

文章编号: 1000-5641(2018)05-0079-12

分布式日志结构数据库系统的主键维护方法研究

黄建伟, 张 召, 钱卫宁

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 目前在电子商务、社交网络、移动互联网等各类应用中存在大量的写密集型负载(例如, 电子商务的秒杀活动、社交用户生成的数据流等), 这使得基于日志结构的存储成为现代数据库系统中普遍的后端存储方式. 而基于日志结构的数据存储方式一般只支持追加操作, 高效的主键维护(主键的生成和更新)可以很好地提升数据库追加操作的性能. 此外, 在分布式和并发的环境中实现主键维护功能还要面临主键唯一性约束、事务性维护、高处理性能的挑战. 因此, 本文针对日志结构数据存储的特点, 研究了如何在分布式日志结构数据库系统中实现高效的主键维护功能. 首先, 我们提出了两类先读后写操作的并发控制模型; 其次, 我们应用这两类模型设计了几种高效的主键维护算法; 最后, 我们在自己的基于日志结构的分布式数据库系统 CEDAR 中实现了本文提出的主键维护方法, 并通过一系列实验验证了所提方法的高效性.

关键词: 日志结构; 分布式数据库; 主键维护; 并发环境

中图分类号: TP392 **文献标志码:** A **DOI:** 10.3969/j.issn.1000-5641.2018.05.007

Primary key management in distributed log-structured database systems

HUANG Jian-wei, ZHANG Zhao, QIAN Wei-ning

(School of Data Science and Engineering, East China Normal University,
Shanghai 200062, China)

Abstract: At present, there are a large number of writing-intensive loads (e.g., second-killing of e-commerce, social user-generated data streams) in many applications such as e-commerce, social networking, mobile Internet and so on, which makes log-structured storage a popular technique for back-end storage of modern database systems. However, log-structured storage only supports the append operation, efficient primary key management (primary key generation and update) functions can improve the performance of database append operations. In the distributed and concurrent environment, implementing primary key maintenance faces challenges, such as primary key unique constraints, transactional maintenance, and high-performance requirements. In light of the characteristics

收稿日期: 2018-07-04

基金项目: 国家自然科学基金(61432006, 61332006, U1401256); 国家863计划项目(2016YFB1000905, 2018YFB1003400)

第一作者: 黄建伟, 男, 硕士研究生, 研究方向为分布式数据库. E-mail: jwhuang@stu.ecnu.edu.cn.

通信作者: 张 召, 女, 教授, 研究方向为区块链系统和海量数据管理与分析.

E-mail: zhzhang@dase.ecnu.edu.cn.

of log-structured storage, this paper explores how to implement efficient primary key management in distributed log-structured database systems. First, we propose two kinds of concurrency control models for WAR (Write After Read) operations; second, we adopt these two models to design efficient primary key management algorithms; and finally, we integrate these algorithms into our distributed log-structured database, CEDAR, and verify the effectiveness of the proposed methods by a series of experiments.

Keywords: log-structured; distributed database; primary key management; concurrent environment

0 引言

随着近年来社交网络、电子商务、移动互联网应用的迅猛发展,对于存储系统的性能提出了极高的要求,尤其是对于写密集型关键任务(例如,电子商务的秒杀活动、社交用户生成的数据流等)更需要高效的支持.为了满足写密集型关键任务的存储需求,越来越多的存储系统采用基于日志结构(Log-Structured)^[1]的数据存储方式,例如 Google 的 BigTable^[2]、Apache HBase^[3]和 Cassandra^[4]等.

在数据库系统中,主键可以唯一标识一个数据单元(例如一个数据行),通过主键可以对数据进行访问和关联.而对于日志结构的分布式数据库系统来说,一般只支持对数据的追加操作,为追加数据快速生成一个唯一主键或高效地更新数据的主键值都可以很好地提升数据库追加操作的性能.然而,在分布式日志结构数据库系统中实现高效的主键维护(主键的生成和更新)还面临着一些挑战:(1)在分布式环境中,数据分布在不同节点的磁盘或内存上,要确保主键的唯一性约束;(2)在主键维护相关操作并发执行时要保证事务性以及冲突可串行化;(3)需要采取一定的优化措施使得主键维护功能达到较好的性能.

本文以分布式日志结构数据库系统的主键维护问题为出发点,总结了一类具有普遍意义的先读后写(Write After Read, WAR)更新操作,进而提出了两类先读后写操作的并发控制模型,最后应用这两类模型设计并实现了高效的主键维护功能.

本文的主要贡献如下.

1) 提出了基于封锁机制的 PCC-WAR (Pessimistic Concurrency Control for WAR) 模型和基于有效性验证的 OCC-WAR (Optimistic Concurrency Control for WAR) 模型去保证在分布式环境中 WAR 更新操作并发执行时的正确性.

2) 分别基于 PCC-WAR 模型和 OCC-WAR 模型设计了多种在分布式日志结构的数据库系统中实现主键维护功能的算法.

3) 在基于日志结构的分布式数据库系统 CEDAR^[5]中实现了本文提出的主键维护方法,并通过实验验证了所提方法的高效性.

本文其余部分的安排如下.第1节介绍了背景与相关工作;第2节详细介绍了先读后写的并发控制模型;第3节给出了日志结构的分布式数据库系统主键维护的多种实现算法;第4节给出了实验评估结果;最后在第5节对本文做出了总结.

1 背景和相关工作

1.1 日志结构的数据库系统

日志结构合并树(The Log-Structured Merge Tree, LSM-Tree)^[6]是一种用来优化数据库系统写性能的数据结构. LSM-Tree 在数据库系统中维护了两个或者多个树形结构,其中一

个常驻于内存存储中, 而其他的则维护在磁盘存储中, 这些结构中的数据可以通过批处理合并的方式高效地实现同步. 与传统的 B-Tree 结构相比, LSM-Tree 极大地提升了数据库系统的写性能, 因为其减少了磁盘随机读取的开销.

我们在分布式日志结构的数据库系统 CEDAR^[5]中实现了本文提出的主键维护方法. 图1给出了 CEDAR 的架构概览, 它是由一个内存事务引擎、一个分布式存储引擎和一个查询处理引擎组成. 一个增量节点 (T-node) 存储了所有新提交的更新数据 (本文称之为增量数据), 并且通过多版本和乐观并发控制 (MVOCC) 去支持事务管理. 众多的基线节点 (S-node) 组成了 CEDAR 的分布式存储引擎, 它们存储了大量的静态数据 (本文称之为基线数据). 增量节点中的内存存储和所有基线节点的磁盘存储构成了 LSM-Tree 的两层存储, 增量节点中的增量数据会通过数据压缩的方式合并到基线节点中. 增量节点和基线节点都采用数据副本去保证系统的高可用性. 多个查询节点 (Q-node) 构成了 CEDAR 的查询处理引擎, 它们负责处理客户端的查询请求和事务的业务逻辑执行.

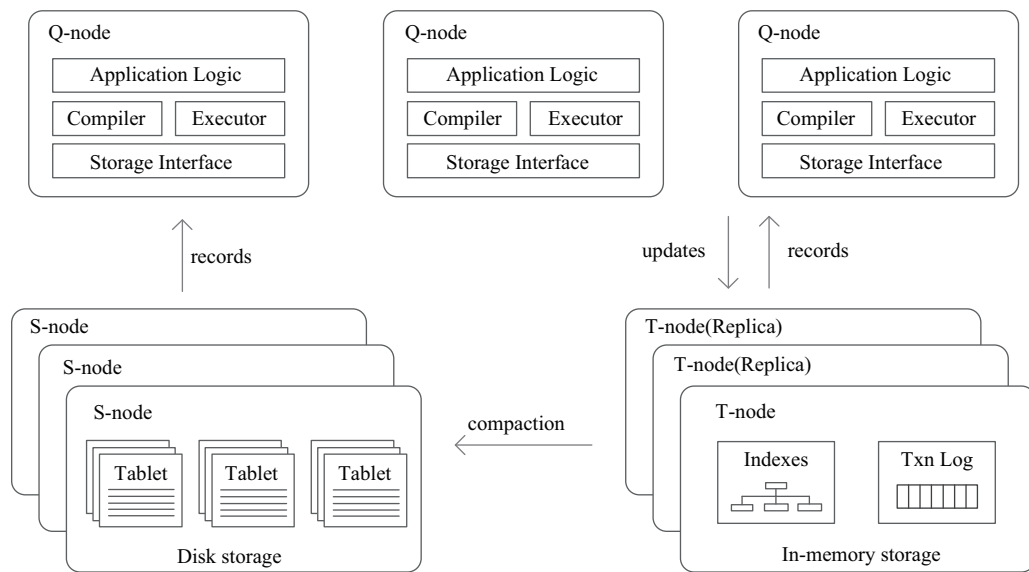


图1 CEDAR 架构

Fig. 1 Architecture of CEDAR

1.2 数据库系统中的主键维护

数据库系统中的主键维护包括了生成和更新主键. 其中, 自动递增的数值类型占用存储空间小, 处理效率高, 是数据库系统生成主键的首选. 主流的关系数据库系统基本都支持生成自增主键的功能, 主要分为直接和间接两种方式. 其中, MySQL^[7]和 Microsoft SQL Server 等数据库系统采用直接的方式, 即在定义表结构的时候标记自增主键列, 在插入新数据行的时候直接生成自动递增的数值型主键值; 而 PostgreSQL、DB2 和 Oracle 等数据库系统采用间接的方式, 即使用数据库提供的其他功能 (如 Sequence) 生成递增的数值型数据, 从而间接地生成自增主键. VoltDB^[8]和 OceanBase^[9]等 NewSQL 数据库目前还没有支持生成自增数值型主键的功能. 同样对于更新主键来说, 传统的关系数据库基本都支持更新主键, 而 NewSQL 数据库中只是部分支持.

在分布式数据库系统中生成或更新主键值的过程本质上就是在其中执行一例长事务,

所以我们需要采用适当的并发控制协议去保证事务的 ACID 特性以及冲突可串行化^[10-11]. 当前比较成熟的并发控制协议有基于锁的悲观并发控制协议和基于有效性验证的乐观并发控制协议^[12-13]. 基于锁的悲观并发控制协议是传统数据库系统中应用最为广泛的一类并发控制技术, 其基本思想是事务在对数据项操作之前必须先申请该数据项的锁. 在基于锁的并发控制协议中, 通常采用两阶段封锁协议 (Two-Phase Locking Protocol, 2PL)^[14]来保证事务的可串行化调度. 基于有效性验证的乐观并发控制协议的基本思想不是阻碍事务的执行, 而是在事务提交时进行冲突验证, 若没有冲突则提交事务, 否则重启事务. 由于事务重启的代价较大, 因此乐观并发控制适用于写冲突较小的场景.

2 先读后写并发控制模型

在本节中, 首先描述了在日志结构的分布式数据库系统中一种常见的先读后写更新操作, 然后针对该类操作提出其实现并发控制的模型方法, 作为实现更新主键功能的理论基础. 最后, 提出了基于缓存的优化方案来提升更新操作的性能.

2.1 先读后写更新操作

在基于日志结构的分布式架构下, 存在一个特殊的更新场景: 在更新操作前首先需要查询数据库, 以确定待更新的数据行, 最后再执行正常的更新流程来修改数据. 该类更新操作包含一个显式的查询数据库过程, 本文称之为显式的先读后写 (WAR) 更新, 简称为先读后写更新. WAR 操作本质上是数据库系统中的一类长事务, 它包含了读取和写入两类操作. 为了便于描述, 我们给出有关 WAR 操作的一些概念.

- 基准值 (Baseline value, v_B). 基准值 v_B 为数据库系统中记录的某个已经存在的值, 由用户自定义查询条件, 查询基线数据和增量数据合并后获得. 基准值一般记录了数据库表格或数据行的某种状态, 基准值不限为单个值, 即有可能为值的集合.

- 目标值 (Target value, v_T). 目标值 v_T 通常与基准值具有相同的含义, 查询节点在本地将基准值通过目标表达式函数转换成目标值. 在 WAR 操作的最后往往需要将目标值更新到增量数据中, 成为新的基准值.

- 目标行 (Target row, r_T). 目标行 r_T 是目标值 v_T 以自定义的关系关联的数据行, 目标行是由目标基线数据和目标增量数据组成的. WAR 操作的最终目标是修改目标行, 因此需要通过目标值访问目标行. 在数据库系统中目标值和目标行一般通过主键关系关联, 即目标值是目标行的主键.

在基于日志结构的分布式数据库系统中, 执行 WAR 操作需要查询节点、基线节点和增量节点相互协作, WAR 操作的具体执行流程如图 2 所示. 客户端的更新请求以 (a)、(b) 和 (c) 的顺序执行, 该流程可分为写准备阶段和写阶段两个阶段.

- 1) 写准备阶段. 在写准备阶段, 查询节点查询基准值, 然后根据基准值计算目标值, 最后根据目标值查询目标基线数据, 为写阶段做数据准备, 分别对应流程中的 (a) 和 (b) 两个步骤.

- 2) 写阶段. 在写阶段, 增量节点已经获得了目标值和目标行的基线数据, 进而更新目标数据行, 同时可能也会更新基准值, 对应流程中 (c) 步骤.

在写阶段如果需要更新基准值, 则最简单的 WAR 操作至少包含了一个读操作和两个写操作, 其中的读操作和一个写操作是针对基准值的, 另一个写操作则是针对目标行. 为了保证 WAR 操作在数据库系统中并发执行时事务的 ACID 特性以及实现冲突的可串行化调度,

我们需要增加并发控制机制.

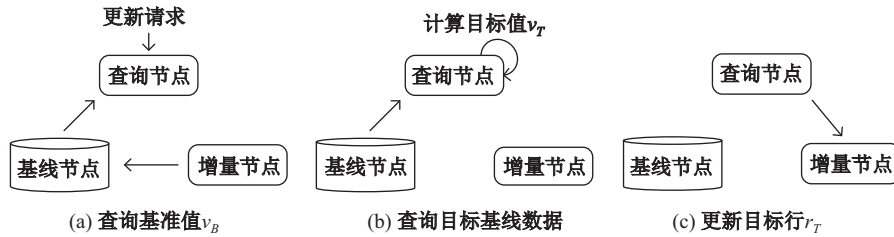


图 2 WAR 操作的执行流程

Fig. 2 Execution process of the WAR operation

2.2 PCC-WAR 模型

PCC-WAR 模型是应用悲观并发控制协议的 WAR 操作的抽象描述, 它基于封锁机制来保证 WAR 操作的事务性. WAR 操作在每次读取基准值的增量数据前锁定该值, 在事务提交后再释放该锁, 保证在此期间基准值不会被其他事务读取或修改.

(1) 锁类型选择

PCC-WAR 模型中的封锁机制针对的是基准值, 而基准值一般是某张表格单个或多个数据行的数据, 因此, 就锁的粒度而言, 基准值上的锁属于行级锁. 就锁的类型而言, 基准值上的锁有共享锁 (S 锁) 和排他锁 (X 锁) 两种选择, 其选择的原则如下.

- 排他锁. 若 WAR 操作在写阶段需要更新基准值, 则选择排他锁. 试分析: 若 WAR 操作在写准备阶段只申请共享锁, 存在两个 WAR 操作同时更新相同目标行, 这两个操作都执行到了写阶段, 因为双方都持有基准值的共享锁, 所以这两个事务都不能申请到基准值的排他锁, 进入死锁状态. 因此, 为避免这种现象, 应该在读取基准值时申请排他锁.

- 共享锁. 除上述情况外, 可放宽锁的权限为共享锁, 只禁止其他事务的写请求.

(2) 正确性保证

PCC-WAR 模型使用封锁协议实现事务的并发控制, 在读取基准值前进行加锁操作, 在事务提交后才释放锁, 这和目标行更新操作的原有加锁和释放锁一起, 组成了一个两阶段封锁协议. 一个只查询一个基准值并且只更新一个目标行的 WAR 操作在执行过程中的两阶段封锁机制如图 3 所示.

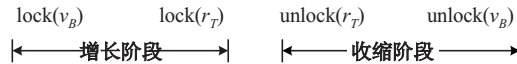


图 3 PCC-WAR 模型的两阶段封锁协议

Fig. 3 Two-phase locking protocol of the PCC-WAR model

在图 3 中, WAR 操作包含对基准值 v_B 和目标行 r_T 的加锁和释放锁操作, 其中 lock() 表示加锁, 而 unlock() 表示释放锁. 两阶段封锁协议被证明可以保证事务调度是冲突串行化的, 因此 PCC-WAR 模型可以保证 WAR 操作的事务性.

2.3 OCC-WAR 模型

OCC-WAR 模型是应用乐观并发控制技术的 WAR 操作的抽象描述, 它基于有效性验证协议来保证 WAR 更新操作的事务性. 查询节点无需申请锁而直接读取基准值, 在增量节点提交事务前进行有效性验证. 若验证通过, 则提交当前事务; 否则当前事务回滚, 由查询节点发起重试操作.

(1) 有效性的验证方法

有效性验证的目的是检查当前事务的读写操作是否与其他正在执行的事务存在冲突, 在WAR操作中即是检查基准值是否被其他事务读取或修改过, 验证在写准备阶段读取到的基准值副本在事务提交前是否依然有效.

在日志结构的分布式数据库系统中, 增量数据一般以追加的形式更新, 较容易实现对数据多版本的支持. 因此, 我们用查询节点记录基准值的状态 (版本号或最后一次被读写的时间戳等) 来对 OCC-WAR 模型进行有效性验证, 具体验证方法如图 4 所示. 查询节点在读基准值 v_B 时, 同时获得该值的一个状态值 v_S , 查询节点再次将该值和目标基线数据一起发送给增量数据节点, 增量节点在事务提交前对比该状态值是否与基准值当前的状态值一致, 如果一致则验证通过, 否则该事务回滚.

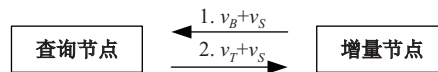


图4 OCC-WAR 模型有效性验证方法

Fig. 4 Method for OCC-WAR model validation

(2) 正确性保证

基于有效性验证的乐观并发控制协议要求在做有效性测试时比较该事务与其他事务的读写集合有没有冲突, OCC-WAR 模型通过比较时间戳 (或版本号) 的方式进行有效性验证. 在日志结构的分布式数据库系统中, 时间戳值的对比和事务的提交可以看作是一个同时执行的瞬时行为. 如图 5 所示, 假设存在 a、b、c、d 四个事务对同一个基准值进行读取, R 表示读基准值的操作 (Read)、V 表示有效性验证 (Validate)、C 表示提交操作 (Commit).

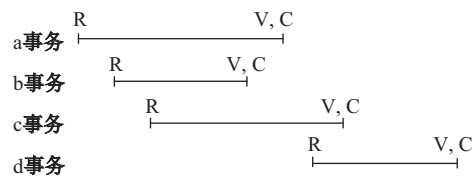


图5 OCC-WAR 模型有效性验证举例

Fig. 5 An example of OCC-WAR model validation

在图 5 中, b、c 事务和 a 事务有交叉, 而 d 事务对 a 事务没有影响, 其中 b 事务提交 (或 c 事务读取基准值) 会引起基准值时间戳的变化, 会导致 a 事务的有效性验证失败. 若时间戳没有变化, 说明没有其他同时执行的事务与当前事务发生写写冲突 (或读写冲突), 则当前事务可提交. 只要存在任何其他事务在当前事务读取基准值到有效性验证前的这段时间里对基准值进行了读取或修改, 都会导致有效性验证失败, 则当前事务重启. 因此, OCC-WAR 模型可以保证事务的执行是可串行化的.

2.4 基于缓存的优化

WAR 操作的写准备阶段, 有一个查询节点向基线节点和增量节点读取基准值的过程, 用以计算目标值, 进而查询出目标行的基线数据. 查询基准值过程需要查询节点、基线节点和增量节点相互协作, 这增加了事务并发控制的复杂性, 且耗时较长. 如果我们想要简化 WAR 操作流程并且优化其性能, 就需要尽可能地避免对磁盘基线数据的访问. 因此, 当满足以下两个条件时, 我们可以将所有基准值缓存在增量节点的内存中, 从而避免对磁盘基线数据的访问.

1) 更新目标行的新值不依赖于旧值. 满足这一条件说明更新目标行时不需要使用其基线数据, 可直接将更新表达式的值追加到增量数据中.

2) 基准值的数据量较小. 满足这一条件保证了增量节点的内存可足够存储下所有的基准值, 而且可以减小基准值缓存的加载和维护代价.

2.4.1 基本流程

基于缓存优化后的 WAR 操作的基本流程如图 6(a) 所示. 查询节点接收到 WAR 请求后, 直接转发给增量节点, 增量节点根据缓存的基准值计算出目标值, 进而更新目标行. 对比原始的 WAR 操作流程 (见 2.1 节) 可知, 优化后的流程不再有基线节点的参与, 查询节点也只是承担请求转发的功能, WAR 操作简化为单节点执行的事务.

2.4.2 缓存的加载和维护

图 6(b) 描述了缓存的加载过程. 若在初始情况下, 增量节点的内存中没有基准值的缓存值, 因此, 在 WAR 请求到达增量节点后未能命中基准值的缓存, 进而执行缓存的加载. 增量节点返回查询节点“基准值未缓存”的错误, 由查询节点向基线数据发起查询基准值的请求, 基线节点融合基线数据和增量数据后返回查询节点, 查询节点将 WAR 请求和基准值一起发送至增量节点执行. 在本次 WAR 操作结束后, 该基准值的缓存保存在增量节点的内存中, 以供后续的 WAR 请求使用.

在基准值缓存到增量节点后, 需要根据数据库某些状态的变化更新基准值, 这种状态的变化可能是 WAR 操作引起, 也可能由其他普通的更新操作引起, 因此, 需要在数据发生变化时及时更新基准值的缓存.

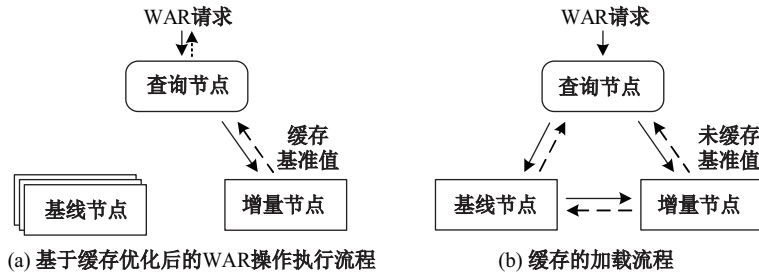


图 6 基于缓存优化的 WAR 操作执行流程和缓存加载流程

Fig. 6 Process of cache-based optimization and the cache loading operation

3 主键维护的实现

在本节中, 我们针对日志结构分布式数据库系统的特点, 应用本文提出的 PCC-WAR 和 OCC-WAR 并发控制模型设计多种算法实现生成和更新主键功能.

3.1 主键的生成

主键的生成是指在插入新行数据时, 数据库系统自动产生一个唯一的自增数值型主键, 该值是在当前最大主键值的基础上通过自增操作得到的新主键值. 因此, 我们可以利用数据库中的系统表去记录每个需要生成全局自增主键的用户表的当前最大主键值.

生成自增数值型主键的是一类典型的 WAR 操作, 用户表当前最大主键值对应 WAR 的基准值 v_B , 新主键值对应目标值 v_T , 新行对应目标行 r_T , 目标值和目标行通过主键关系关联, 目标表达式函数即是最大主键自增的操作, 而用户自定义的查询条件指根据查询表名映射其对应的系统表.

生成主键操作需要读写同一个当前最大的主键值, 故会产生较大冲突, 并不适合 OCC-WAR 模型. 因此, 本节只应用 PCC-WAR 模型的实现生成自增主键的并发控制, 该方法保证全局连续有序自增, 在数据插入成功后才会更新记录的最大主键值.

3.1.1 PCC-WAR 模型主键生成算法

PCC-WAR 模型生成主键的方法是在读取系统表中记录的最大主键值前, 将系统表中最大主键值所在数据行加锁 (因为生成主键操作最终要更新系统中的最大主键值, 故选择排他锁), 待事务提交后释放锁. 在插入新行前, 根据新主键值查询基线数据和增量数据判断新主键是否存在, 若存在则自增最大主键值并发起重试, 否则完成插入操作并更新最大主键值. 其具体流程如算法 1 所述.

算法 1 PCC-WAR 模型主键生成算法

输入: 不包含主键值的新行数据 new_row

输出: 含有主键值的新行数据 new_full_row

```

1: Exclusive_lock( $max\_rowkey$ );
2: 读取  $max\_rowkey$ ;
3: 计算主键值  $rowkey \leftarrow max\_rowkey + 1$ ;
4: 根据  $rowkey$  查询基线数据  $d_B$ ;
5: 根据  $rowkey$  查询增量数据  $d_I$ ;
6: if ( $d_B == NULL \ \&\& \ d_I == NULL$ ) then
7:    $new\_full\_row \leftarrow rowkey + new\_row$ ;
8:   插入含有新主键值的数据  $new\_full\_row$ ;
9:   更新最大主键值  $max\_rowkey \leftarrow rowkey$ ;
10:  同步操作日志并刷盘;
11:  Commit();
12:  Unlock( $max\_rowkey$ );
13: else
14:  更新最大主键值  $max\_rowkey \leftarrow rowkey + 1$ ;
15:  Retry();
16: end if

```

3.1.2 PCC-WAR 模型主键生成缓存优化算法

生成自增数值主键操作中基准值是系统表中的最大主键值. 首先, 这是一个数据量很小的基准值; 其次, 插入新行操作读取基准数据的目的是用来验证主键是否重复, 而自增主键可以保证主键值不会重复, 因而该读取基准数据的操作是非必要的. 以上两点满足了 WAR 操作缓存优化的两个基本条件, 因此可以使用缓存来优化生成主键的操作. 在增量节点缓存了最大主键值的情况下, 缓存优化后 PCC-WAR 模型生成自增主键的具体流程如算法 2 所述.

算法 2 PCC-WAR 模型主键生成缓存优化算法

输入: 不包含主键值的新行数据 new_row

输出: 含有主键值的新行数据 new_full_row

```

1: Exclusive_lock( $max\_rowkey$ );
2: 读取  $max\_rowkey$ ;
3: 计算主键值  $rowkey \leftarrow max\_rowkey + 1$ ;
4:  $new\_full\_row \leftarrow rowkey + new\_row$ ;
5: 插入含有新主键值的数据  $new\_full\_row$ ;
6: 更新最大主键值  $max\_rowkey \leftarrow rowkey$ ;
7: 同步操作日志并刷盘;
8: Commit();
9: Unlock( $max\_rowkey$ );

```


对比PCC-WAR模型主键生成算法, PCC-WAR模型主键生成缓存优化算法极大提高了数据库系统生成主键的效率. 从时间复杂度上来看, PCC-WAR模型主键生成算法需要查询基线和增量数据来确定包含新主键值的数据是否有效, 其中查询基线数据造成的磁盘I/O极大地限制了生成主键的事务吞吐量, 而缓存优化后的算法避免了查询基线数据的过程, 从而消除了磁盘I/O的瓶颈; 从空间复杂度上来看, 缓存优化后的算法在增量节点的内存中增加了相关系统表去维护自增数据表的当前最大主键值, 由于一个自增数据表只会有一个当前最大自增值, 所以空间消耗也较少.

3.2 主键的更新

在日志结构的数据库系统中更新主键值本质是在增量数据中插入新数据行的同时删除原数据行. 其中一个必要步骤是根据需要在原行的基础上计算新主键值, 并将新主键值和原行非主键列组成新数据行.

更新主键也是一类典型的WAR操作, 原行数据对应WAR操作中的基准值 v_B , 新主键值对应目标值 v_T , 新行数据对应目标行 r_T , 目标值和目标行通过主键关系关联, UPDATE语句中的SET子句和WHERE子句分别对应目标表达式函数和用户自定义的查询条件.

3.2.1 PCC-WAR模型更新主键算法

PCC-WAR模型更新主键的方法是在查询原行数据前将原数据行加锁(因为更新主键操作最终需要删除原数据行, 即修改基准值, 故选择排他锁), 待更新主键事务提交后再释放锁. 根据新行主键值表达式函数计算获得新主键值, 进而利用新主键值和原行主键值分别查询新行和原行的基线和增量数据. 通过融合得到新行的基线和增量数据来判断新主键值是否存在, 若存在则返回“主键重复”错误, 否则插入新数据行并删除原数据行. 其具体流程如算法3所述.

算法3 PCC-WAR模型更新主键算法

输入: 原行主键值 $K_{original}$, 新主键值表达式 F

输出: 新行数据 new_row

```

1: Exclusive_lock( $original\_row$ );
2: 读取原行数据 $original\_row$ ;
3: 计算新主键值 $K_{new} \leftarrow F(original\_row)$ ;
4: 根据 $K_{new}$ 和 $K_{original}$ 分别查询新行基线数据 $d_{NB}$ 和原行基线数据 $d_{OB}$ ;
5: 根据 $K_{new}$ 和 $K_{original}$ 分别查询新行增量数据 $d_{NI}$ 和原行增量数据 $d_{OI}$ ;
6: if ( $d_{NB} == NULL \ \&\& \ d_{NI} == NULL$ ) then
7:   插入新行 $new\_row \leftarrow K_{new} + d_{OB} \cup d_{OI}$ ;
8:   标记删除原行 $original\_row$ ;
9:   同步操作日志并刷盘;
10:  Commit();
11:  Unlock( $original\_row$ );
12: else
13:   Abort() and Unlock( $original\_row$ );
14: end if

```

PCC-WAR模型更新主键算法是基于锁的悲观并发控制协议, 所以此算法的时间复杂度与数据库系统的锁维护代价相关.

3.2.2 OCC-WAR模型更新主键算法

在OCC-WAR模型更新主键的方法中, 查询节点在读取原行的增量数据时同时获取该增量数据最近一次修改的时间戳. 增量节点除了要正常地执行插入新行数据和删除原行数据操作, 还要在事务提交前验证由查询节点发来的时间戳与当前的原行增量数据的修改时间戳是否一致,

若一致则提交事务, 否则事务回滚并发起重试. 其具体流程如算法 4 所述.

算法 4 OCC-WAR 模型更新主键算法

输入: 原行主键值 $K_{original}$, 新主键值表达式 F

输出: 新行数据 new_row

```

1: 读取原行数据  $original\_row$ ;
2: 计算新主键值  $K_{new} \leftarrow F(original\_row)$ ;
3: 根据  $K_{new}$  和  $K_{original}$  分别查询新行基线数据  $d_{NB}$  和原行基线数据  $d_{OB}$ ;
4: 根据  $K_{new}$  和  $K_{original}$  分别查询新行增量数据  $d_{NI}$  和原行增量数据  $d_{OI}$ ;
5: if ( $d_{NB} == \text{NULL} \ \&\& \ d_{NI} == \text{NULL}$ ) then
6:   插入新行  $new\_row \leftarrow K_{new} + d_{OB} \cup d_{OI}$ ;
7:   标记删除原行  $original\_row$ ;
8:   if( $\text{Check\_validation}() == \text{true}$ ) then
9:     同步操作日志并刷盘;
10:     $\text{Commit}()$ ;
11:   else
12:     $\text{Rollback}()$  and  $\text{Retry}()$ ;
13:   end if
14: else
15:    $\text{Abort}()$ ;
16: end if

```

OCC-WAR 模型更新主键算法是基于有效性验证的并发控制协议, 此算法在低冲突场景中一般具有良好的性能, 而在高冲突场景下, 由于大量的有效性验证失败导致事务重做从而会限制更新主键的事务吞吐量.

4 实验评估

我们在分布式日志结构的数据库系统 CEDAR 中, 应用本文提出的 PCC-WAR 和 OCC-WAR 并发控制模型方法实现了主键维护功能. 在本节中, 我们评估了各种实现方法的性能, 证明了所提方法的高效性.

4.1 实验环境

我们将 CEDAR 数据库部署在主备双集群环境中, 每个集群中都有 2 台 T-node (增量节点, 主备部署), 3 台 Q-node (查询节点), 3 台 S-node (基线节点). 集群中的每台机器都有两个 Intel(R)Xeon(R)E5606@2.13GHz 型号的 4 核心 CPU 以及 100 GB 内存和 100 GB 的 SSD 盘. 同时, 本文使用基准测试工具 Sysbench^[15]测试数据库事务处理性能.

4.2 实验方法与结果

4.2.1 生成主键的性能

我们通过客户端模拟 (通过数据库事务功能封装一组 SQL 操作模拟生成主键)、PCC-WAR 模型和缓存优化的 PCC-WAR 模型三种方法实现了生成主键的功能, 并且对比了各自方法的性能, 实验结果如图 7 所示.

由实验结果可知, 相对于客户端模拟生成主键的性能, 基本的 PCC-WAR 模型方法对生成主键性能提升不大, 这是因为在高冲突的情况下查询最大主键值和磁盘基线数据耗时太长, 加剧了事务的冲突, 导致过多的操作执行超时. 而经过缓存优化的 PCC-WAR 模型方法极大提升了生成主键的性能, 这是因为增加缓存后避免了对磁盘基线数据的访问, 缓解了事务的冲突. 但是在并发执行生成主键操作时, 多个线程不可避免地竞争持有系统表中的最大主键值的锁, 所以不

管何种方法都将很快达到了吞吐量的上限, 后续随着线程数增加不但不会提升吞吐量, 反而会加剧事务的冲突, 延长响应时间。

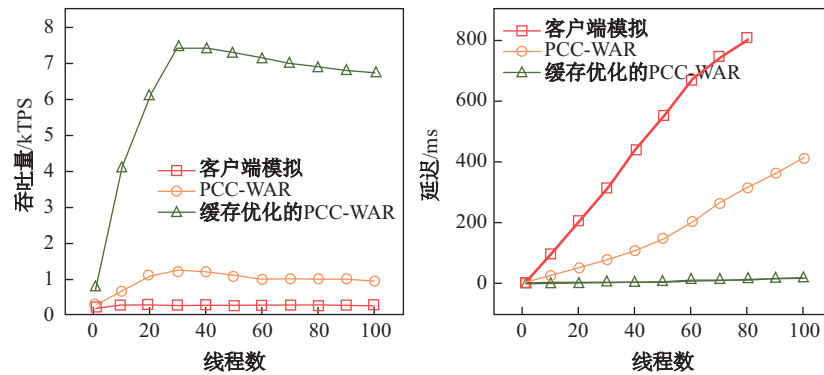


图7 生成主键的性能

Fig. 7 Performance of generating the primary key

4.2.2 更新主键的性能

由于PCC-WAR模型和OCC-WAR模型使用不同的并发处理机制去实现更新主键的功能, 而且在不同的读写冲突情况下的性能表现也不同, 因此分别模拟低冲突和高冲突两种场景, 测试不同方法更新主键的性能。

(1) 低冲突场景

我们控制每个线程更新不同的数据行, 这样就可以模拟一种极端的“低冲突”场景, 即所有事务都不存在冲突。作为对比我们还测试了更新普通列和客户端模拟更新主键的性能。

由图8可知, 在低冲突的实验环境中, PCC-WAR和OCC-WAR两种模型对更新主键性能的效果大致相当, OCC-WAR模型的性能略高于PCC-WAR模型。通过分析可知, 由于CEDAR增量节点相当于一个内存数据库, 每个数据行都有一个行级锁, 所以锁管理代价非常低, 因此相较于OCC-WAR模型更新主键方法, 基于封锁协议的PCC-WAR模型更新主键方法也能表现出较好的性能。同时, 本文实现的更新主键性能相较于更新普通列和客户端模拟更新主键的性能也在合理的范围内。

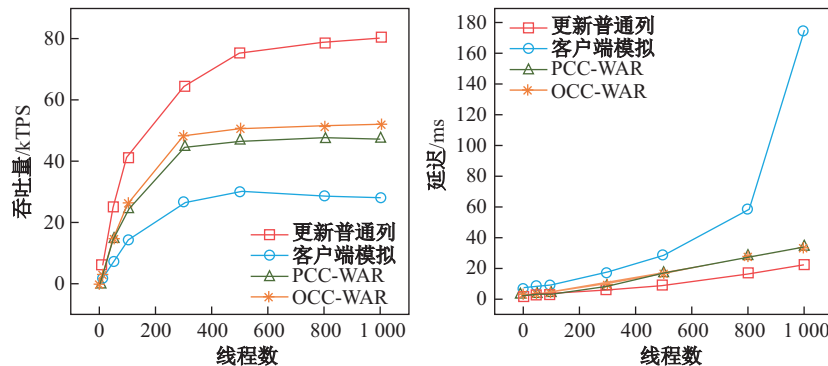


图8 低冲突下更新主键的性能

Fig. 8 Performance of updating the primary key under low-conflict scenarios

(2) 高冲突场景

在不同的主键范围内随机更新数据行, 可以模拟不同程度的冲突场景, 我们采用“1:1”的比

例(即 N 个线程同时随机更新 N 行数据)测试 PCC-WAR 模型和 OCC-WAR 模型的性能, 这是一种冲突较高的场景. 实验结果如图 9 所示.

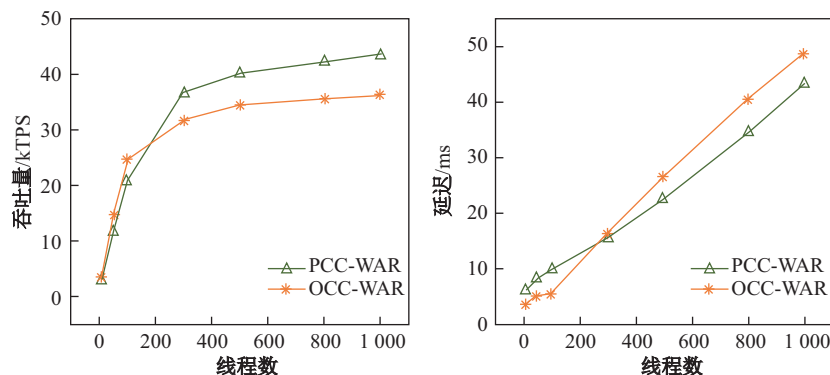


图9 高冲突下更新主键的性能

Fig.9 Performance of updating the primary key under high-conflict scenarios

由实验结果可知, 存在冲突的情况下 PCC-WAR 和 OCC-WAR 模型更新主键的性能均有所下降. 在线程数较少的情况下, OCC-WAR 模型方法的性能高于 PCC-WAR 模型方法. 随着线程数的增加, OCC-WAR 模型更新主键事务重启开始增多, 因此 PCC-WAR 更新主键的性能优于 OCC-WAR 更新主键.

5 总 结

本文以分布式日志结构数据库系统中的主键维护问题为出发点, 首先总结了一类具有普遍意义的先读后写更新操作, 进而分别介绍了 PCC-WAR 和 OCC-WAR 并发控制模型, 最后应用这两类模型设计并实现了高效的主键维护功能, 又通过实验验证了本文的方法是可行且高效的.

[参 考 文 献]

- [1] ROSENBLUM M, OUSTERHOUT J K. The design and implementation of a log-structured file system [J]. ACM Transaction on Computer Systems, 1992, 10(1): 26-52.
- [2] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data [C]//Symposium on Operating Systems Design and Implementation. USENIX Association, 2006: 205-218.
- [3] Hbase. [EB/OL]. [2018-07-02]. <http://hbase.apache.org/>.
- [4] Cassandra. [EB/OL]. [2018-07-02]. <http://cassandra.apache.org/>.
- [5] CDEAR. [EB/OL]. [2018-07-02]. <https://github.com/daseECNU/Cedar/>.
- [6] O'NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree (LSM-tree) [J]. Acta Informatica, 1996, 33(4): 351-385.
- [7] Mysql Cluster. [EB/OL]. [2018-07-02]. <https://www.mysql.com/products/cluster/>.
- [8] STONEBRAKER M, WEISBERG A. The voltdb main memory DBMS [J]. IEEE Data Eng Bull, 2013, 36(2): 21-27.
- [9] OceanBase. [EB/OL]. [2018-07-02]. <https://github.com/alibaba/oceanbase/>.
- [10] GARCIA-MOLINA H, ULLMAN J D, WIDOM J D. Database System Implementation [M]. London: Prentice Hall, 1999: 576-601.
- [11] The Open Group. Distributed TP: The XA Specification [EB/OL]. [2018-07-02]. <https://publications.opengroup.org/c193>.
- [12] KUNG H T. On optimistic methods for concurrency control [C]// International Conference on Very Large Data Bases. IEEE, 1981: 351-351.
- [13] CHOI H J, JEONG B S. A timestamp-based optimistic concurrency control for handling mobile transactions [C]// International Conference on Computational Science and Its Applications. Berlin: Springer-Verlag, 2006: 796-805.

(下转第 119 页)

- [20] YU X. An evaluation of concurrency control with one thousand cores [D]. Boston: Massachusetts Institute of Technology, 2015.
- [21] PANDIS I, JOHNSON R, HARDAVELLAS N, et al. Data-oriented transaction execution [J]. Proceedings of the Vldb Endowment, 2010, 3(1/2): 928-939.
- [22] THOMSON A, THOMSON A, ABADI D J. An evaluation of the advantages and disadvantages of deterministic database systems [J]. Proceedings of the Vldb Endowment, 2014, 7(10): 821-832.
- [23] PAVLO A, CURINO C, ZDONIK S B, et al. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems [C]//International conference on management of data, 2012: 61-72.
- [24] GOTTEMUKKALA V, LEHMAN T J. Locking and latching in a memory-resident database system [C]// Intl Conf on Very Large Databases, 1992: 533-544.
- [25] HELLAND P, SAMMER H, LYON J, et al. Group Commit Timers and High Volume Transaction Systems [C]//High performance transaction systems workshop, 1987: 301-329.
- [26] JOHNSON R, PANDIS I, STOICA R, et al. Aether: a scalable approach to logging [J]. Proceedings of the Vldb Endowment, 2010, 3(1/2): 681-692.
- [27] ALPERN D, ARORA G, BARCLAY C, et al. Oracle Database Advanced Application Developer's Guide, 11g Release 2 (11.2) E17125-06[R/OL]. [2018-07-10]. https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/toc.htm.
- [28] SOISALON-SOININEN E, YLÖNEN T. Partial strictness in two-phase locking [C]// International Conference on Database Theory. Springer-Verlag, 1995: 139-147.
- [29] MALVIYA N, WEISBERG A, MADDEN S, et al. Rethinking main memory OLTP recovery [C]// International Conference on Data Engineering. IEEE, 2014: 604-615.
- [30] STOICA R, LEVANDOSKI J J, LARSON P A. Identifying hot and cold data in main-memory databases [C]//International Conference on Data Engineering. IEEE, 2013: 26-37.
- [31] ELDAWY A, LEVANDOSKI J, LARSON P Å. Trekking through Siberia: Managing cold data in a memory-optimized database [J]. Proceedings of the Vldb Endowment, 2014, 7(11): 931-942.
- [32] DEBRABANT J, PAVLO A, TU S, et al. Anti-caching: A new approach to database management system architecture [J]. Proceedings of the Vldb Endowment, 2013, 6(14): 1942-1953.
- [33] THOMSON A, DIAMOND T, WENG S C, et al. Calvin: Fast distributed transactions for partitioned database systems [C]//ACM SIGMOD International Conference on Management of Data. ACM, 2012: 1-12.

(责任编辑: 李万会)

(上接第 90 页)

- [14] ESWARAN K P, GRAY J N, LORIE R A, et al. The notions of consistency and predicate locks in a database system [J]. Readings in Artificial Intelligence & Databases, 1989, 19(11): 523-532.
- [15] Sysbench. [EB/OL]. [2018-07-02]. <https://dev.mysql.com/downloads/benchmarks.html>.

(责任编辑: 林 磊)