doi:10.3772/j.issn.1002-0470.2017.01.002

一种面向事务型数据库的无锁并发 B+tree 索引结构①

李 乔②* 赵鸿昊* 江 鹏* 张兆心**

(*上海期货信息技术有限公司 上海 200122)

(**哈尔滨工业大学网络与信息安全研究中心 哈尔滨 150001)

摘 要 为了克服现有多版本并发控制(MVCC)进行数据的并发访问控制中短暂阻塞的缺点,达到读写完全并发,提出了一种基于写时复制的多版本并发 B+tree(BCMVBT)索引结构。BCMVBT 通过复制分离读写的操作空间以使读写事务在任意时刻完全并发执行,规避比较与交换(CAS)操作带来的高 CPU 消耗,达到一写多读场景下的完全并发。同时针对现有多版本开发 B+tree(MVBT)范围查询的复杂操作,提出了无锁的 BCMVBT 的范围查询算法和回收机制,从而实现了索引的插入、查询、更新与回收的无锁并发操作。通过与事务型 MVBT(transaction MVBT)的对比,在读写并发环境下 BCMVBT 的时间消耗降低了 50%,实验进一步表明 BCMVBT 在大事务的场景下具有更高的优势。

关键词 事务,索引,B+tree(BT),多版本并发,写时复制(COW)

0 引 言

在多核多处理器时代,数据的并发访问控制是影响数据访问性能的关键因素。多版本并发控制(multi version concurrent control,MVCC)技术极大提高对历史版本的查询效率。近年来,云计算、大数据等数据分析研究热点已经引起学术界与产业界的广泛关注。大数据分析包含非结构化数据的融合分析和结构化数据的历史变化趋势分析,前者主要着力于非结构化数据的存储、合并与关联分析,而后者则关注同一类数据的时空变化规律。无论是逻辑层面还是物理层面,数据的并发访问控制都是影响数据访问性能的关键因素。在现实社会中,许多时序型中心化系统为了保证数据的安全性,在处理多写服务时均采用了排队系统对写序列进行串行化,在最终的处理平台上的每个时刻只接受单写操作,如拍卖系统、订票系统、金融交易平台等。因此在事务型中心化系统中,多写多读并发最终退化为一写多读的并发。

MVCC 的并发操作是由锁机制控制的,现有 MVCC 的锁机制还有缺陷,如存在短暂阻塞的缺点。为了克服锁的缺点,达到读写完全并发,本文提出了一种基于写时复制的(based on copy on write, BC)多版本并发(multi-version concurrency, MV) B+tree(BT)索引结构,简称 BCMVBT。BCMVBT 通过复制分离读写的操作空间,在任意时刻读写事务都并发执行,不需要闩锁且避免比较和替换(compare and swap, CAS)操作带来的高 CPU 消耗。在 BCMVBT 中,读写事务互不干扰,达到一写多读场景下的完全并发;同时本文针对现有多版本开发 B+tree(MVBT)范围查询的复杂操作,提出了互不阻塞的 BCMVBT 的范围查询算法和回收机制,从而实现索引的插入、查询、更新与回收的全并发操作。通过与 MVBT 的对比表明,BCMVBT 能有效减少读事务时间。

① 国家科技支撑计划(2012BAH45B01),国家自然科学基金(61100189, 61370215, 61370211)和国家信息安全计划(2014A085, 2015A072)资助项目。

② 男,1984 年生,博士;研究方向:分布式计算,网络计算;联系人,E-mail: li.qiao@sfit.shfe.com.cn

(收稿日期:2016-08-19)

1 无锁结构设计挑战

在并发控制技术中,串行化用于保证事务隔离性。典型的串行化操作依赖 2 阶段锁协议对读写冲突的事务进行调度,但锁协议将读写操作互相阻塞,尤其当读事务远大于写事务时,这种阻塞将严重影响系统并发性能。进而研究人员提出用多版本并发控制(MVCC)技术提高查询效率[1]。MVCC 采用与序列化方式不同的机制解决并发操作,利用空间换时间的思路,每次更新都建立新的数据版本从而不影响并发读事务对旧版本的操作。当前已有许多并发版本控制策略,例如传统的数据库及虚拟机中广泛应用的快照隔离技术[2,3]。

文献[4]对并发数据结构进行了详细的阐述,指出 wait-free,lock-free,obstruction-free 的区别:(1) Wait-free 是指线程必定在有限步骤内获得执行权;(2)Lock-free 是保证线程操作必定在有限步骤内执行结束;(3)Obstruction-free 是指在停止其他线程对计数器的引用之后,操作必定在有限步骤内完成。当前无锁策略主要分为以下两类:

(1)多版本数据结构(multi-version data structure),典型的应用是多版本 B + 树(multi-version B + tree, MVBT),该数据结构由文献[5]提出。可将 MVBT 视为由多个 B + tree 重叠产生的树形拓扑,事务操作根据事务版本号能够迅速定位入口,从而获取对当前版本可见的数据。当前对 MVBT 的改进主要包括:1)在 ImmortalDB 系统原型中提出基于时分的 B + Tree,但不考虑旧版本节点的回收[6];2)提出在针对事务的索引结构 TMVBT(Transaction MVBT),在写事务中最多锁住 5 个节点,降低 MVBT 结构上的阻塞面积,但在事务提交阶段仍需进行复杂的锁操作以保证数据正确性[7];3)采用比较与交换(compare and swap,CAS)操作,循环检测节点被读写状态[8],然而 CAS 操作过度频繁会耗费大量 CPU 资源,甚至导致线程在时间维度上出现真空等待,即在等待时间指数化退避过程中,所有试图获取操作权限的线程均处于等待期。以上文献都没有详细地阐述如何实现垃圾回收,没有提出快速易用的回收策略。

(2)写时复制(copy on write,COW)技术。COW 通过用空间换时间的方式提高读写的并发性,该技术已被广泛应用于各类工程项目,例如打开一份共享文件时,操作系统为每个用户均复制了一份,文献[9]提出基于 COW 的单机文件系统并发快照索引技术,但由于该文关注的是文件系统及快照,并没有考虑范围查询。现有研究表明多版本并发控制(MVCC)比快照隔离具有更好的操作性以及更高的效率[10]。

MVCC 的本质是在整个索引结构中存储多版本信息以此满足对不同版本数据的并发获取。上述的多版本策略在更新或回收时均阻塞读写,且锁操作在实际应用中不仅实现复杂且易导致死锁。

2 分 析

为了更精确地描述数据在索引结构中的变化并尽量避免混淆,首先定义本文中出现的术语。

定义 1 记录值(key data):插入索引结构中的记录值,不包含事务版本号。

定义 2 关键值(key):插入索引结构中的记录值及事务版本号,是多版本结构中唯一可确定位置的标识。

定义 3 删除操作(delete):修改索引值对应的事务版本号,并没有在结构中擦除该索引值,仅改变该记录值的可见性。

2.1 MVBT 分析

当前的 MVBT 结构是先通过版本号找到多版本树中的版本入口,然后根据记录值在该版本树中查询。如图 1 所示,记录值 a 在 t_1 时刻建立,在 t_3 时刻删除,在 t_4 时刻重新插入,$[t_3, t_4]$ 之间是不可见的。在 MVBT 中,关键值 A 一般被描述为 $<a, t_1, t_3>$ 和 $<a, t_4, *>$,表示 a 在 t_1 和 t_3 之间可见,在 t_4 之后也可见,但在 t_3 和 t_4 之间是不可见的,此时 MVBT 中有 2 个 a 同时存在。当关键值 A 被频繁进行 *delete* 和 *insert* 操作时,MVBT 中将保存多个 a。对于给定 t_i 的事务,通过 t_i 定位可见版本树的根节点,然后再根据关键值在该版本子树中查找对应的索引记录。

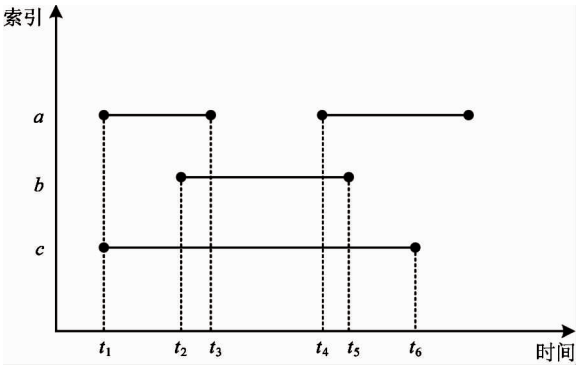

图 1　多版本数据时空示意图

图 2 展示了 MVBT 节点分裂和融合的过程:图(a)展示了事务 1 插入了 1、2、3 以及事务 2 删除 3、插入 8、9、10; 图(b)表示事务 4 新增了 3,由于之前的 3 对事务 4 是不可见的,因此需要新增 <3,4 > 记录,但一页最多容纳 3 条记录,节点产生分裂;图(c)显示经过分裂及融合后的版本树形态, <3,1 > 和 <3,1,2 > 合并, <3,4 > 作为新的版本入口。

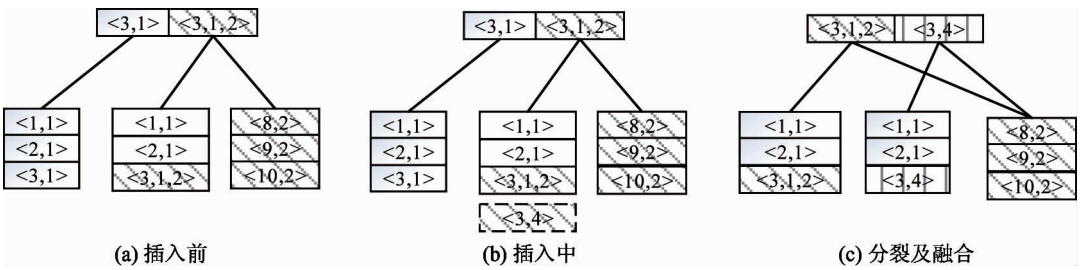

图 2　MVBT 节点分裂融合示意

由此可知,MVBT 可视为由许多版本 B + tree 叠加组合形成的树结构。正由于关键值的多版本性,MVBT 的回收机制需要对树节点进行融合以提升空间利用率。当回收操作与读操作进入相同节点时,需要进行互斥操作避免冲突。文献[11]指出在实际工程中,主要采用闩锁和 CAS 操作缩短读写阻塞时间提高并发性。

2.2　思　路

MVBT 适用于给定事务版本号的检索操作,但节点融合与回收操作的复杂性是其薄弱环节。BC-MVBT 采用写时复制的思想以达到读写完全并发的目的。如图 3 所示,读事务和写事务的入口均为元节点(entry)的旧版本根节点,两者区别在于写事务在获取旧根节点后将其复制为拷贝根节点,并修改 entry 中的新版本根号,之后写事务中的所有操作均从新根节点进入。在遍历路径中,写操作均将所有获取到的节点先复制再操作,而读事务则只操作旧节点。图 3 中的 T_w 表示写事务,T_w 在整个查询路径上获取到的节点均先拷贝一份,而后在新节点内进行操作。从图中可见所有的写操作都是在阴影节点上进行,读写事务互不干扰。

2.3　复杂度分析

本小节通过写操作对结构的变化进行定性分析。写操作可分为三种情况:

(1)写入新的索引记录,该索引的记录值是全新的,且不触发节点分裂。BCMVBT 与传统的 MVBT 一样,均沿路复制访问节点,最终将新索引记录值插入到复制后的叶子节点中;

(2)写入新的索引记录,该索引记录值是全新的,但产生节点分裂。最复杂的情况是叶子节点分裂之后,逐步上溯至根节点的路径均产生分裂。

(3)修改已有索引记录,该索引记录值不是全新的,旧版本中已存在。

MVBT 的节点分裂类型可分为两种:版本分裂与溢出分裂。前者是指进行删除操作时修改记录值可见性,从而需要将当前节点进行分裂,新产生的节点不包含被删除记录值,旧节点修改记录值版本,此时旧节点需要进行互斥阻塞避免读事务。溢出分裂与 B + tree 的分裂操作一致。正是由于传统 MVBT 优先匹配是版本号而非记录值,对于删除操作增加了额外空间和时间代价。

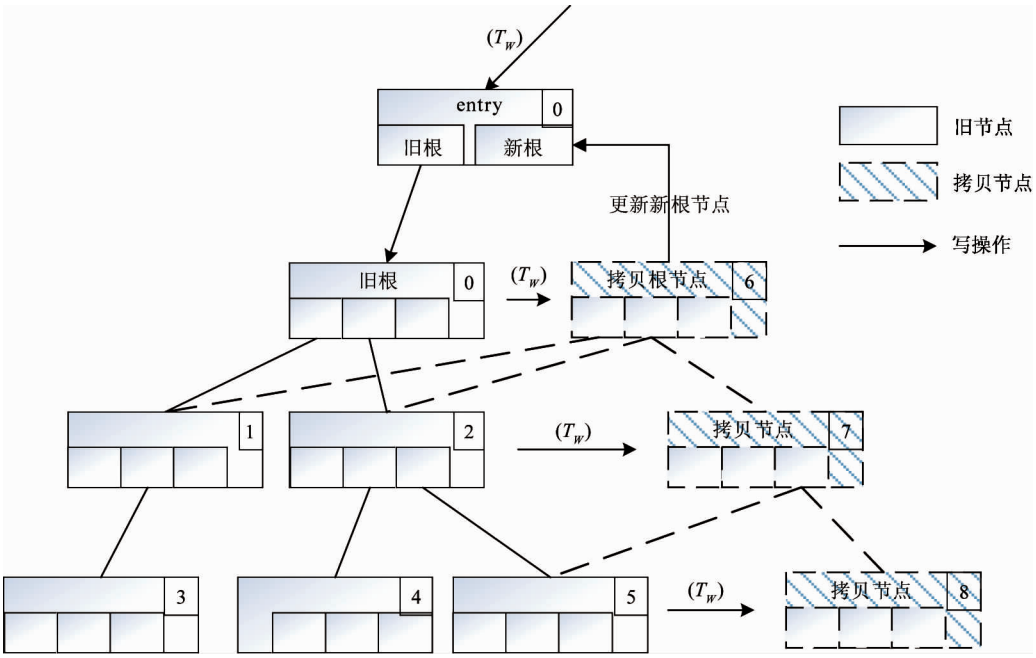

图 3　BCMVBT 结构示意图

不失一般性，假定访问单个节点的时间代价为 C_{na}，复制节点的代价为 C_{nc}，分裂节点的平均代价为 C_{ns}，树深度为 d，节点互斥的代价为 C_{ne}，融合 2 个节点的代价为 C_{nm}。

可以得出传统 MVBT 插入 1 条全新记录的代价，如下式所示：

$$C_{a1} = (d+1)\cdot(C_{na}) + (d+1)\cdot(C_{ns}+C_{ne})\cdot\alpha \qquad 0 \leqslant \alpha \leqslant 1 \quad (1)$$

其中 $(d+1)\cdot(C_{na})$ 表示定位索引过程的代价，从根节点到叶节点共计 d 个节点，当从叶子节点自下而上的分裂至根节点时，增加了树的深度，因此分裂的代价为 $(d+1)\cdot(C_{ns}+C_{nm})$。

写操作更新 1 条记录的代价 C_{a2} 如下式所示：

$$C_{a2} = (d+1)\cdot(C_{na}) + (2d+2)\cdot(C_{ns}+C_{ne}+C_{nm})\cdot\alpha \qquad 0 \leqslant \alpha \leqslant 1 \quad (2)$$

其中 $(d+1)\cdot(C_{na})$ 表示定位代价。由于节点的分裂和融合一般涉及 2 个节点，即当前节点、分裂后节点/被融合节点，分裂融合操作是自底向上进行，最终可能在根节点建立版本入口，由此可知共计涉及 $(2d+2)$ 个节点。删除 1 条记录主要过程分为三个步骤：

(1)查询定位该索引，访问路径从根节点至叶节点，长度为 d；

(2)在旧节点中修改关键值版本，进行版本分裂，复制旧节点，生成不包含该关键值的新节点，并自底向上逐步修改，最终在根节点建立新的入口；

(3)在修改旧节点过程中，由于关键值的删除，为了保持关键值顺序，还需进行节点融合[5]。

由此可知，在步骤(2)和(3)中额外产生互斥代价。

对于 BCMVBT 而言，插入全新记录的代价 C_b 如下式所示：

$$C_b = (d+1)\cdot(C_{na}+C_{nc}) + (d+1)\cdot(C_{ns})\cdot\alpha' \qquad 0 \leqslant \alpha' \leqslant 1 \quad (3)$$

更新 1 条记录的代价与插入全新记录在 BCMVBT 中的代价是一致的，这是由于 BCMVBT 是先通过关键值定位，而后根据版本号。仅从公式中并不能有效判断两种索引机制的优劣性，而只能断定 BCMVBT 的操作复杂度远低于传统的 MVBT。

假定在分析中两种结构的 d 相等，则可以得到插入全新关键值时两种结构的代价比为

$$W_1 = \frac{C_{a1}}{C_b} = \frac{(d+1)\cdot(C_{na}) + (d+1)\cdot(C_{ns}+C_{ne})\cdot\alpha}{(d+1)\cdot(C_{na}+C_{nc}) + (d+1)\cdot(C_{ns})\cdot\alpha'}$$

$$= \frac{C_{na} + \alpha C_{ns} + \alpha C_{ne}}{C_{na} + \alpha' C_{ns} + C_{nc}}$$

类似可以得出更新已有关键值的代价比为

$$W_2 = \frac{C_{a2}}{C_b}$$

$$= \frac{(d+1)\cdot(C_{na}) + (2d+2)\cdot(C_{ns} + C_{ne+C_{nm}})\cdot\alpha}{(d+1)\cdot(C_{na} + C_{nc}) + (d+1)\cdot(C_{ns})\cdot\alpha'}$$

$$= \frac{C_{na} + 2\alpha C_{ns} + 2\alpha C_{ne} + 2\alpha C_{nm}}{C_{na} + \alpha' C_{ns} + C_{nc}}$$

α 和 α' 是小于 1 的比例因子,用于表示分裂的程度,当其为 1 时,表示分裂至根节点,为 0 则表示不分裂。

为了简化分析,假定 α 和 α' 相等。本研究根据不同的场景分别讨论 W_1 与 W_2:

(1)单写单读。该场景下, W_1, 这是由于 B + tree 的节点大小数量级一般为 kB,例如 PostgreSql 的节点页默认大小为 4kB[12],拷贝较大块内存的代价一般高于闩锁,即 $C_{nc} > C_{ne}$。只有当写事务长时间占据某个页面阻塞读时,才导致读性能急剧下降。节点融合代价 C_{nm} 的代价一般不低于 C_{nc}, 这是由于融合时需要对两个节点所处内存区域进行整体遍历。W_2 在该场景下的数值主要取决于 α, 当 α 为 0 时,$W_2 < 1$,当 α 为 1 时,$W_2 > 1$,从而可知触发分裂时,BCMVBT 的读写性能均高于传统的 MVBT。

(2)单写多读。与(1)类似,该场景下, W_1, 但受到锁机制阻塞,读性能效率随着读线程数量增大而下降。W_2 变化与(1)基本相同。

根据以上分析可知,锁机制阻塞读操作,在大量读线程并发的场景下,读操作的效率明显下降,读线程的效率由于阻塞而降低;基于锁的策略效率随树结构变化增大而降低。可以得出以下结论:在单写多读场景下,随着写事务内操作数的增加,无锁 BCMVBT 的优势愈加显著,即读线程性能不随线程增多而降低,适用于对大量事务型数据库的数据分析。

3　实　现

本文提出的 BCMVBT 结构采用读写区域分离的思路进行设计,其主要解决传统 MVBT 改变结构状态的代价过大且阻塞读操作的问题。本节主要介绍 BCMVBT 结构中的插入、更新、范围查询及回收算法的实现。

3.1　插入

算法 1. 数据插入(Insert key)

Input:BCMVBT 元节点 *e*,当前事务号 *t*,记录值 *key*

Output:无

1. 读取节点 e,获取当前旧版本根节点 *old _ r*;
2. 复制 *old _* r 为 *new _ r*;
3. 调用 *btree _ search*(*new _ r*, *key*, *&leaf*),根据 *key* 找到叶子节点 *leaf*,并将查找路径上的节点都存入 *stack*,用于节点分裂时自底向上回溯;
4. 调用 *btree _ findloc*(*new _ r*, *leafid*, *key*),通过二分查找该 *key* 存储的位置 *loc*;
5. 调用 *insertkeytonode* (*new _ r*, *leaf*, *stack*, *key*,*loc*),将该 *key* 插入到叶子节点 *leaf* 内;insertkeytonode 与一般的 *btree* 插入数据到节点内部的操作一致,其中的节点分裂操作也一致,包括自底向上回溯 *stack* 中的每个节点。

和传统的 B + tree 的查询操作一样,首先从根节点开始查找,依次下降直到查找到对应的 *key* 所在位置,并将查找路径上访问过的节点均放入一个存储栈 *stack* 内,用于当叶子节点分裂时能够自底向上回溯分裂。算法 1 展示了插入过程。

3.2　更新

算法 2　数据更新(Update key)

Input:元节点 *e*,当前事务号 *t*,旧记录值 *oldkey*,新记录值 *newkey*

Output:无

1. 读取节点 *e*,获取当前旧版本根节点 *old _ r*;
2. 复制 *old _ r* 为 *new _ r*;
3. 调用 *btree _ search*(*new _ r*, *oldkey*, *&leaf*),根据 *key* 找到叶子节点 *leaf*,并将查找路径上的节点都存入 *stack*,用于节点分裂时自底向上回溯;
4. 若找到且可见(叶子中的 key 的创建事务号 *t _ old* < *t* 且删除事务号 *t _ del* > *t*),调用 *insert key tonode*,将 *newkey* 插入节点;
5. 否则,返回空。

更新过程如算法 2 所示,与插入过程类似,先通过 *key* 值查找到对应的叶子位置,修改旧 *key* 的删除事务号 *t _ del*,调用节点内插入算法将新 *key* 插入到叶子节点。本文的索引更新考虑到关键值长度的可变性,并不仅修改查找到的 *key* 的删除事务号,同时重新进行插入操作。因此更新操作和插入操作所需要的空间和时间是一样的:(1)均需要对查找路径上的节点进行复制;(2)均可能导致自底向上的节点分裂。

3.3　范围查询

范围查询是数据库的一个基本功能,文献[9]提出的 B-link tree 是单版本 B + tree 范围查询的最优方式。然而 B-link tree 正是由于在同层节点之间建立链接导致并发操作复杂,尤其在节点分裂时可能导致从叶子到根的整条路径需要锁维护并发的安全性。而 MVCC 的范围查询由于多版本数据的并存也导致查询效率迅速下降。本文的范围查询过程如图 4 所示,读事务进行[20, 74)之间的范围查询操作时,虚线箭头线段展示了整个查询路径走向。整个路径并不涉及写事务新复制的节点,从而保证并发的安全性。范围查询分为左值定位和右向遍历两个阶段,如算法 3,4 所示。整个范围查询的时间代价在最坏情况下为遍历整棵树。

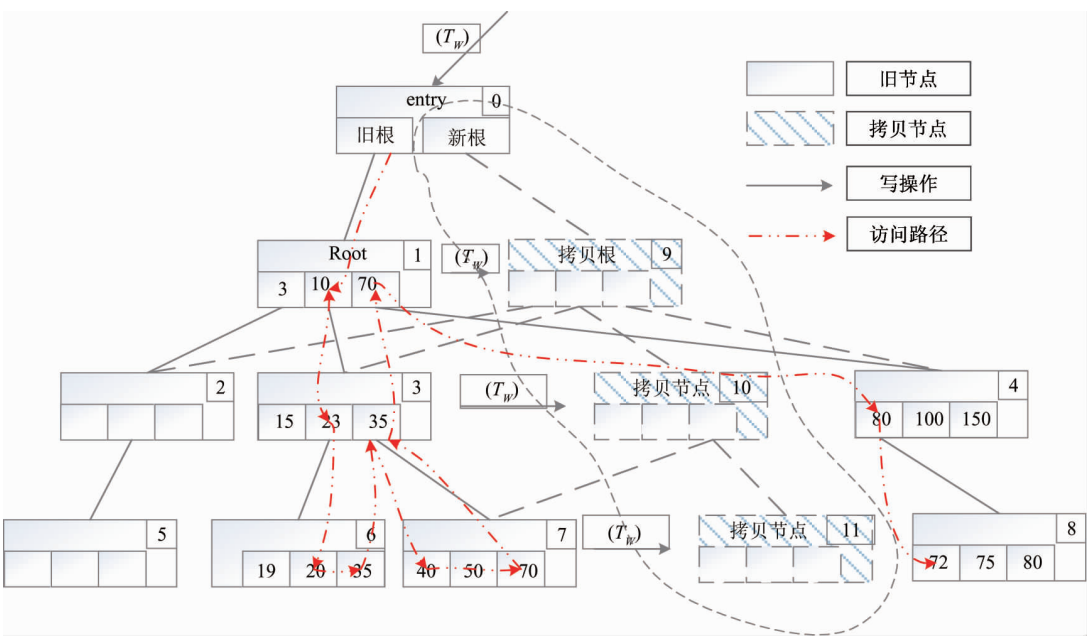

图 4　范围查询示例

算法 3　左值查询(Find mostleft)

Input:btree 元节点 *e*,当前事务号 *t*,左值 *leftkey*,右值 *rightkey*,

Output:*stack*

1. 读取节点 e,获取当前旧版本根节点 *old _ r* 的 *pageid*;

2. 根据 leftkey 定位范围最左的索引位置, 将路径上的节点以及节点内部的索引位置 *loc* 都存入 *stack*;

3. *while*(*stack* ! = *NULL*)

3.1 获取 *stack* 栈顶节点 *page*;

3.2 若 page 是叶子;

3.2.1 在 *page* 内进行二分查找满足大于 *lefikey* 且小于 *rightkey* 的最小索引位置 *loc*,并判断可见性;

3.2.2 若满足,*pushstack*(*stack*, *page*, *loc*, *false*),返回 *stack*;否则 *continue*;

3.3 否则:

3.3.1　若 *page* 的 *flag* = = *true*, 表示该节点及其孩子均没有被访问过,把相邻的下一个索引位置 *nextloc* 及当前 *page* 放入 *stack*;

3.3.2　若 *page* 的 *flag* = = *flase*,则压栈 *pushstack*(*stack*, *page*, *loc*, *true*), 并将 *page* 的 *flag* 设置为 *true*;并向下查询,依次将 *page* 压入栈;直到 *page* 为叶子,*break*。

算法 4　右向遍历(Getnext forward)

Input:*stack*, 当前事务号 *t*,右值 *rightkey*

Output:索引值 *key*

1. *while*(*stack* ! = *NULL*)

1.1.　获取 *stack* 栈顶节点 *page*;

1.2　若 page 是叶子:

1.2.1　在 *page* 内进行二分查找满足小于 *rightkey* 的最小索引位置 *loc*,并判断可见性;

1.2.2　若满足,*pushstack*(*stack*, *page*, *nextloc*, *false*),把相邻的下一个索引位置 *nextloc*,返回找到的 *key*;否则 *continue*;

1.3.　否则:

1.3.1　若 *page* 的 *flag* = = *true*,表示该节点及其孩子均没有被访问过,把相邻的下一个索引位置 *nextloc* 及当前 *page* 放入 *stack*;

1.3.2　若 *page* 的 *flag* = = *flase*,则 *pushstack*(*stack*, *page*, *loc*, *true*); 并向下查询,依次压栈;直到 *page* 为叶子,*break*。

3.4　回收

当前已有的 MVCC 回收机制主要通过节点融合进行,而节点融合通常采用细粒度锁进行,如节点级锁,或采用 CAS 操作循环检测节点状态避免阻塞。然而无论是锁或 CAS 操作均占用一定的时间代价或 CPU 代价。本文的索引结构采用写时复制的思想,将读写事务在空间上进行隔离,虽然占用更多的空间代价,但减少了操作复杂性与时间消耗。回收机制作为索引机制中的重要功能,若引入锁机制将降低插入性能。考虑到整个树结构的冗余性是由写事务的复制产生,因此只需要将已被复制过且复制该节点的事务大于所有读事务的死亡节点回收即可。如算法 5 所示,BCMVBT 的页面回收只需根据当前活跃事务及页面创建事务号即可,整个过程简单且易操作。

算法 5 索引节点回收过程(Getnext procedure)

Input:当前最小的读事务号 t

Output:无

1. 扫描索引页,获取该索引页 page 的创建事务号 t_p;
2. 若 $t_p < t$ 且该 page 被复制过且被复制的事务号 $t_{copy} < t$,则该页面可以被回收;
3. 否则跳回步骤 1。

4 试 验

本小节通过试验描述 BCMVBT 的性能,并通过多个指标对比测试传统 MVBT 与 BCMVBT。测试的硬件环境如表 1 所示。

表 1 测试环境

CPU	Intel(R) Xeon(R) 3.40 GHz
Memory	256GB
OS	Linux 2.6.32 Redhat 6.6

为了更详细对两种机制进行对比,本研究先给出测试数据描述及参数取值,如表 2 所示。

表 2 测试数据

符号	含义	取值
n	总索引数量	10000
m	一个事务的操作数目(插入与更新)	100
t	一个事务中全新索引数的比例	0%, 25%, 50%, 75%
B	页大小(表示 1 页最多存 B 个索引)	8192 byte
s	单个索引空间大小	60 byte

一般而言,单个索引大小远小于节点空间,当 m 较小时,导致在复制节点时将产生大量的重复索引记录,增加额外的开销;m 较大时,则复制节点的代价占比较低。根据第 3 节的分析可知,t 值的大小直接影响多版本结构的性能。为了更直观的对比 BCMVBT 与传统 MVBT 的性能,本文与文献[13]中提出的 TMVBT 结构进行比较。

图 5 展示了两者的时间代价,BCMVBT 在测试环境下的时间消耗比 TMVBT 降低了 40%。由于 TMVBT 读写锁的短暂阻塞以及更新操作引起的版本分裂消耗了较大的代价,BCMVBT 读写完全隔离且不存在版本分裂,在读写并发的情况下占优。

图 5 中的右图显示出 TMVBT 和 BCMVBT 的时间代价与更新操作占比均大致呈线性关系。为了更细致的分析 BCMVBT 的性能,本文分别测试小事务(100次插入/更新操作),中事务(1000次),大事务

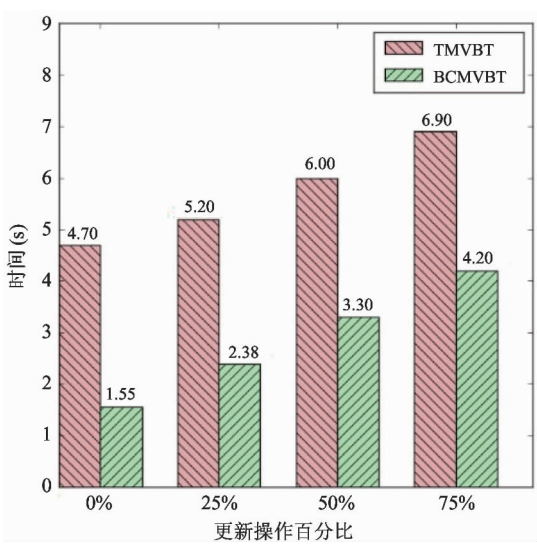

图 5 时间代价对比图

(10000 次)的代价,如图 6 所示。从图 6(a) ~ 图 6(c)中可知 BCMVBT 时间消耗几乎与事务数呈线性关

系,并随着更新比例的增大,时间消耗明显增加。这是由于在 BCMVBT 中的更新操作实际上包括 1 次查询和 1 次插入,更新操作比例的增大对时间消耗的影响也越大。图 6(d)展示了建立相同索引数目时,不同体积的事务所消耗的时间,从图中可看出,小事务和中事务的时间几乎一样,而大事务消耗的时间明显较低,这种差别主要受到拷贝页面次数影响。在测试数据中,1 个节点页容纳 1300 个索引,因此当 1 个事务中插入的新索引数目小于 1 页时,将会增加复制页面的数量。

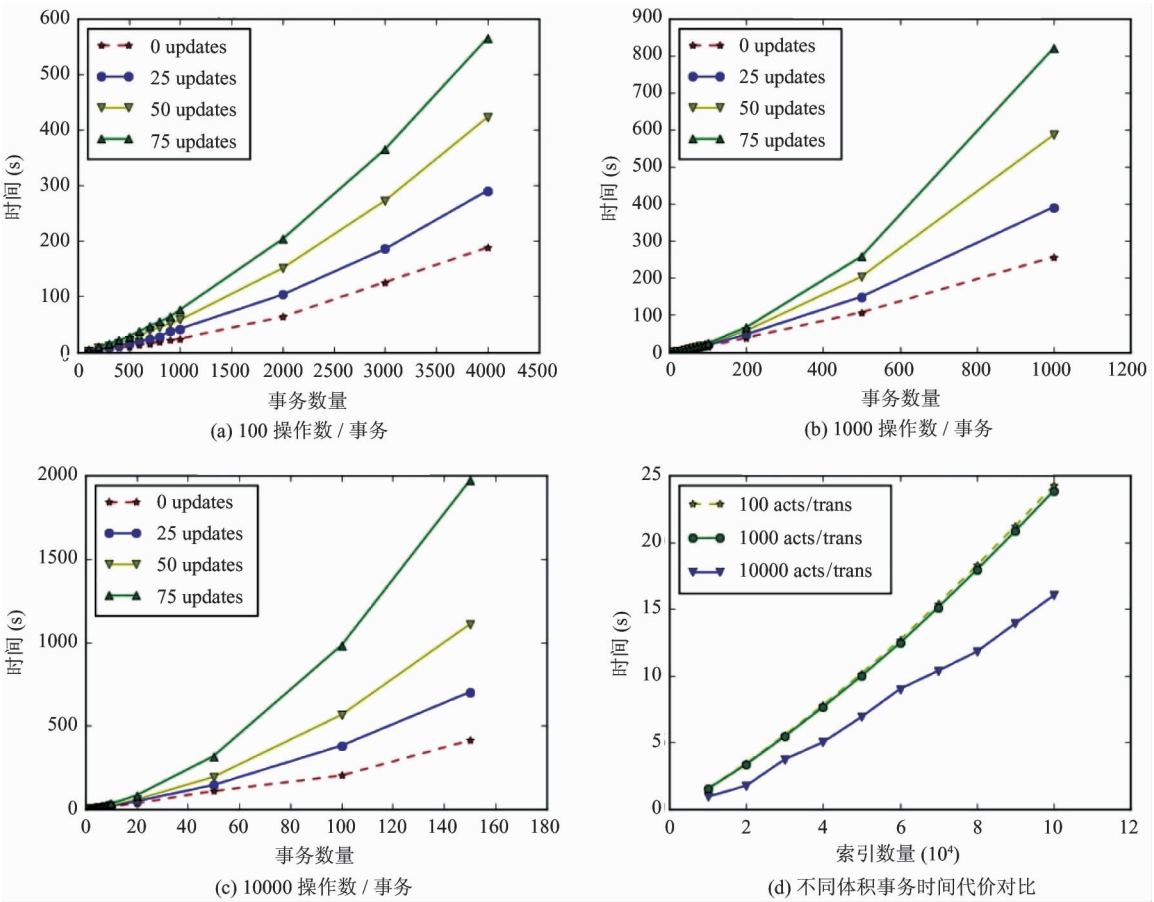

图 6　不同体积事务时间代价对比

与 TMVBT 相比,无锁的 BCMVBT 的垃圾回收机制具有更大的优势。我们首先给出回收率 φ 的计算公式:φ = 已回收页面数目/当前总页面数目。图 7展示了不同大小事务在运行过程中的页面回收率,回收率保持在 40% 以上。从图中可知当事务越小时,回收的页面越多,这是由于每个新事务都需要复制旧的页面,随着事务中的操作增加,复制的页面越少。由于回收机制采用定时方式进行页面扫描,因此回收率与读事务也相关,当读事务较大时,回收率较低。总体而言,在 BCMVBT 中的回收率在整个运行过程中较为稳定。

5　结　论

本文提出了一种基于复制的多版本 B + tree 索引结构,它采用写时复制的思想规避读写冲突,进而实现整个索引结构无锁并发。同时本文进一步提出了在基于写时复制的多版本并发 B + tree(BCMVBT)索引结构下的范围查询算法和垃圾回收机制。与传统的多版本并发 B + tree(MVBT)索引结构相比,本文范围查询算法时间复杂度最坏情况下与遍历全树一致,但实现机制简单,更适用于工程应用。

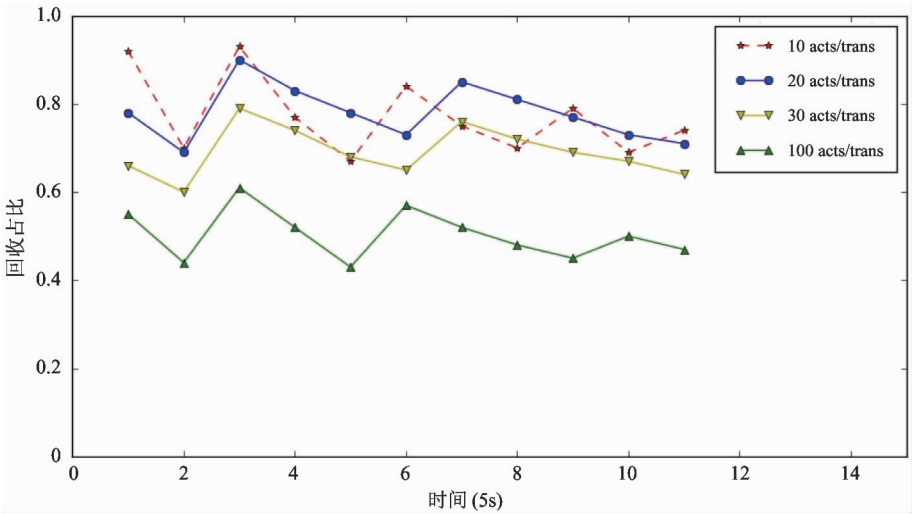

图 7 回收率对比

相比其他多版本结构,本文的垃圾回收机制的简单便捷性具有更大的优势。本文还将 BCMVBT 和 TMVBT 索引进行比较分析,说明 BCMVBT 在高并发读的场景下具有极大的优势,且更适用于大事务的单写场景。今后将进一步研究适用于分布式场景的 BCMVBT。

参考文献

[1] Mohan C, Pirahesh H, Lorie R. Effcient and flexible methods for transient versioning of records to avoid locking by read-only. *Transactions SIGMOD Record*, 1992, 21(2): 2

[2] Prokopec A. SnapQueue: lock-free queue with constant time snapshots. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Portland, USA, 2015. 1-12

[3] Fekete A, Liarokapis D, O'Neil E, et al. Making snapshot isolation serializable. *ACM Transactions on Database Systems* (*TODS*), 2005, 30(2):492-528

[4] Moir M, Shavit N. Concurrent Data Structures. Boca Raton: CRC Press, LLC, 2001. 1-2

[5] Becker B, Gschwind S, Ohler T, et al. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 1996, 5(4): 264-275

[6] Lomet D, Salzberg B. Access methods for multiversion data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, USA, 1989. 315-324

[7] Haapasalo T K, Jaluta I M, Sippu S S, et al. Concurrency control and recovery for multiversion database structures. In: Proceedings of ACM 17th Conference on Information and Knowledge Management, Napa Valley, USA, 2008. 73-80

[8] Braginsky A, Petrank E. A lock-free B + tree. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, Pittsburgh, USA, 2012. 58-67

[9] Rodeh O. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 2008,3(4): 2

[10] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Australia, 2015. 677-689

[11] Graefe G. A survey of B-tree locking techniques. *ACM Transactions on database systems*, 2010, 35(3): 16

[12] PostgreSQL. http://www. postgresql. org: PostgreSQL Global Development Group, 2016

[13] Haapasalo T, Jaluta I, Seeger B, et al. Transactions on the multiversion B + -tree. In: Proceedings of the 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, 2009. 1064-1075

A wait-free concurrent B + tree index scheme for transaction databases

Li Qiao[*], Zhao Honghao[*], Jiang Peng[*], Zhang Zhaoxin[**]

([*] Shanghai Futures Information Technology Co., Ltd, Shanghai 200122)

([**] Network and Information Security Research Center, Harbin Institute of Technology, Harbin 150001)

Abstract

In order to overcome multi-version concurrent control (MVCC)'s disadvantage of short obstruction in its concurrent control of data access to achieve the full read-write concurrency, a new index scheme based on copy on write (BC) with multi-version concurrency B + tree, called the BCMVBT, was presented. The BCMVBT uses the copy of the operation space for read-write separation to make the read-write transaction be concurrently conducted. Moreover, it avoids the CPU consumption caused by compare and swap (CAS) operations to achieve the full concurrency under write-once-read-many scenarios. Additionally, the current MVBT range query algorithm was improved and a recovery mechanism with wait-free manner was proposed in order to achieve full concurrency of insert/delete/recycle operations. Compared to TMVBT, BCMVBT reduces time cost by 50%. Further experiments show that BCMVBT is more efficient in large transaction environment.

Key words: transaction, index structure, B + tree (BT), multi-version concurrency, copy on write (COW)