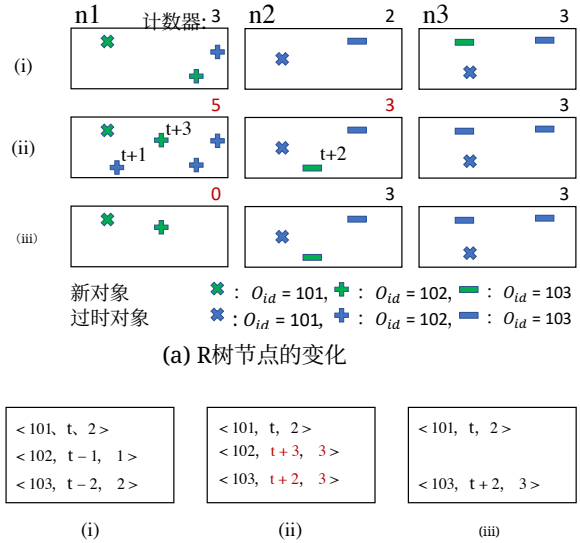
我们解释 LSM RUM 树如何执行搜索操作。$Eager$ 和 $Validation$ 策略都需要额外的磁盘 I/O 来验证搜索结果，这是由于需要访问额外的数据结构，即已删除键 B$^+$-树或主键索引（B$^+$-树）。UM 驻留在内存中（得益于第 4 节中的清理策略）。因此，除访问基于磁盘的 LSM R 树外，LSM RUM 树不需要磁盘 I/O。通过从 LSM RUM 树中移除额外的树结构，我们节省了额外的树遍历时间以及维护成本，并预计在搜索期间获得更好的性能。

算法 3 说明了如何利用 UM 验证从 LSM RUM 树返回的搜索结果。首先，我们检查来自 R 树搜索的候选项 O_{cand}是否为最新。如果 UM 不包含具有相同 O_{id} 的条目，则 O_{cand} 是最新的，并且属于搜索结果。如果存在一个条目，例如 e，其具有相同的 O_{id}，则将候选项的 ts 字段与 UM 中的 $e.ts$ 进行比较。如果 $O_{cand}.ts < e.ts$，则该候选项已过时，会被丢弃，否则，也会被丢弃。请注意，不存在 $O_{cand}.ts > e.ts$ 的情况，因为我们始终维护一个 UM 条目，以反映对象的最新时间戳。此外，如果存在一个最新对象（例如，表 1c 中的“t5：插入 ⟨(40, 40) 4⟩”），并且同一对象没有其他更新，则不存在具有相同 $O_{id} = 4$ 的 UM 条目。例如，在图 2 中，如果从 (30,30) 到 (40,40) 执行范围搜索，则 LSM RUM 树中的候选项为[⟨(30,30) 3, $t4$⟩、 ⟨(40,40) 4, $t5$⟩、 ⟨(30,30) 2, $t8$⟩]。候选项 ⟨(30, 30) 3, $t4$⟩ 的时间戳小于其在 UM 中的条目（$\langle 3, t7, 1\rangle$），这是由于 t7 时的删除⟨3⟩ 造成的。因此，该候选项会被丢弃。对于⟨(40,40) 4, $t5$⟩，UM 不包含具有$o_{id} = 4$的条目，即该对象是最新的，并且该候选项属于搜索结果。对于对象 ⟨(30,30) 2, $t8$⟩，其条目 $\langle 2, t8, 1\rangle$ 表明该对象同样属于结果。因此，最终的搜索结果为 [⟨(40,40) 4, $t5$⟩、⟨(30, 30) 2, $t8$⟩]。

由于 UM 基于哈希映射，大量删除/更新操作会增大 UM 的体积。此外，随着哈希映射增大，每个哈希映射桶中的条目数量也会增加，导致查找性能下降，进而降低搜索性能。因此，在下一节中，我们将介绍 LSM 感知的 UM 清理策略，以限制 UM 的大小并提升搜索性能。

4 LSM 感知的 UM 清理策略

为确保 UM 能够放入内存，我们通过清理策略来控制其大小。我们介绍了多种清理策略，能够以较低开销高效地缩小 UM 的体积。如第3.1节所述，当 UM 中某个条目的字段 cnt 变为 0 时，该条目 e 将被移除。因此，我们关注的是如何减少 LSM RUM 树上每次操作所需的 $e.cnt$ 。我们为 LSM RUM 树介绍了四种清理策略：(1) 缓冲清理，(2) 真空清理，(3) 刷新时清理，以及 (4) 合并时清理。前两种策略通过内存 R 树清理 UM；相比之下，后两种策略则通过磁盘端的 R 树清理 UM。虽然 RUM 树 [23] 也采用清理策略来清除底层 R 树中的过时对象，但由于基于 LSM 的 R 树更为复杂，无法直接将相同策略应用于基于 LSM 的 R 树。



(a) R树节点的变化

(b) 更新备忘录的变化

图 3：缓冲清理示例

4.1 缓冲清理

当应用使用 LSM RUM 树作为二级索引来处理持续的更新密集型工作负载（例如，持续跟踪移动对象）时，内存 R 树的某个节点很可能包含多个过时对象。为清理 UM 和这些内存 R 树节点，我们提出了缓冲清理策略，该策略根据节点内累积的更新来清理内存 R 树节点。这尤其适用于热点清理。我们为内存 R 树的每个节点维护一个更新计数器 。当 LSM RUM 树缓冲一项更新操作时，该节点上的更新计数器便递增 1。计数器达到某个阈值后，我们会从节点中移除过时对象，并针对每个被移除的过时对象，将 UM 条目的 cnt 值减 1，从而清理该 UM 条目。创建新的 R 树节点时（例如，由于节点分裂），或者缓冲清理清理某个特定节点时，我们会将该节点的更新计数器设为 0。

例如，图 3a 和 3b 展示了缓冲清理在 LSM RUM 树中的工作方式。三个不同对象（其 O_{id}s 分别为 101、102 和 103）分布在三个 R 树节点（$n1$、 $n2$ 和 $n3$）上。我们将新对象标为绿色，将过时对象标为蓝色。假设更新计数器的阈值为 5。图 3a(i) 和 3b(i) 给出了某个时刻 t 的 R 树节点和更新备忘录的状态。请注意，节点 $n1$、 $n2$ 和 $n3$ 的计数器分别为 3、2 和 3。在 $t+1$ 处又进行了三次更新，

$t+2$和 $t+3$ 发生，从而得到图 3a(ii) 和 3b(ii)，如第 3.2.3 节所示。由于 $n1$的计数器达到阈值，我们也会清理 $n1$以及更新备忘录（UM）。更新计数器的阈值是一个决定调用缓冲清理策略频率的变量。阈值设置得越低，调用缓冲清理策略的频率就越高。请注意，频繁的缓冲清理会产生额外开销，用于清理节点和更新备忘录（UM）。因此，我们需要根据工作负载的特征调整阈值。例如，如果传入的位置更新是连续且渐进的（例如，由于对象进行小幅位移），较低的阈值能够高效运行。如果位置更新是周期性的且在地理位置上随机分布（例如社交媒体），较高的阈值可以避免额外的清理开销。

算法 4： 一个节点和更新备忘录清理

输入： $objects$: 节点中的对象列表

```
1 对于 O ← objects，执行
2     O_id ← O.O_id;
3     如果 entry e for O_id exist in UM，则
4         如果 O.ts < e.ts，则
5             从节点中移除 O ;
6             e.cnt − −;
7             如果 e.cnt == 0，则
8                 从更新备忘录中移除 e ;
```

算法 4 展示了我们如何清理 R 树节点及其在更新备忘录中的对应条目。在遍历节点中的对象时，我们丢弃过时的

通过比较节点中对象的时间戳与 UM 中该对象条目所记录的时间戳，将过时对象从节点中移除（第 2–5 行）。如果移除了一个过时对象，则将 UM 中与该对象对应的 $e.cnt$ 条目减 1，以跟踪具有相同 O_{id} 的过时对象数量（第 6 行）。如果 $e.cnt = 0$，则 LSM RUM 树中不存在具有相同 O_{id} 的过时对象。因此，我们从 UM 中移除 e。请注意，存在两种满足 $e.cnt = 0$的情况：（1）LSM RUM 树中仅剩一个具有 O_{id} 的对象副本，且该对象是最新的；或者（2）不存在这样的对象，因为所有过时对象均已被移除，并且具有相同 O_{id} 的对象的最后一次操作是删除操作。缓冲清理策略以一致的方式处理这两种情况。在清理步骤结束时，缓冲清理策略将更新计数器重置为 0。

4.2 真空清理

真空清理是缓冲清理的补充，因为它主要针对 LSM RUM 树中的热点节点。仍有一些情况是缓冲清理无法很好处理的：（1）节点位于冷点，导致缓冲清理的计数器未达到阈值；或（2）节点中的对象因其他节点的更新而过时。未清理这些冷点节点会导致更新备忘录（UM）的大小不断增长，且无法加以控制。为了弥补缓冲清理未处理的冷点节点，我们引入真空清理，以公平地清理内存 R 树节点，从而限制更新备忘录的大小。

在真空清理中，我们维护一个全局计数器和一个真空清理器。全局计数器记录整个 LSM RUM 树中的更新次数，而真空清理器则记录内存 R 树中下一个要清理的叶节点。当更新操作使全局计数器达到某个阈值时，我们使用真空清理器清理下一个节点。节点和更新备忘录的清理方式与算法 4 中相同。该节点清理完成后，我们将全局计数器重置为 0，并将真空清理器设为下一个叶节点。我们可以跳过最近已由缓冲清理策略清理过的节点。通过调整全局计数器的阈值 th，可以改变真空清理的频率。如果内存 R 树的叶节点数量为 N，则最多需要 $N \times th$ 次更新才能完全清理 R 树。例如，较小的 th 会带来更短的完整周期，如图 4 所示。根据 R 树的容量，需要调整 th 以获得最佳性能结果。Sim-

与缓冲清理类似，较低的 th 值会导致 VacuumCleaning 执行频率降低，并可能造成过多开销。根据内存中 R 树的大小及其节点容量，需要进行微调，以便高效清理 R 树并避免额外开销。

缓冲与真空清理策略有几个优点。它们有助于减小 UM 大小。内存 R 树中存在过时对象会导致不必要的 UM 条目。通过清理 UM，我们可以限制其大小。此外，这两种清理策略都会清理内存 R 树的节点。更新密集型工作负载会导致 R 树节点中出现许多过时对象，进而造成不必要的 R 树节点分裂。通过清理内存 R 树中的节点，可以使节点保持新鲜，避免因过时对象填满节点而导致不必要的分裂。此外，通过清理 R 树节点，我们预计能够提升搜索性能，因为搜索 R 树时，假阳性候选项（叶节点中的过时条目）应该会更少。

4.3 刷新时清理

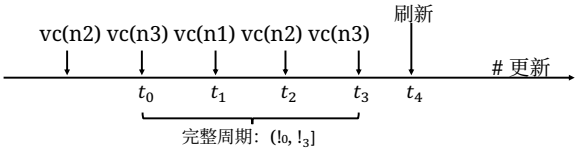


图 4：真空清理的清理周期

缓冲与真空清理策略用于清理更新备忘录和内存 R 树。对于基于 LSM 的索引，这两种方法有时不可避免地会漏掉过时对象，因而导致更新备忘录大小增加。例如，当 LSM 决定将内存内容转储到磁盘，而内存 R 树尚未完全清理时，就会出现这种情况。我们提出了刷新时清理技术，该技术将在 LSM RUM 树的刷新操作期间执行。假设内存 R 树中有三个节点，真空清理按轮询方式对 R 树节点运行，且缓冲清理处于关闭状态。在采用基于组件的刷新的 LSM RUM 树中，当内存组件的大小超过其容量时，组件就会刷新到磁盘。在图 4 中，随着更新在 LSM RUM 树中不断累积，定期进行真空清理（即 vc(node)），并在 t_4发生刷新操作。请注意，尽管在 t_3时已完成一轮完整的真空清理周期，但在刷新时刻 t_4仍会有过时对象。具体而言，在上一轮真空清理周期结束时（t_3），所有过时对象

在 t_1 之前生成的对象都会被清理。然而，有些过时对象（例如，在 t_1 和 t_3 之间于节点 $n1$ 中生成的对象）在 t_4 时不会被清理，因为上次清理 $n1$是在$t1$时。此外， $t3$ 之后的所有过时对象也都不会被清理。内存组件刷新到磁盘后，修改磁盘组件的代价很高，因为这会导致额外的磁盘 I/O，甚至会违反 LSM 策略：LSM树的磁盘组件是不可变的。

因此，我们提出了刷新时清理策略，在对象被移至磁盘之前清理过时对象。采用刷新时清理策略时，我们会在刷新时使刷写组件保持最新，同时避免产生过多计算开销。具体而言，我们在刷新之前增加 R 树和 UM 的清理步骤。当 LSM RUM 树刷新到磁盘时，会按照 Z序或希尔伯特曲线 [14,20] 对对象排序，以便高效地将其存储到磁盘。采用刷新时清理策略时，在刷新操作对对象排序的过程中，我们按照算法 4，将对象与 UM 中的条目进行比较，以检查对象是否过时。如果对象已过时，我们就丢弃该对象，使其不会被刷新到磁盘，并清理对应的 UM 条目。如果 UM 条目的 cnt 为 0，我们就从 UM 中移除该条目。

我们预计，刷新时清理策略能以多种方式提升性能。首先，清理会缩小 UM大小。缩小 UM大小是所有清理策略的首要任务。此外，刷写组件中的所有对象在刷新时都是新对象。因此，在某些情况下，刷写组件的大小可能会大幅缩减。例如，如果内存 R 树中的大多数对象都已过时，只有少数对象是新对象，那么已刷写 R 树的大小将与新对象的数量成正比。设想一种极端情况：来自一百个不同对象的更新达到了百万次。采用刷新时清理策略后，已刷写 R 树中只会包含一百个新对象，而不是百万个。因此，由于刷写组件更小，搜索已刷写的 LSM RUM 树所需的磁盘I/O会更少，搜索性能也将显著提升。

4.4 合并时清理

尽管采用了内存清理策略并在刷新时进行清理，磁盘中仍可能存在过时对象。例如，如果磁盘 R 树中的某个位置有一个新对象 O_{disk}，且该对象包含某些 O_{id}，那么对 O_{id} 的任何更新都会使 O_{disk} 过时。在移动对象应用中，这种情况很可能发生，因为对象的位置会被频繁更新。**因此，磁盘中的对象组件会随着新的更新不断累积到内存组件中而变得过时。**

我们提出合并时清理这一清理策略，用于清理磁盘上的过时对象及其对应的 UM 条目。直接修改磁盘组件成本高昂，也不适用于 LSM。因此，我们利用 LSM 合并操作。LSM 合并操作的目标是通过合并来限制磁盘组件的数量，从而控制搜索时间。磁盘组件过多会降低搜索性能，因为搜索需要检查所有基于磁盘的 R 树。进行合并操作时，会对多个磁盘组件进行批量加载，并将它们合并为一个磁盘组件。采用合并时清理时，我们会在合并操作的批量加载过程中执行一个验证步骤，如算法 4 所示。批量加载现有组件时，我们会实时检查每个对象，并将其与 UM 条目进行比对。如果某个对象已过时，就将其丢弃。显然，我们会以与刷新时清理及其他清理策略类似的方式，相应地更新 UM 条目。

合并时清理只需对 LSM 合并操作进行少量修改，即可清理磁盘组件中的过时对象。请注意，使用合并时清理执行合并后，如果系统中只有一个磁盘 R 树，就能保证系统中不会存在过时对象（即更新备忘录为空）。如果与刷新时清理结合使用，由于访问磁盘上更少且更紧凑的 R 树，两者都能提升搜索性能。实验结果表明，与验证策略相比，同时使用刷新时清理和合并时清理可使查询性能提升 400 倍以上。

5 并发控制

带有闩锁协议的 LSM RUM 树内存组件会保护树中的节点，使其能够在多线程环境中执行并发读写操作。遍历 LSM RUM 树时，线程必须获取/释放节点上的闩锁，以安全地执行搜索或插入/更新操作。例如，AsterixDB 分别遵循 ARIES/KVL [18] 和 GiST [15] 中描述的闩锁协议，用于 B 树和 R 树。传统的 LSM树虽然通过闩锁协议保护其内存组件，但引入更新备忘录后，要支持对 UM 的并发操作便面临新的挑战，因为数据结构一旦损坏（例如 UM 条目的 ts 或cnt 不正确），就会导致搜索结果错误或 UM 清理失败。本节将演示如何在 UM 中实现并发性。

5.1 更新备忘录中的并发控制

我们将介绍更新备忘录如何在多线程环境中支持并发操作。如第 3.1 节所述，UM 基于哈希映射结构，其中 UM 的键是对象标识符（ O_{id} ），值则是一对数据：最近的时间戳（ts）和过时对象计数器（cnt）。当现有对象被删除或更新时，它便成为 LSM 树中的过时对象，因此我们会将 ts 更新为当前时间戳，并递增 cnt ，对应于 O_{id}的 UM 条目。相反，如果我们像第 4 节所述执行 LSM 感知的 UM 清理，就会递减 O_{id}的 UM 条目中的 cnt ，因为清理会移除过时对象。

算法5： 比较并在较小时交换

```
输入： O_id:对象的键
       val：待比较的时间戳
1 ts ← 更新备忘录[O_id].ts;
2 重复
3     curr ← ts.get();
4     如果 curr ≥ val 则
5         返回 curr;
6 直到 ts.compareAndSet(curr, val) is true;
7 返回 val;
```

为了支持更新备忘录中的并发，首先，我们需要将更新备忘录条目中的 ts 和 cnt 设为原子整数，以便任何线程都能无冲突地读取或写入其值。此外，我们引入一种新的原子操作原语，比较，如果较小则交换（简称 CILS），用于更新条目的时间戳值。CILS 原子操作是比较并交换（CAS）操作的一种变体 [13]。CAS 将内存位置的当前值与给定值进行比较，只有在两个值相同时才会修改它。然而，在我们的策略中，当多个线程尝试更新 ts 的同一内存位置，且最后一个成功执行 CAS 的线程持有的并非 ts的最高值时，CAS 就会失败。如果并发线程 th_1 以 $ts = 4$和 th_2 以 $ts = 5$ 尝试更新同一内存位置，且 th_1 比 th_2更晚成功，那么 ts 最终会变成 4，而这并非最高的 ts。顾名思义，CILS（m、 v_{old}、 v_{new}）操作会将 m （即一个内存位置）处的 v_{old} （即旧值）与 v_{new} （即新值）进行比较，并且仅当 v_{old} 小于v_{new}时才修改 m 。这是因为更新备忘录始终需要保存最新对象的时间戳值；对于具有相同键的对象，该值必须是其中最高的（即最新的时间戳）。

算法5展示了如何在更新备忘录上实现 CILS。在第 1 行，线程获取将与给定的 val 进行比较的原子整数变量（ts）。之后线程以原子方式获取其值（$curr$）（第 3 行），并将其与 val 进行比较。如果 $curr$ 大于或等于 val，则表示另一个线程已经用更大的值更新了时间戳，因此我们中止 CILS 操作（第 4–5 行）。如果 $curr$ 小于 val，线程就尝试将 $curr$ 更新为 val （第 6 行）。如果 CAS 操作（第 6 行）失败（即 $ts.compareAndSet()$ 为 false），线程就会重复这些操作，直到成功为止。使用 CILS 时，对于给定的对象，我们始终保留最大的（最新的） ts 。

算法 6： 更新备忘录更新

```
输入： O_id：对象键
       ts：已更新对象的时间戳
1 ret ← 更新备忘录.putIfAbsent(O_id, (ts, 1));
2 如果 ret is not null 则
3     curr ← 更新备忘录.CILS(O_id, ts);
4     check ← UM.putIfAbsent(O_id, (curr, 1));
5     如果 check is not null 则
6         UM.atomicIncCnt(O_id);
```

如第3.2节所述，LSM RUM 树中的删除/更新操作需要添加或更新 UM 条目。算法 6 展示了在插入/更新操作期间如何添加或更新 UM 条目。在将对象插入 R 树组件后，算法立即执行。在第 1 行中，我们对 UM 调用 putIfAbsent()，其键为 O_{id} ，值为 (*ts*, 1)。这是为了在给定的 O_{id} 不存在于 UM 中时，以原子方式初始化该条目；当多个线程同时执行 putIfAbsent() 函数时，只有一个线程会获得 $null$ 作为返回值，而其他线程将获得针对 O_{id} 的某个映射值（即 (*ts*,*cnt*)）。如果该条目存在、 ret不是 $null$，并且我们按照算法 5 执行 CILS 操作，则会在 UM 中以原子方式递增 O_{id} 对应条目的 cnt （第 4–6 行）。

对于第 4 节中讨论的 UM 清理策略，我们将 UM 条目的 cnt 减一，并在其值等于 0 时将其移除。在清理操作期间，对象的 ts 保持不变，但我们只需对从 LSM RUM 树中移除的过时对象递减 cnt （算法 7）。

算法 7： 更新备忘录清理

```
输入： O_id：对象键
1 ret ← 更新备忘录.atomicDecCnt(O_id);
2 如果 ret == 0 ，则
3     UM.remove(O_id, 0);
```

5.2 实现与正确性

由于没有针对 CILS 操作的硬件支持（CPU 指令），我们需要像算法5中那样，使用基于 CAS 的自旋锁方法在高级语言中模拟该操作。接下来，我们需要使用线程安全的更新备忘录数据结构来支持并发。如算法5-7所示，由于多个线程会同时读取和/或写入，更新备忘录操作需要保证线程安全。$ConcurrentHashMap$ [4] 是 JAVA 中的一个示例，它通过为不同的哈希桶分别加锁，支持各种 HashMap 操作（例如 put()、putIfAbsent() 或 remove()）的线程安全执行。此外，变量 ts 和 cnt 需要支持原子操作，以递增和递减 cnt ，并在 CILS 操作中对 ts 执行 compareAndSet()。

为确保 CILS 和 UM 操作是线程安全的，我们将 UM 可处理的操作分为三种情况，并证明其正确性。

5.2.1 UM 条目的并发更新

假设 $UM[O_{id}].ts$（对象的当前时间戳）为 1，且 $UM[O_{id}].cnt$（ O_{id}当前过时对象的数量）为 1。进一步假设两个线程，即 th_1 和 th_2，按照算法 6 中的方式，以时间戳值 6 和 7 并发更新 UM。执行完第 1 行后， th_1 和 th_2 都获得相同的 ret =1。随后，它们按照算法 5 执行 CILS 操作。在第 3 行，两个线程的 $curr$ 相同，均为 1。因此， th_1 和 th_2 的并发执行都会跳过第 4 行和第 5 行，因为对于 th_1 和 th_2， val 分别为 6 和 7。接着，每个线程都在第 6 行执行 $ts.compareAndSet(curr, val)$。由于$ts$ 是原子变量，且 $compareAndSet()$ 操作仅在 $curr$ 等于 val时才会原子地将 val 设为相应值，因此 th_1 或 th_2 会有一个成功并退出 CILS。i) 如果th_1成功，当前的 $UM[O_{id}].ts$将被设为 6，然后退出 CILS 操作。此时，th_2 会继续下一轮循环。由于 $curr$ 现在已设为 6，而 th_2 的 val 为 7，它会跳过第 4 至第 5 行，并再次尝试执行 $compareAndSet(curr, val)$。 th_2 完成操作后， $UM[O_{id}].ts$会被设为 7，这就是 ts 的期望结果（即 O_{id}最近的时间戳）。相反，ii) 如果 th_2 在初始循环中先成功执行了 $compareAndSet(curr, val)$，那么 $UM[O_{id}].ts$会被设为 7。在这种情况下， th_1会在第二轮循环的第 5 行退出 CILS，因为$UM[O_{id}].ts = 7$已大于 th_1的 val，即 6。CILS 操作完成后，两个线程都会对 O_{id} 的 cnt 执行原子递增，最终值为 3，因为 th_1 和 th_2 各自将其递增了一次。即使有 N 个交错执行的线程并发访问同一个 UM 实体，无论 ts 顺序如何，每个线程都会递增 cnt，并且仅当 ts 小于某个线程的 ts 时才更新 $UM[O_{id}].ts$。因此，这正是预期的执行情况，并会使并发更新正确执行。

5.2.2 UM 条目的并发清理

假设 th_3 对 R 树的一个叶节点执行真空清理，以处理 O_{id}，而 th_4 执行刷新时清理操作，该操作也涉及 O_{id}。清理过程会从 LSM RUM 树中移除过时对象，但不会影响 UM 条目的 ts 。因此，我们只需对 UM 更新执行一次原子递减操作，递减 $UM[O_{id}].cnt$。两个线程 th_3 或 th_4 中的一个将获得 $ret = 0$。由于 atomicDecCnt()，该线程将是最后执行的线程。因此，只有这个最后执行的线程会移除 O_{id} 的 UM 条目。存在一种边界情况，即移除 UM 条目可能导致错误结果。我们将在下一节处理这种情况。

5.2.3 对 *UM* 条目的并发更新与清理操作

假设 LSM RUM 树中有两个对象 O_{id} 的副本（一个当前副本和一个过时的副本），并且两者都放置在内存组件中。假设对于最近对象和过时对象， ts 的值分别为 $\mathbf{9}$ 和 7。在这种情况下，当前 UM 条目（即 $UM[O_{id}]$）为 $\langle id, \mathbf{9}, 1\rangle$。当多个线程同时参与 UM 更新和清理操作时，我们需要检查并发的正确性（即，th_5 更新位于 $ts = \mathbf{10}$ 的 UM 条目，而 th_6 同时执行真空清理以移除过时的对象）。

i) 如果 th_5 在 th_6 的算法 7 的第 1 行之前到达算法 6 的第 4 行，那么显然，UM 条目将由 th_5 设置为 $\langle id, \mathbf{10}, \mathbf{2}\rangle$ ，随后再由 th_6 更新为 $\langle id, \mathbf{10}, \mathbf{1}\rangle$ 。

即使**ii)** th_5 在 th_6执行第 1–2 行之后执行第 6 行，UM 条目也会由 th_6变为 $\langle id, \mathbf{10}, \mathbf{0}\rangle$ ，随后又会因 th_6 尚未移除该 UM 条目（第 3 行）而变为 th_5 $\langle id, \mathbf{10}, \mathbf{1}\rangle$ 。 th_5 完成更新后， th_6 的第 3 行将中止，因为当前的 $O_{id}.cnt$ 不等于 0，这正是预期的结果。

最后，**iii)** 如果 th_6 在 th_5 执行第 3 行时完成第 1 至 3 行，那么 O_{id} 的 UM 条目已被 th_6 移除。因此，th_5 会继续在第 4 行重新设置 UM 条目。如果有多次更新，只有一个线程会使用 $curr$ 初始化 UM 条目，其他线程则只会原子地递增 cnt （第 5 至 6 行）。

这是正确且符合预期的行为，与i) 和 ii)一致。

6 性能研究

我们评估了 LSM RUM 树及其插入、删除、更新和搜索性能。所有实验均在一台运行 Mac OS 10.15.5 的计算机上进行，该计算机配备 2.3 GHz Intel Core i7 处理器、16 GB 内存和 512 GB SSD。我们使用了三个包含数百万个点的真实数据集：Gowalla [10]，BerlinMOD [11]，和 ChicagoTaxi [21]，分别包含 640 万、5600 万和 1500 万个点。对于这些数据集，其唯一键（对象 ID）的数量分别为 10.7 万、2000 和 5200 个，同时记录了这些对象随时间变化的位置。

我们在 AsterixDB [1]，中实现了 LSM RUM 树，并将其与 AsterixDB 中已有实现的 Eager 和 Validation 策略进行比较。对于 LSM RUM 树，我们参照 [6] 将内存 R 树的预算设置为 256 MB，页面大小设置为 2 KB。合并策略设置为带有阈值 $=5$ 的 前缀 策略。我们将 UM 的实现整合到已有的 LSM R 树中。为了研究 UM 本身的影响，我们没有在 LSM R 树上使用任何优化，例如布隆过滤器 [8] 或范围过滤器 [7]，因为这些优化与本文的研究重点相互独立。

我们使用以下记号来指代各种清理策略。更新备忘录 表示不采用任何清理策略的 LSM RUM 树，更新备忘录 +(清理策略)表示采用清理策略组合的 LSM RUM 树，其中 F、M、B 和 V 分别指刷新时清理、合并时清理、缓冲清理和真空清理。

与第 4.1 节和第 4.2 节一样，我们通过调整阈值来获得最佳整体性能。降低阈值会更频繁地清理 UM 条目和内存中的树节点，但需要额外的计算开销。根据经验，我们将缓冲清理的更新计数器阈值设为 4，将真空清理的全局计数器阈值设为 8。

6.1 更新性能

为了表示各种更新密集型工作负载场景，首先，我们按照数据对象的时间戳对其进行排序，然后将数据对象摄取到每个 LSM RUM 树中。每个数据对象的第一次摄取是一次插入，随后进行多次更新操作。我们测量完成所有插入和更新的数据摄取所需的总执行时间，不包括设置和数据馈送时间。在比较 LSM RUM 树与 LSM R 树时，为了仅突出显示 LSMR树的性能，我们排除 LSM R 树$Eager$ 策略中基于主键的点查询时间，仅将对象的旧位置以及对象删除和更新请求馈送进去。

图 5 对比了 LSM RUM 树机制与 LSM R 树的 $Eager$ 和 $Validation$ 策略，展示了三个数据集上的更新性能，包括更新过程、处理刷新操作所需的时间以及处理合并操作所需的时间。与第 4 节一样，LSM RUM 树清理策略会清理内存层和磁盘层中的过时对象，以及 UM 条目。在更新密集型工作负载中，清理过时对象能够提升更新性能，因为这可以避免 R 树中不必要的节点分裂，并最大限度地减少刷新/合并操作。表 1 给出了每个数据集的更新所需的刷新和合并操作总数。根据实验结果，可以看到，LSM RUM 树采用各种清理策略时，性能比$Eager$ 策略提升 3 到 9.6 倍，比 $Validation$ 策略提升 1.4 到 4 倍，具体取决于数据集。

策略	Gowalla		BerlinMOD		ChicagoTaxi	
	刷新	合并	刷新	合并	刷新	合并
Eager	3	0	28	6	14	3
Validation	3	0	30	6	10	2
UM	3	0	30	6	10	2
UM+F	3	0	30	6	10	2
UM+M	3	0	30	7	10	2
更新备忘录+刷新时清理	3	0	30	7	10	2
更新备忘录+刷新时清理	0	0	0	0	6	1
更新备忘录+刷新时清理	0	0	0	0	6	1

表 1：每个数据集的刷新和合并操作次数

特别要注意，刷新时清理（即 更新备忘录+刷新）在所有数据集中都产生了预期的清理效果。由于它会在刷新操作中丢弃过时对象，内存 R 树会比原始的未清理 R 树小得多。因此，图 5 和表 1 中的更新备忘录 +刷新 在执行相同次数的刷新操作时，将刷新操作的处理时间最多降低 60 倍。在更新密集型工作负载场景中，内存 R 树包含许多过时对象。通过在刷新过程中筛除过时对象，刷写组件的大小得以缩小，更新操作成本也大幅降低。

图 5b 展示了合并时清理（即 UM+M）将合并

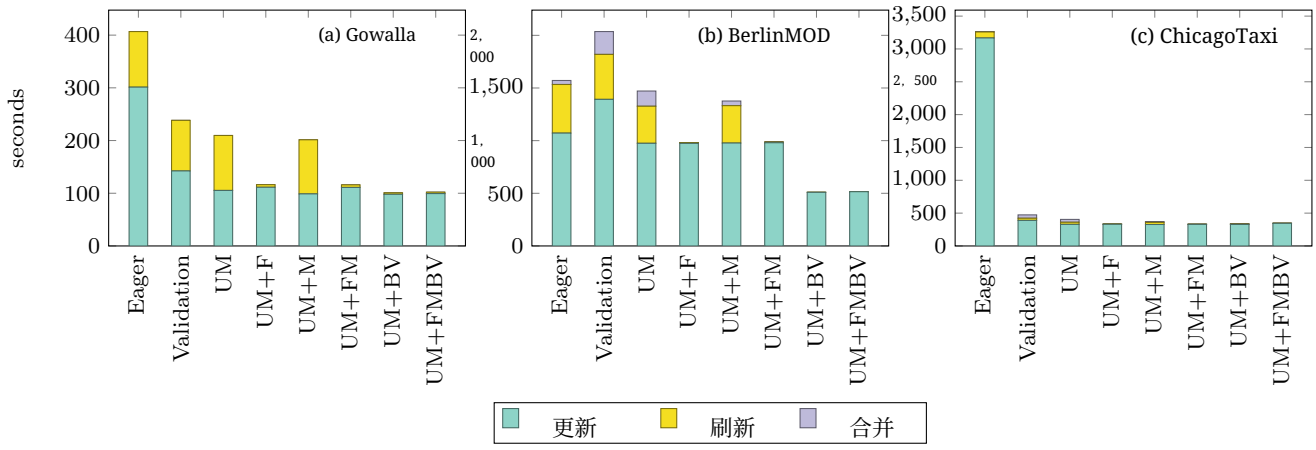


图 5：LSM R 树更新性能对比

操作所需时间缩短至 UM 的三分之一。请注意，*UM+M* 在磁盘层清理过时对象的效率有多高。*UM* 会合并磁盘层中的所有对象，但不进行清理；而 *UM+M* 则利用 UM 丢弃过时对象，因此合并后的 R 树规模小得多（也就是说，所需的磁盘 I/O 更少），尤其是在更新密集型工作负载环境中。通过缩短合并操作所需时间，UM+M 提升了更新操作的整体性能。

此外，请观察图 5b 中缓冲和真空清理策略（即更新备忘录*+BV*）的效果，与其他清理策略相比，其更新处理时间有所缩短。Gowalla 数据集包含用户的签到位置，ChicagoTaxi 数据集收集出租车行程的上车地点，而 BerlinMOD 数据集则跟踪每辆车的移动对象。随着 BerlinMOD 中的对象逐渐在 R 树覆盖的空间内移动，大多数旧位置和新位置的更新都落在相同的 R 树节点中。由于更新备忘录*+BV* 会执行内存中R 树清理，因此避免了过时对象导致的不必要节点分裂。因此，在 BerlinMOD 数据集中，更新性能提升了 2 倍。此外，内存清理策略还避免了刷新操作。表 1 列出了各数据集的刷新和合并操作次数。请注意，更新备忘录*+BV* 专门减少了移动对象（即 BerlinMOD）上的刷新/合并操作次数。与第 4.1 节和第 4.2 节所述相同，清理策略清理内存 R 树中的节点时，会延迟刷新操作。

表 2 列出了数据摄取过程中 UM 的最大大小。显然，在不采用任何清理策略的情况下，UM 的大小受 LSM RUM 树中已更新对象数量的限制。请注意，缓冲与真空清理策略可将 Gowalla 和 BerlinMOD 数据集中的 UM 大小最多减少 92%。由于 ChicagoTaxi 数据集收集的是乘客的上车点（即随机分布），缓冲与真空清理策略对 UM 大小的影响有所减弱。

策略	Gowalla	BerlinMOD	ChicagoTaxi
UM	99185	2000	5221
UM+F	74334	2000	5138
UM+M	99185	2000	4892
更新备忘录+FM	74334	2000	4790
更新备忘录+BV	7728	156	5188
UM+FMBV	7728	156	4967

表 2：更新操作期间 UM的最大大小

6.2 搜索性能

数据摄取完成后，我们从每个数据集中随机选取 100 个查询点，并测量获取查询结果的平均耗时。如 [16]，*Validation*策略由于需要额外的验证步骤，因此性能不如*Eager* 策略。图 6 展示了不同查询选择性下各数据集的搜索（查询）性能。总体而言，不进行任何清理的 UM 策略（即*UM*）的性能最高可比 *Validation* 策略提升 3 倍。虽然*UM* 在 Gowalla 数据集上的性能与*Eager* 策略相当（见图 6a），但在其他数据集上的性能则差了一个数量级（见图 6b 和图 6c）。尽管*UM* 驻留在内存中，但查询涉及的过时对象需要耗费大量时间来验证其状态。因此，如第 4 节所述，采用适当的清理策略至关重要。请注意，*UM* 的性能更好

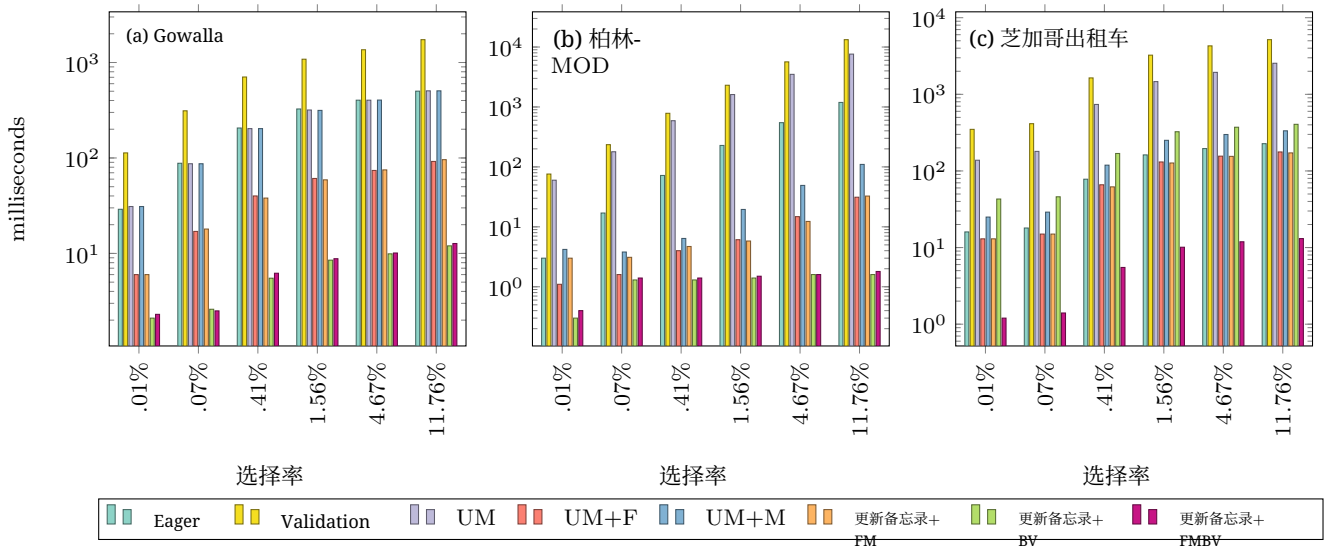


图 6：LSM R 树的查询性能比较

在所有数据集上都优于*Validation* 策略，因为其内存结构无需通过输入/输出进行验证。

性能的大幅提升归功于清理策略。如第 4 节所述，清理策略有助于减小 UM 的大小、清理过时对象，并缩小磁盘组件（即 R 树）的大小。首先，我们重点关注刷新时清理（即 *UM+F*）的效果。如第 4.3 节所述，*UM+F* 通过在刷新操作之前过滤掉过时对象，最大限度地减少了 R 树中过时对象的数量。因此，刷新到磁盘中的 R 树比原始内存 R 树更小。内存 R 树中过时对象越多，清理后刷写到磁盘中的 R 树就越小。如图 6 所示，*UM+F* 优于 *Eager* 和 *Validation* 策略，尤其是在具有较大查询范围的 BerlinMOD 数据集中，性能最高可提升 424 倍。由于 BerlinMOD 中对象的持续移动导致大多数对象变得过时，因此，对于 *UM+F* ，磁盘中 R 树的大小会显著减小，从而在搜索操作中需要更少的磁盘 I/O。

接下来，我们重点讨论图 6a 和图 6b 所示的缓冲与真空清理策略（即 *UM+BV*）的优势。由于能够清理内存 R 树中的节点，R 树的大小在更新密集型工作负载中不会显著增加，从而最大限度地减少刷新操作。因此，在表 1 的 Gowalla 和 BerlinMOD 数据集中，*UM+BV* 不会执行刷新/合并操作，查询操作也不会产生磁盘 I/O。

由于更新密集型工作负载会使磁盘上的大多数对象过时，因此很可能合并时清理（即，更新备忘录+合并）会清理磁盘上大多数过时的对象。与其他策略结合使用时，最佳性能为 更新备忘录+*FMBV* ，如图 6c 所示；在所有清理策略中，它在较高选择率下的性能比 更新备忘录 高出 200 倍。请注意，内存侧和磁盘侧清理策略对查询性能都至关重要；内存侧清理可减少过时对象（即降低假阳性查询结果），而磁盘侧清理则可减少输入/输出操作。

6.3 混合更新与查询工作负载的性能

在实际应用中，更新和查询性能都很重要，只有这样才能提供稳定的服务。例如，共享车辆服务会更新 车辆的当前位置，并查询 特定范围，以识别可供乘客选择的候选车辆。另一个例子是天气应急应用，它会更新 用户的位置，并查询 应急区域，以便向受影响的用户发送紧急通知。对于 LSM R 树上的更新/查询混合操作，随着内存 R 树的大小不断增长并被刷新到磁盘，查询性能会有所下降。

在第6.2节的上一个实验中，我们在所有数据摄取完成后执行随机查询。在本次实验中，我们使用相同的三个数据集，但将查询与插入/更新操作交错进行。为了模拟使用该数据集的真实更新/查询场景，我们按照第6.1节中的说明，按时间戳对数据对象排序，然后选取1%的

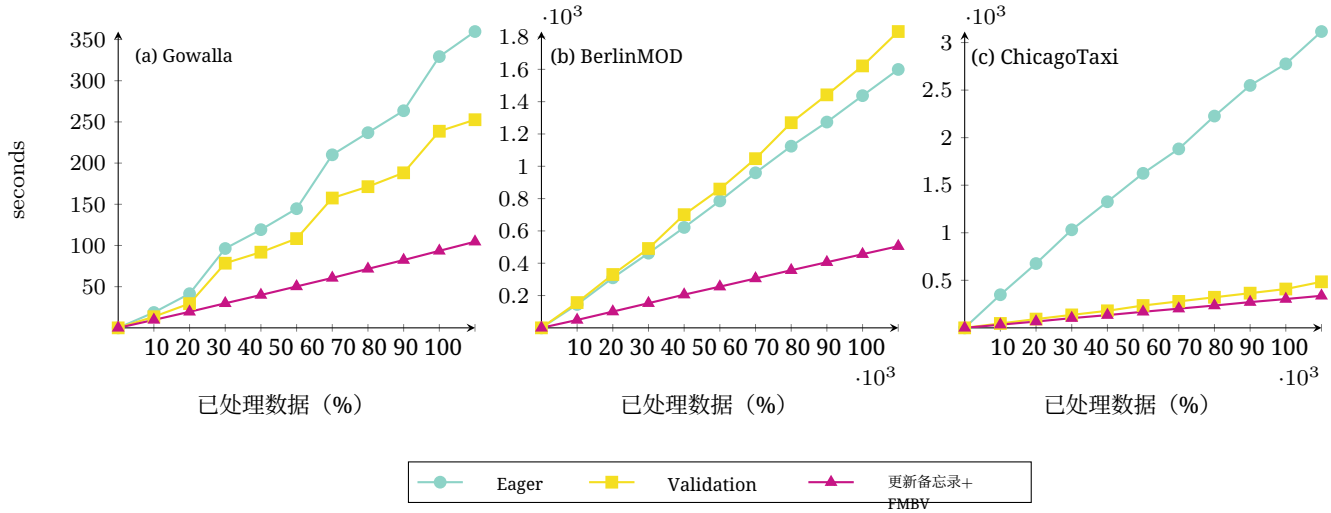


图 7：累积更新处理时间

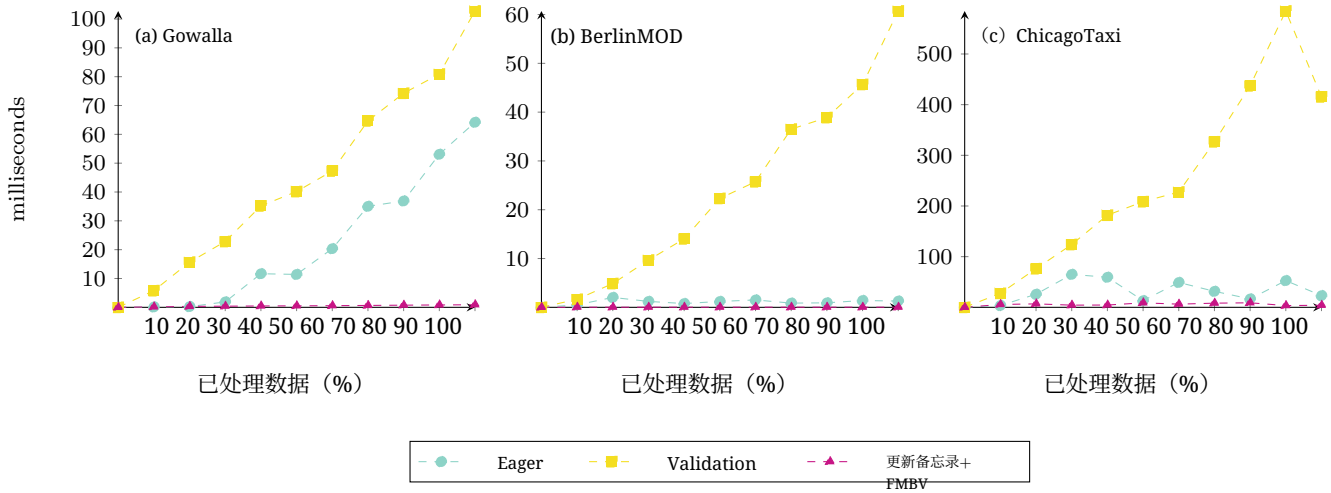


图 8：数据摄取期间的平均查询处理时间

随机对象，并使用其位置作为查询点，而不是更新对象的位置。查询范围在数据空间的0.01%以内均匀变化。换言之，Gowalla共包含640万个对象，因此约有6.4万个（1%）查询与630万个（99%）更新交错进行。BerlinMOD和ChicagoTaxi数据集也采用相同方式。图7、图8和图9分别展示了每个数据集的累积更新处理时间、各数据处理阶段的平均查询处理时间，以及执行更新和查询操作的累积处理时间。我们比较更新备忘录+*FMBV*与更轻量的和*Validation* 策略。更新备忘录+*FMBV* 在第6.1节和第6.2节中总体表现最佳。

从图 9 可以看出，由于验证步骤的计算开销较高，整体处理_超碰

Validation 策略的处理时间最长可达 UM+FMBV 的 6.7 倍，而后者的性能优于 Eager 和 Validation 两种策略。造成性能差异的原因如下。首先，如第3.2节所述，使用 UM 的更新过程更简单，因此性能有所提升。UM 中的更新操作简单直接，不会产生额外的维护成本。其次，刷新或合并操作会减慢更新处理速度。如图7a所示，Eager 和 Validation 策略的累积更新处理时间出现了三次跃升，这是由于刷新操作所致（即分别在已处理数据的 20%–30%、50%–60% 和 80%–90% 期间执行刷新）。然而，由于能够仅通过缓冲与真空清理策略清理节点，使用 Gowalla 数据集时，UM+FMBV 不会产生刷新操作，而且

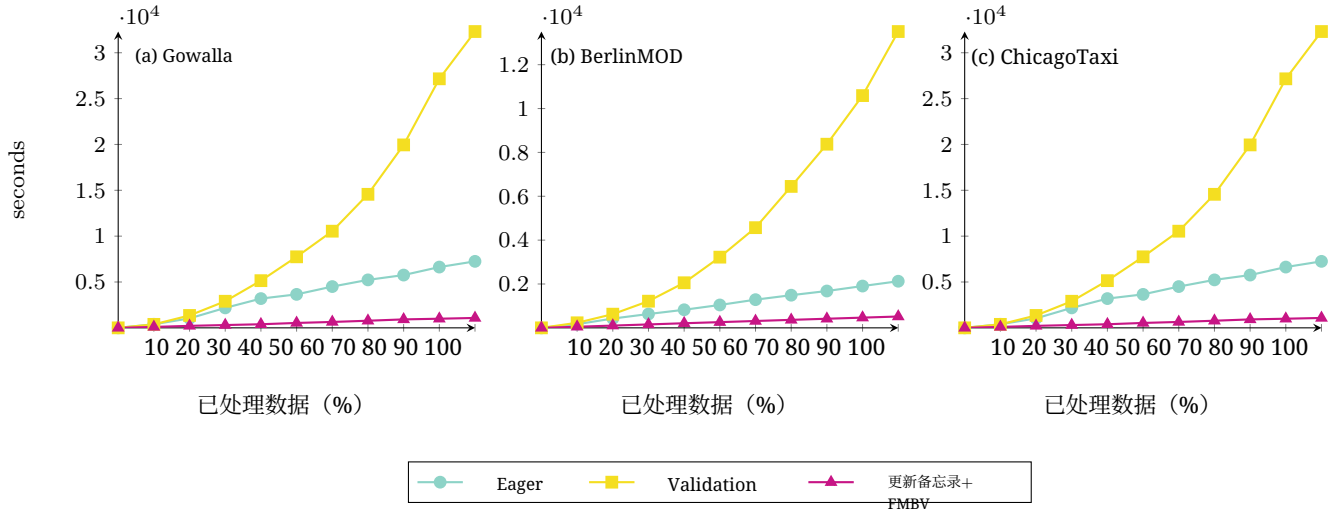


图9：累计处理时间，其中每个数据集执行99%的更新和1%的查询

还会采用真空清理策略。此外，可以看到，两种清理策略都通过避免内存 R 树因过时对象而发生不必要的节点分裂，提升了更新性能。尽管 UM+FMBV 在 Berlin-MOD（图7b）和 ChicagoTaxi（图7c）数据集上会执行刷新操作，但刷新时清理能使更新密集型工作负载下已刷写R树的规模非常小，因此更新处理时间保持稳定。最后，随着更新不断堆积在 LSM R树中，Eager 和 Validation 策略下的搜索性能会下降。由于 R 树的规模不断增大、磁盘I/O增加，以及验证步骤的计算开销，LSM R树的搜索处理时间随之增加。

在图8a中，由于数据更新，所有策略的搜索性能均有所下降。尽管 *UM+FMBV* 不会产生磁盘 I/O，但内存中 R 树不断增大的规模会增加搜索处理时间。此外，正如图8a中的 Eager 和 Validation 策略所示，磁盘I/O对搜索性能至关重要。刷新操作后（接近数据处理量的30%、60%和90%时），搜索处理时间最多增加至原来的6倍。与 *Eager* 和 *UM+FMBV* 策略相比，*Validation* 策略在更新期间的搜索性能最差，具体见图8b和图8c，这是因为其验证步骤使用了主键索引。由于 BerlinMOD 和 ChicagoTaxi 数据集包含的不同对象数量较少（分别为2k和5.2k个唯一对象），搜索产生的候选项中包含许多过时对象，而使用主键索引 进行验证会导致索引遍历产生更高的计算开销，并可能带来更多磁盘 I/O。在图8b中，接近数据处理结束时，*UM+FMBV*（0.025毫秒）在搜索处理时间方面比 *Eager*（1.3毫秒）和 *Validation*（60毫秒）策略分别实现了52倍和2400倍的加速。采用 UM 策略时，备忘结构基于哈希映射，仅驻留在内存层中，因此验证步骤的计算开销非常小。总体而言，采用清理策略的 LSM-RUM树在更新和搜索性能方面均显著优于传统的LSM R树策略。

6.4 多线程支持

如第5.1节所述，我们提供了一种在 UM 结构上支持并发控制的机制，使其也能在多线程环境中运行。为分析其性能，我们改变运行 Eager 和 Validation 策略的线程数量，并将其更新性能与采用所有清理策略的 LSM RUM 树进行比较。我们使用相同的真实数据集 Gowalla、BerlinMOD 和 ChicagoTaxi，并为运行更新操作的每个线程提供相应的数据文件。

从图10可以看出，LSM RUM 树的性能优于之前的策略。对于 Gowalla 数据集，LSM RUM 树在 8 个并发工作线程下可处理 378.9 个对象/毫秒，其吞吐量分别是急切策略（73.5 个对象/毫秒）的 19.9 倍和验证策略（19.7 个对象/毫秒）的 5.2 倍。此外，将线程数从 1 个增加到 8 个后，LSM RUM 树的加速比达到 4.5 倍，远高于验证策略；后者的性能仅提升了 1.7 倍。来自 BerlinMOD 数据集和

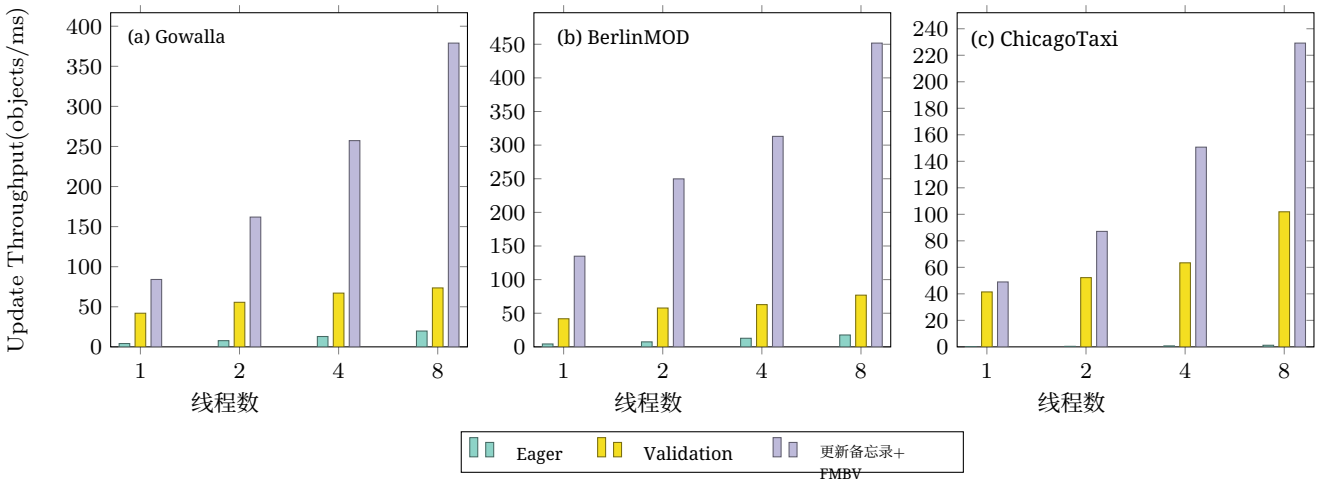


图 10：多线程环境下更新性能的比较

ChicagoTaxi 也类似。LSM RUM 树在两个数据集上的吞吐量分别达到 451.7 和 229.2 对象/毫秒。相比之下，验证策略仅达到 73.5 和 19.7 对象/毫秒，而急切策略达到 101.9 和 1.2 对象/毫秒。使用 LSM R 树时，LSM RUM 树明显优于验证策略和急切策略。

7 结论

本文介绍了一种新的索引结构，称为 LSM RUM 树。它将内存中的更新备忘录结构（UM）与基于 LSM 的 R 树结合起来，以提升更新密集型空间数据工作负载的更新和搜索性能。我们说明了如何在 LSM RUM 树中利用 UM 执行删除、更新和搜索操作。内存中的 UM 结构能够在查询处理过程中高效地进行验证，并简化更新操作。为了使 UM 能够常驻内存，缩小其体积至关重要。为此，我们提出了四种 UM 清理策略：缓冲清理、真空清理、刷新时清理和合并时清理。这些策略不仅通过清理 UM 条目来缩小其体积，还能缩小磁盘 R 树的体积，从而提升搜索性能并减少输入/输出开销。性能研究表明，LSM RUM 树能够高效处理更新密集型工作负载，其性能优于 LSM R 树中的 *Eager* 和 *Validation*策略。此外，搜索性能也有所提升。我们计划拓展这项研究，探索更新备忘录如何提升 R 树以外其他二级索引（例如作为二级索引的 B^+ 树或倒排索引）在更新密集型工作负载下的性能。

致谢 Walid G. Aref 感谢美国国家科学基金会（National Science Foundation）通过资助编号 IIS-1910216 提供的支持。

参考文献

1 http://asterix.ics.uci.edu/.
2 AsterixDB。网址。Cassandra。网址 http://cassandra.apache.org/。LevelDB。
3 网址 https://github.com/google/leveldb/。
4 https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html Oracle JAVA ConcurrentHashMap。网址
5 。
RocksDB。网址 https://rocksdb.org/。
6 Alsubaiee, S., Behm, A., Borkar, V., Heilbron, Z., Kim, Y.S., Carey, M.J., Dreseler, M., Li, C.：AsterixDB 中的存储管理。VLDB 基金会会刊 (PVLDB) **7**(10), 841–852 (2014) 7. Alsubaiee, S., Carey, M.J., Li, C.：基于 LSM 的存储与索引：一个因时而获益的旧想法。载于：第二届 ACM 国际丰富地理空间数据管理与挖掘研讨会 (GeoRich)，第 1–6 页 (2015) 8. Bloom, B.H.：允许误差的哈希编码中的空间/时间权衡。ACM 通讯 **13**(7), 422–426 (1970) 9. Chen, S., Jensen, C.S., Lin, D.：用于评估移动对象索引的基准测试。VLDB 基金会会刊 (PVLDB) **1**(2), 1574–1585 (2008) 10. Cho, E., Myers, S.A., Leskovec, J.：友谊与移动性：基于位置的社交网络中的用户移动。载于：ACM 知识发现与数据挖掘国际会议 (SIGKDD) 论文集，第 1082–1090 页。ACM (2011) 11. D¨untgen, C., Behr, T., G¨uting, R.H.：BerlinMOD：移动对象数据库的基准测试。VLDB 期刊**18**(6), 1335 (2009) 12. Guttman, A.：R 树：用于空间搜索的动态索引结构。载于：ACM 数据管理国际会议 (SIGMOD) 论文集，第 47–57 页 (1984) 13. Herlihy, M.：无等待同步。ACM 编程语言与系统汇刊 (TOPLAS) **13**(1), 124–149 (1991)

14. Hilbert, D.: ¨论一条曲线到平面区域的连续映射。《数学年刊》**38**, 459–460 (1891) 15. Kornacker, M., Mohan, C., Hellerstein, J.M.：广义搜索树中的并发与恢复。ACM SIGMOD Record **26**(2), 62–72 (1997) 16. Luo, C., Carey, M.J.：基于 LSM 的存储系统中的高效数据摄取与查询处理。VLDB 基金会会刊 (PVLDB) **12**(5), 531–543 (2019) 17. Mahmood, A., Aly, A., Kuznetsova, T., Basalamah, S., WG, A.：近期轨迹的基于磁盘的索引。ACM 空间算法与系统汇刊 (TSAS)**4**(3), 1–27 (2018) 18. Mohan, C., 等：ARIES/KVL：一种用于对 B 树索引执行多操作事务进行并发控制的键值锁定方法。IBM Thomas J. Watson 研究部 (1989) 19. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.：日志结构合并树 (LSM树)。《信息学学报》**33**(4), 351–385 (1996) 20. Peano, G.：论一条填满整个平面区域的曲线。《数学年刊》**36**(1), 157–160 (1890) 21. Portal, C.D.：2019 年出租车行程 (2019)。网址 `https://data.cityofchicago.org/Transportation/Taxi-Trips-2019/h4cq-z3dy` 22 。Shin, J., Wang, J., Aref, W.G.：LSM RUM 树：一种面向更新密集型空间工作负载的日志结构合并 R 树。载于：2021 年 IEEE 第 37 届国际数据工程会议 (ICDE)，第 2285–2290 页。IEEE (2021) 23. Silva, Y.N., Xiong, X., Aref, W.G.：RUM 树：使用备忘录支持 R 树中的频繁更新。VLDB 期刊 **18**(3), 719–738 (2009) 24. Xiong, X., Aref, W.G.：带更新备忘录的 R 树。载于：第 22 届国际数据工程会议 (ICDE)，第 22–22 页 (2006) 25. Xiong, X., Mokbel, M.F., Aref, W.G.：Lugrid：面向移动对象的更新容忍型基于网格的索引。载于：第 7 届国际移动数据管理会议 (MDM)，第 13–13 页 (2006) 26. Zhu, Y., Wang, S., Zhou, X., Zhang, Y.：RUM 树：一种支持频繁更新的新型多维索引。载于：国际 Web 时代信息管理会议 (WAIM)，第 235–240 页 (2013)