2017 届硕士专业学位研究生学位论文（全日制研究生）

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51141500077

華東師範大學

East China Normal University

硕士专业学位论文

MASTRER'S DEGREE THESIS (PROFESSIONAL)

论文题目：可扩展数据库管理系统的主键维护

院 系 名 称：	计算机科学与软件工程学院
专业学位类别：	工程硕士
专业学位领域：	软件工程
指 导 教 师：	周敏奇 副教授
学 位 申 请 人：	刘柏众

2016 年 10 月 10 日

2017 届硕士专业学位研究生学位论文（全日制研究生）

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51141500077

華東師範大學

East China Normal University

硕士专业学位论文

MASTRER'S DEGREE THESIS (PROFESSIONAL)

论文题目：可扩展数据库管理系统的主键维护

院 系 名 称：	计算机科学与软件工程学院
专业学位类别：	工程硕士
专业学位领域：	软件工程
指 导 教 师：	周敏奇 副教授
学 位 申 请 人：	刘柏众

2016 年 10 月 10 日

Dissertation for professional master's degree in 2017

University Code: 10269

Student ID: 51141500077

East China Normal University

Title: **THE MANAGEMENT OF PRIMARY KEY IN SCALABLE DBMS**

Department:	**School of Computer Science and Software Engineering**
Professional degree category:	**Master of Engineering**
Professional degree field:	**Software Engineering**
Supervisor:	**Associate Prof. ZHOU Miqi**
Candidate:	**LIU Bozhong**

October, 2016

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《可扩展数据库管理系统的主键维护》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名：_____ **日期：** 年 月 日

华东师范大学学位论文著作权使用声明

《可扩展数据库管理系统的主键维护》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的研究成果归华东师范大学所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和相关机构如国家图书馆、中信所和“知网”送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（ ）1.经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文*，于 年 月 日解密，解密后适用上述授权。

（ ）2.不保密，适用上述授权。

导师签名_____ 本人签名_____

年 月 日

* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《可扩展数据库管理系统的主键维护》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名：_____ **日期：** 年 月 日

华东师范大学学位论文著作权使用声明

《可扩展数据库管理系统的主键维护》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的研究成果归华东师范大学所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和相关机构如国家图书馆、中信所和“知网”送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（ ）1.经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文*，于 年 月 日解密，解密后适用上述授权。

（ ）2.不保密，适用上述授权。

导师签名_____ 本人签名_____

年 月 日

* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

刘柏众 硕士专业学位论文答辩委员会成员名单

姓名	职称	单位	备注
钱卫宁	教授	华东师范大学	主席
翁楚良	教授	华东师范大学	
张蓉	副教授	华东师范大学	
蔡鹏	副教授	华东师范大学	
沙朝锋	副教授	复旦大学	

摘　要

近年来，社交网络、移动互联网和电子商务等应用的迅猛发展，对数据库管理系统的数据存储规模和数据处理可扩展性均提出了极高的要求，传统数据库管理系统逐渐无法胜任当前应用的发展需求。然后，基于读写分离的高可扩展数据库管理系统，例如，Bigtable，Cassandra 等，在数据存储规模、处理性能和高可扩展性三方面实现了很好的权衡。阿里巴巴的开源数据库管理系统，OceanBase 是这类系统的又一个典型实例。目前，OceanBase 依旧处于不断发展、不断完善的过程中，该系统对部分功能的支持并不完善，特别是对表主键的管理，依旧无法满足关键性应用的需求。在数据库管理系统中，表主键管理包括表主键的生成、更新和删除。本文重点研究 OceanBase 系统中的表主键管理，设计并实现了一整套适用于该系统的表主键管理策略和方法。

本文立足于高可扩展数据库管理系统，特别是在唯一性约束、事务隔离级别和高处理性能等方面对表主键管理提出的挑战，设计并实现了一整套先读后写操作的数据处理模型，并且将其应用于表主键管理中。所提表主键管理模型在 OceanBase 系统中进行了充分实验，均获得了较好的处理效果。本文工作的主要贡献如下：

1) 提出了一套先读后写（WAR）数据处理的策略，并分别基于悲观并发控制协议和乐观并发控制协议扩展出两个 WAR 模型：PCC-WAR 模型和 OCC-WAR 模型，同时提出了相关的基于缓存的优化方案，使表主键管理性能得到了较高的提升。

2) 提出了在基于读写分离的高可扩展数据库管理系统中实现主键维护功能的方法，分别是：基于 PCC-WAR 生成主键、基于缓存优化生成主键、基于 PCC-WAR 更新主键、基于 OCC-WAR 更新主键等优化策略。

3) 在 OceanBase 中实现了表主键管理的功能，并分别从功能和性能两个方面进行了充分实验，实验结果表明所提策略和方法均能够满足事务隔离要求和主

键唯一性约束条件。与外部间接实现主键管理的方法相比，其性能有显著提升。

本文提出的表主键管理功能的实现方法在可扩展数据库管理系统中得到了成功应用，并获得了较高的性能。同时，本文提出了先读后写操作的抽象模型，同样可扩展并应用于其它数据库管理系统中。

关键词：可扩展数据库管理系统，读写分离，并发控制，主键生成，主键更新

Abstract

Recent years, with the rapid development of social networking, mobile Internet and e-commerce, the demands for storage capacity and processing scalability of DBMS are becoming increasingly higher, and consequently result in traditional DBMS facing more and more challenges and failing to satisfy all the requirements from its applications. The newer DBMSes based on read/write separation strategy make a better trade-off among performance, storage capacity and processing scalability. The DBMS of OceanBase, open-sourced by Alibaba, is a typical instance of them. Nevertheless, Oceanbase is new and unmatured in terms of a set of unsupported functionalities. The management of primary key is one of these functionalities, which consists of key generation, modification, and deletion in the database systems. In this paper, we focus on the primary key management levering on Oceanbase, and propose a set of strategies for primary key management which takes advantages of Oceanbase.

Based on the challenges of primary key management in the scalable DBMSes, in terms of the constrains of uniqueness, transactional isolation levels, and performance requirements, we proposed a new processing model, called operation of writing after reading, which suites for primary key management. Extensive experimental studies are conducted, which confirms of the effectiveness and efficiency of strategies. The main contributions of this paper are as follows:

1) We proposed a processing model, called Writing After Reading (WAR) operation, with two extensions, say PCC-WAR model and OCC-WAR model, which are based on the Pessimistic Concurrency Control protocol and Optimistic Concurrency Control protocol respectively, as well as a cache-based optimization scheme was proposed for specific scenarios.

2) We optimized the primary key management in the scalable DBMS based on

read/write separation architecture, say OCC-WAR-based generation, cache-base optimization on PCC-WAR generation, PCC-WAR-based modification, OCC-WAR-based modification and etc., for higher performance.

3) We implemented the primary key management in OceanBase, and conducted a set of experimental studies in terms of functionality and performance. The experimental results confirms that its functionality satisfies all the requirements of transactional processing and the constrains of uniqueness. The performance is enhanced considerably when comparing to others.

The method proposed in this paper for primary key management is applable for scalable DBMS with higher performance. Furthermore, its abstract model of WAR is adaptable for other database systems.

Key Words: Scalable Database Management System, Read/Write separation, Concurrency Control, Primary Key Generation, Primary Key Modification

目 录

插　图

表　格

第一章 绪　论

1.1 研究背景与意义

数据库技术一直是计算机领域的一个热门课题，它将数据以一定的形式组织起来形成数据库。数据库管理系统(Database Management System, DBMS)是数据库技术的一个重要应用，DBMS 对数据库中存储的数据进行统一的管理和控制，并且保证数据库的安全性和完整性。在计算机文件系统中，一般通过文件路径和文件名访问某个文件，而在数据库管理系统中，往往需要给每个数据项指定一个标识符，方便用户访问特定的数据项。

从上世纪 60 年代末开始，传统的数据库系统经历了层次数据库[1]、网状数据库[2]和关系数据库三个阶段，其中使用最广泛、影响最深远的是基于关系模型的关系数据库。关系模型[3]由 E.F Codd 在 70 年代提出，它将现实世界的实体以及实体之间的联系用关系来表示，并将数据的逻辑结构表示为一张二维表，每个数据项对应二维表中的行，从而引入了主键(Primary Key)[3]的概念。主键即是数据项的标识符，它唯一地标识了二维表中的每一行数据。和数据行的普通列一样，用户可以针对主键进行一系列的增、删、改、查等操作。关系模型以其数据结构简单、理论严谨、易于理解等特点迅速得到认可，被广泛应用于数据库系统中，主键成为数据库的一个重要组成部分。

此后，关系数据库进入快速发展期，到本世纪初，关系数据库已在市场上占据统治地位，几乎所有的主流商用数据库是基于关系模型的，例如 Microsoft 公司的 SQL Server[4]、Oracle 公司的 Oracle 数据库[5]和 IBM 公司的 DB2[6]等。这类关系数据库系统的服务模式一般采用客户端-服务器架构，即将数据集中于单个服务器中，所有数据库共享存储空间，集中处理来自客户端的服务请求。

近年来，互联网应用快速积累而膨胀的海量数据在数据库系统的可扩展性、低延迟访问以及高可用性等方向提出了更高的要求。在以应用需求为导向下，分

布式存储技术[7]在近十年的时间里取得了长足的进步，并开始从理论研究转向工业实践，各种新型的分布式数据库不断面世，成为了互联网大数据存储应用中不可忽视的一支力量。特别是高可用性和高扩展性著称的新型的 NoSQL 键值数据库和 NewSQL 数据库，成为了新的研究热点。

NoSQL 键值数据库打破了关系数据库的范式约束[8]，它采用键值(Key-Value)数据模型，特别适合于存储半结构化的数据，例如 Amazon 公司的 SimpleDB[9]、MongoDB（原 10Gen）公司的 MongoDB[10]、Redis[11]和 Memcached[12]、Google 公司的 BigTable[13]以及 BigTable 的开源版本 HBase[14]。NoSQL 数据库的海量的数据存储能力和的优异的动态水平扩展性使得传统集中式关系数据库望尘莫及，但它以放弃 SQL 查询语言和事务强一致性要求为代价，这很大的限制 NoSQL 键值数据库的应用。而 NewSQL 数据库是对目前各类可扩展数据库的简称，这类数据库既支持海量数据的存储，又支持标准 SQL 语言和事务特性，是介于传统关系数据库和 NoSQL 之间的一类数据库，例如 Google 公司的 Spanner[15]、Microsoft 公司的 SQL Azure[16]、VoltDB[17]以及阿里巴巴集团的 OceanBase[18]等。NewSQL 数据库支持关系模型，比 NoSQL 键值数据库具有更广泛的应用范围，同时也具有良好的扩展性，代表未来分布式数据库的发展趋势。

在上述介绍中，可以清晰地看出数据库管理系统从集中式架构到分布式架构、从传统关系数据库到 NoSQL 键值数据库和 NewSQL 数据库的发展脉络。在诸多类型的数据库中，都绕不开主键的概念。主键是区别不同数据行的关键词，无论是关系模型还是键值模型都是基于主键访问数据的。因此，在 NoSQL 键值数据库和 NewSQL 数据库中，不可避免地和传统关系数据库一样面临着主键维护和管理的问题。主键维护是指生成、更新和删除主键的操作，是传统关系数据库中的一项基本功能。

可扩展数据库系统一般采用读写分离的方式来提高系统的可扩展性。读写分离机制将数据按时间划分为增量数据和基线数据，并且分别存储于不同的服务器节点。只能对增量数据进行修改，读取操作需要同时访问增量数据和基线数据，增量数据定期合并到基线数据中去，从而将读请求和写请求分流到不同的服务器

节点。本文将针对读写分离机制的特点，研究如何在该类可扩展数据库中实现正确且高效的主键维护功能。本文的研究具有重要的研究价值和现实意义。

首先，在关系模型中，需要保证表格中每一条记录的主键都是唯一的，即需要满足主键唯一性约束，主键维护功能的实现需要考虑在分布式的网络环境中，且数据被切分为增量数据和基线数据的情况下不破坏主键唯一性约束；其次，需要应用事务处理技术，使得主键维护相关操作在执行过程中满足事务的 ACID 特性；最后，在满足主键唯一性约束和事务性的前提下，还需要考虑采取一定的优化措施使得主键维护功能达到较好的性能。因此，在可扩展数据库中实现主键维护功能具有一定的研究价值。

同时，基于分布式架构的可扩展数据库正处于起步和推广的初期阶段，尽管在系统架构上与传统关系数据库有很大的差异，但其在 SQL 接口上正越来越向关系数据库看齐，即在利用分布式存储技术解决数据库海量存储问题的同时，也尽量支持传统关系数据库系统提供的功能，而目前可扩展数据库系统对主键维护功能的支持程度不高。因此，增加对主键维护功能的支持也将有利于可扩展数据库系统的推广和使用，具有重要现实意义。

1.2 本文工作

本文以可扩展数据库中的主键维护的问题为出发点，总结了一类具有普遍意义的先读后写（Write After Read, WAR）的更新操作。在该操作中，数据库系统首先读取数据库中的某个数据，经过适当处理后，再写入到数据库中，WAR 操作本质是数据库长事务中一个由读操作和写操作组成的特例。本文针对 WAR 操作如何保证事务性提出通用的解决方案，并针对某些特殊场景进行了优化，然后将解决方案应用于维护主键值功能的设计与实现中。本文的的工作主要包括以下几个方面：

1) 提出了一类在可扩展数据库中保证 WAR 类型操作并发执行的事务性的模型，分别是 PCC-WAR (Pessimistic Concurrency Control for WAR) 模型和 OCC-WAR (Optimistic Concurrency Control for WAR) 模型，其中前者

是基于封锁协议的悲观模型，而后者是基于有效性验证的乐观模型。同时针对某些特殊应用场景，提出了基于缓存的优化方案。

2) 分别基于 PCC-WAR 模型和 OCC-WAR 模型提出了可扩展数据库中实现维护主键功能的多种设计方案，这些方案可同时满足单行事务性和主键唯一性约束条件，分别适应于不同的应用场景，并且表现出了较高的性能。

3) 将本文提出维护主键功能设计方案在开源的分布式可扩展的关系数据库 OceanBase 进行了实现，并通过一系列实验对比验证了该方案的正确性和高效性，从实践的角度证明本文提出的 PCC-WAR 模型和 OCC-WAR 模型的正确性和可行性。

本文提出的 PCC-WAR 模型和 OCC-WAR 模型具有通用性，不仅适用于主键维护功能，而且可应用于其它任何需要先读取一次数据再更新的场景，例如 SELECT INTO 功能和某些特殊带子查询的更新操作等。本文的研究可以为在可扩展数据库系统实现相关功能的研究提供借鉴。

1.3 本文结构

本文的结构组织如下：

第一章和第二章介绍了本文研究的背景和相关工作，介绍了可扩展数据库管理系统、数据库主键、读写分离存储架构和事务处理等相关概念和技术，并介绍了基于读写分离机制的开源可扩展分布式数据库 OceanBase 的系统架构、存储引擎和读写事务流程等。

第三章给出了主键唯一约束相关的形式化定义，并描述了应用程序对可扩展数据库管理系统中主键维护的需求，最后分析了读写分离的可扩展架构实现主键维护功能的挑战。

第四章详细描述可扩展数据库中 WAR 操作的模型：PCC-WAR 模型和 OCC-WAR 模型，同时应用事务处理相关理论证明了该模型的正确性，最后针对某些特殊场景对该模型进行了流程优化。

第五章详细介绍了基于 PCC-WAR 模型和 OCC-WAR 模型维护主键的的设计方案，并且应用基于缓存的优化方案对生成主键功能进行了优化。

第六章是实验部分，针对可扩展数据库系统的维护主键功能进行了一系列的验证和对比实验，验证本文提出的设计方案的正确性和高效性。

第七章总结本文的全部研究内容，并对未来的研究工作进行了展望。

第二章 相关工作

本章首先将介绍基于读写分离的可扩展数据库的实现原理以及可扩展关系数据库 OceanBase 的架构，然后介绍事务并发控制的相关技术，最后介绍主键维护相关技术在传统关系数据库和可扩展数据库系统中的应用现状。

2.1 可扩展数据库管理系统

数据库的可扩展性(Scalability)或可伸缩性是指数据库系统服务过程中动态调整系统整体性能的能力。在集中式的数据库中，性能的扩展主要采用向上扩展(Scale Up)的方式，比如通过增加 CPU 核数、内存容量和磁盘容量等方式提高处理能力，这种方式往往比较僵化且提升空间有限，已经越来越不适应海量数据的存储和计算需求。而新型的可扩展数据库管理系统(Scalable Database Management System)利用分布式存储技术，将大量廉价的服务器聚集成一个集群，通过增加或减少集群内部服务器节点的数目来线性地调整集群的处理能力。这种由普通服务器组成的集群架构不仅摆脱对大型设备的依赖，而且又具备向外扩展(Scale Out)的能力。

2.1.1 读写分离存储技术

采用集群架构的数据库系统一般采用读写分离(Read/Write Spliting)的方式提高系统的可用性和可扩展性。传统的读写分离架构是一个无共享(Shared-nothing)[19]的架构，将数据分别存储为主(Master)和备(Slave)两个副本，在主副本上执行写操作，而在备副本上执行读操作，同时主副本利用数据库的复制技术，将数据的更新同步到备副本中，例如 MySQL 集群[20]等。显然，无论是 Master 节点还是 Slave 节点都必须存储数据库全部的数据，集群存储容量扩展性受限于单个服务器节点的单机存储能力。

新型读写分离架构的可扩展数据库一般是基于 LSM 树[21](Log-Structrured

Merge Tree, 日志结构合并树)实现的。Log-Structrued 是由 Rosenblumn 和 Ousterhout 提出[22]的一种磁盘存储管理方式，它将磁盘内容的更新操作顺序地写入一个类似日志的结构中，这种顺序写入方式可以加快文件的写入速度。LSM-Tree 是利用计算机存储器的层次结构，将最近的更新操作保存在缓冲区，当缓冲区的大小达到某个阈值时，将修改数据批量转储到磁盘中，最后当磁盘文件达到一定阈值后，进行压缩操作，将多个旧文件合并成一个新的文件[21]。

基于读写分离机制的可扩展数据库系统存储引擎的底层是一棵 LSM 树，LSM 树由两个或多个类似树的数据结构组件组成，其中只有一个组件存储在内存中。最简单的 LSM 树包含两个组件C_0树和C_1树，如图 2.1 所示。

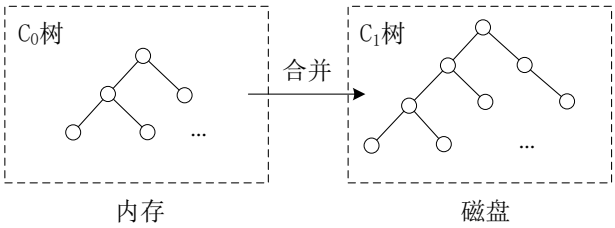


图 2.1　LSM 树的经典两组件结构

在上图中，C_0树和C_1树分别存储于内存和磁盘，所有写操作都追加到内存中的C_0树，所有读请求同时读取C_0树和C_1树。当C_0树达到一定大小后，分配一个新的空C_0'树，新的更新操作都追加到C_0'树中，同时，将C_0批量合并到C_1树中。类似的，含有 n(n>2)个组件C_0、C_1、…、C_n的 LSM 树，其中C_1、…、C_n等 n-1 棵树存储于磁盘中，当C_1树大小达到一定阈值后，合并到C_2树中，以此类推。

LSM 树延迟并批量顺序处理更新数据的操作，有效地降低了磁盘 I/O 次数，从而提高系统整体性能。基于读写分离机制的可扩展数据库系统非常适合大规模数据的存储和管理，尤其是写操作数据量占比不大但对写性能要求较高的场景，该类数据库系统具有以下几个特点：

1) 将数据按时间划分为增量数据和基线数据分开存储。

2) 将数据的修改保存在内存中，读操作需要同时访问内存和磁盘。

3) 存在一个增量数据向基础数据滚动合并的过程。

增量数据和基线数据是读写分离架构的可扩展数据库系统的重要概念，它们

的具体含义分别是：

- **增量数据**：又称为动态数据，最近更新的数据，一般存储于服务器主存中，可供读写，一般以追加的形式更新增量数据。
- **基线数据**：又称为基准数据或静态数据，由增量数据转储和压缩而来的数据，一般存储于服务器磁盘中，只可读。

在可扩展数据库系统中，本文将增量数据存储的服务器称为增量数据节点，简称为“增量节点”，类似的，将基线数据存储的服务器称为基线数据节点，简称为“基线节点”。集群环境中一般部署了多个增量节点，根据这多个增量节点是否可以同时读写增量数据，读写分离架构可分为单节点写入和多节点写入两种模式

在单节点写入模式中，增量节点一般设置为一主一备或一主多备，其中只有主增量节点拥有执行写事务的权限，备增量节点通过日志保持和主节点的状态一致。该模式的优点在于将读写操作集中于一台服务器，退化成传统的单机内存存储系统，规避了分布式事务，可以很好地支持事务管理。同时，其局限性也是明显的，将读写操作集中在一台服务器中执行，该服务器的性能很可能成为整个系统的一个瓶颈。采用单点写入模式的可扩展数据库有阿里巴巴的 OceanBase 等。

在多节点写入模式中，增量数据的分为多个副本存储于多个增量节点，其中每个增量节点都具有读写权限，且同时对外提供可提供读写服务。所有增量数据节点可同时接收客户端的读写请求，各增量节点需要使用两阶段提交(Two-phase Commit, 2PC)协议[23]和 Paxos 协议[24]等分布式协议保证跨节点操作的原子性。相对于点写入模型，该模型解决了单点写入节点性能瓶颈的问题，但增加了读写操作复杂性，实现该模型的系统一般在读写性能和事务性保证两方面有一定的折衷。例如 BigTable 仅支持单行事务，Spanner 虽然支持多行事务，但牺牲了一定的读写性能。

2.1.2 可扩展数据库管理系统 OceanBase

OceanBase 是阿里巴巴集团为满足其快速增长的数据存储和处理的需求，而

设计并实现的可扩展的分布式关系数据库，它基于增量数据和基线数据分离的架构，同时采用单节点写入的模式。本节将介绍 OceanBase 的系统架构、内存事务引擎和读写事务的流程。

（1）系统架构

OceanBase 中包括四类服务器节点[25] [26]，分别为：主控服务器 RootServer、更新服务器 UpdateServer、合并服务器 MergeServer 和基线数据服务器 ChunkServer，分别对应图 2.2 中的总控节点、增量节点、查询节点和基线节点。

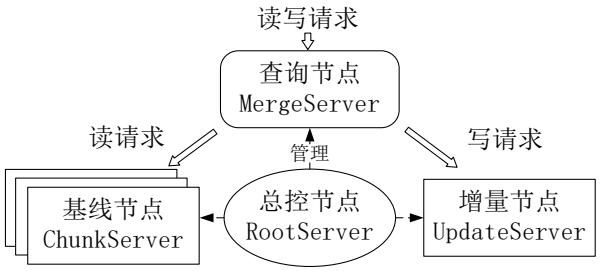


图 2.2　OceanBase 的架构

- RootServer 主控服务器：简称为 RS，它是整个集群的总控制中心，管理集群内所有的服务器节点。
- UpdateServer 更新服务器：简称为 UPS，它存储了所有的增量更新数据，是系统中惟一可执行写事务的服务器节点，支持主备同步。
- MergeServer 合并服务器：简称为 MS，它是对外提供读写服务的接口，支持 JDBC 和 ODBC 协议，协调事务执行。
- ChunkServer 基线数据服务器：简称为 CS，它用于存储基准数据。

（2）内存事务引擎

OceanBase 的内存事务引擎由 UpdateServer 上的内存表（MemTable）及其管理组件组成，实现满足 ACID 特性的数据库事务管理[27]。MemTable 是存储数据和处理查询的核心内存数据结构，它由索引结构和行操作链表两部分组成，如图 2.3 所示。其中索引结构为 B+树，支持插入、删除、更新、随机读取和范围扫描操作。数据以行为单位存储在 MemTable 中，每次更新事务对行数据的更新（包

括删除标记）记录都保存在一个变长的内存块中，称为一个“Cell”操作，这些内存块按时间顺序连接成链表，当读取某一行数据时，遍历整个链表将同一列的更新操作合并，然后与基线数据对应行数据融合得到最终完整的数据。

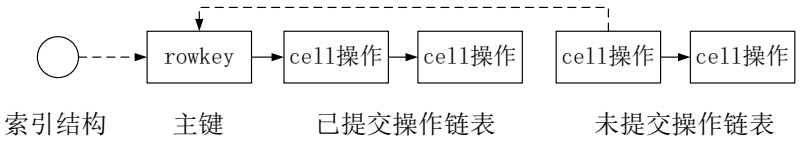


图 2.3　内存表 MemTable 的结构

MemTable 通过 MVCC(Multi-Version Concurrency Control，多版本并发控制)机制实现并发修改。行操作链表包含已提交链表和未提交链表两部分，首先事务执行线程互斥地锁定待更新行，然后将更新操作追加到该行的未提交操作链表末尾，在提交事务前，生成操作日志批量发送至备机并刷入本地磁盘，提交事务时，只需要将未提交行操作链表的头指针指向已提交行操作链表的尾部即可。

（3）读写事务流程

OceanBase 采用读写分离机制将写事务集中在增量节点，而将读事务分配给各个基线节点。因此，读事务流程和写事务流程存在很大的区别，如图 2.4 所示：

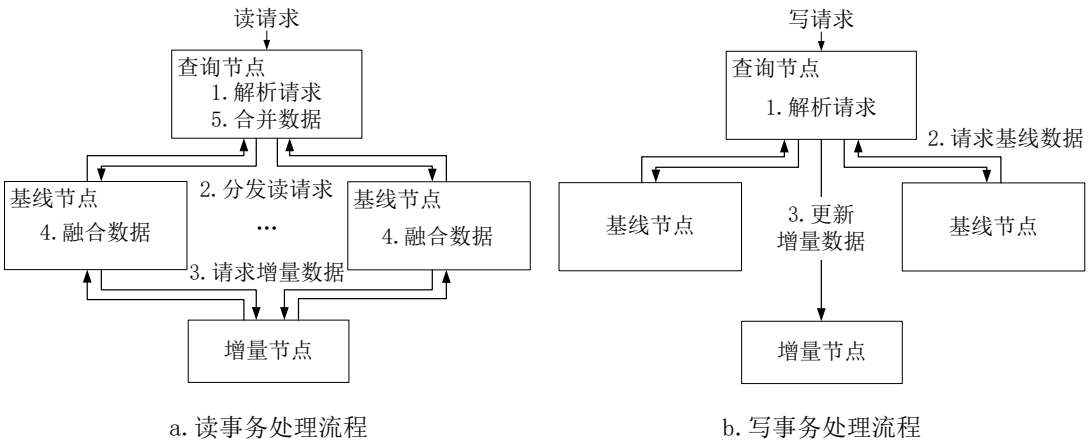


图 2.4　OceanBase 的读事务和写事务的处理流程

只读事务即只有查询操作，不涉及数据的修改，其执行流程如图 2.4 中图 a 所示，在只读事务中，首先 MergeServer 解析 SQL 语句，转换成系统内部的执行计划，并将请求拆分后同时发给多个 ChunkServer 并发执行，ChunkServer 查询

本机的基线数据，并向 UpdateServer 请求对应的增量数据进行融合得到完整的数据。每台 ChunkServer 将各自读取数据返回 MergeServer，MergeServer 将这些数据合并后返回给客户端。

写事务包括插入、更新和删除等操作，它可能需要读取基线数据，用于判断更新或插入的数据行是否存在或执行原列值自增等操作，执行流程如图 2.4 中图 b 所示，在写事务中，首先 MergeServer 解析 SQL 请求，转换成系统内部的执行计划，同时向 ChunkServer 请求待写行的基准数据，并将基准数据和执行计划一起发送至 UpdateServer。UpdateServer 查询出待写行的增量数据，与基准数据融合得到完整的数据行，计算出新的数据行追加到内存表中，并在这个过程中记录操作日志。

（4）OceanBase 的特点

由于 OceanBase 读写分离架构和单节点写入模式的特殊性，OceanBase 具有以下几个特点：

1) OceanBase 采用典型的读写分离的内外存混合存储的架构，拥有分布式存储系统特有的高可扩展性、高可靠性等优点。

2) OceanBase 采用单节点更新模式，实现了集中式的写事务处理，支持范围查询，支持跨行跨表事务，且支持关系数据库的 SQL 功能。

OceanBase 具有高可扩展性、高可用和代成本的优点，非常适合于互联网应用大规模数据的存储和管理，拥有广阔的发展前景。

2.2 事务处理技术

2.2.1 事务概述

事务(Transaction)[28]是传统关系数据库一个重要的概念，它是一个原子的工作单元，由一组预先定义好的逻辑操作序列组成，执行这一组操作后，系统将由初始的一致状态变换到下一个一致的状态，并完成预先定义的功能。事务具有几个重要性质，分别是原子性(Atomic)、一致性(Consistency)、隔离性(Isolation)和持

久性(Durability)，简称为事务的 ACID 特性，由 OpenGroup 在事务处理 XA 规范(1991)中定义[29]，即：

1) 原子性：一个事务内的所有操作要么全部执行，要么全部不执行。当事务非正常终止时，需要撤销其对数据库已做的所有修改影响。

2) 一致性：事务的执行结果应该满足某些数据完整性约束。

3) 隔离性：事务执行过程中产生的中间状态对于其它事务来说是不可见的，防止其它事务对该中间状态的数据进行读写操作。

4) 持久性：一个事务一旦完成，它所产生的结果是持久的，而与事务提交后系统发生的故障无关，事务的操作结果具有可恢复性。

事务的上述特性要求事务只能以两种方式结束：提交(Commit)或回滚(Rollback)。事务提交指该事务中的所有操作结果都应用到数据库中，并存储到易失性的存储介质中，而事务回滚指的是将该事务相关操作所做的修改全部撤消，事务任何一个操作的失败都将触发事务的回滚操作。

2.2.2 并发控制

（1）并发控制概念

事务并发指的是数据库系统中多个事务同时、相互交叉执行的现象。理想情况下，事务应该是串行执行的，即系统中任意时刻只允许一个事务处理执行状态，其它事务必须等待该事务执行结束后才能开始执行。串行执行可保证多个事务间完全不干扰，实现也简单，但串行执行的方式不能充分利用系统资源，过多的事务处于等待状态，导致事务的响应时间过长，在实际应用中很少使用。事务并发执行可以有效的利用系统资源，但若不加以控制，则会带来一系列问题，如：

- 丢失更新(Lost Update)：一个事务对数据的更新被其它事务覆盖，或某个事务的回滚导致前一个已提交事务的修改丢失；
- 读脏数据(Dirty Read)：事务读取到其它未提交事务中间状态的无效数据；

- 不可重复读(Unrepeatable Read)：事务在执行过程中两次读取相同数据的结果不相同。

因此，必须对事务的并发执行进行控制。事务并发控制技术在数据库技术几十年的发展历程中一直是研究的热点领域，其核心是在保证事务 ACID 特性中的隔离性和一致性，经过学术界和工业界的大量研究和实践，形成了基于锁协议、基于时间戳和乐观并发控制三类并发控制技术[30]。

（2）并发控制技术

a. 基于锁协议的并发控制

基于锁协议的并发控制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，只有申请到锁以后，才可以对数据项进行操作[31]。数据库中锁的研究较为成熟，锁的类型有多种。通常数据库支持两种锁：共享锁(Shared Lock, S 锁)和排他锁(Exclusive Lock, X 锁)。共享锁用于读取数据，防止其它事务对该数据进行修改操作，排他锁用于修改数据，防止其它事务对该数据进行读取和修改操作。

在基于锁的并发控制中，通常采用两阶段封锁协议(Two-phase Locking Protocol, 2PL) [32]来保证事务的可串行化调度。其基本思想是：一个事务的加锁和释放锁被严格地安排在增长和收缩两个阶段进行，增长阶段（Growing phase）：事务可以获得锁，但不允许释放锁；收缩阶段（Shrinking phase）：事务可以释放锁，但不允许再获得锁。若所有事务都遵守两段锁协议，则能保证这些事务是冲突可串行化的。

封锁机制往往会导致死锁(Deadlock)[33]问题，即并发执行的一组事务相互等待其他事务占有的资源，所有事务都不能完全持有本事务继续执行所需要的资源，导致整个系统处于无法正常工作的状态。死锁的处理一般分为死锁预防（Deadlock Prevention）和死锁恢复（Deadlock Recovery）两种方法。死锁预防是指系统主动地采取某些措施保证避免进入死锁状态，其方法一般是提前对所有数据项赋予一个次序，一个事务在执行前，首先分析自己在执行过程中所需要的锁，然后按照数据项的次序申请对应的锁。而死锁恢复指的是系统进入死锁状态后，

及时地采用适当的恢复机制使系统从死锁状态中恢复过来，一般是使用某种策略放弃一部分事务，使得其它事务得以继续执行。

b. **基于时间戳的并发控制**[34][35]

时间戳(Timestamp)是系统给每个事务分配的一个唯一的数值，可惟一地标识一个事务，时间戳记录该事务最后读和写数据的时间或先后顺序，时间戳可以使用系统时钟和逻辑计数器两种方式表示。若使用系统时钟表示，则系统按照当前的系统物理绝对时间为每个事务赋予唯一的时间戳；若使用逻辑计数器表示，则系统维护一个内部计数器，当一个新事务开始时，计数器自增 1 得到新的时间戳并分配给该事务。在基于时间戳的并发控制中，调度事务按时间戳排序的串行调度等价于事务的实际调度[36]。

c. **乐观并发控制**

乐观并发控制(Optimistic Concurrency Control)[37][38]技术的基本思想是允许事务不封锁数据，无阻碍地执行直到全部操作完成，在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，这种方式是假设本事务与其它事务没有冲突，因此称为乐观的并发控制。采用乐观并发控制的事务执行分为读阶段(Read Phase)、验证阶段(Validation Phase)、写阶段(Write Phase)三个阶段：

1) 读阶段：读取出待操作的数据，并复制到事务独立的局部工作空间，事务仅针对该数据副本进行操作，不对系统中原始数据进行真正修改。

2) 验证阶段：进行有效性测度(Validation Test)，检查事务调度是否可串行化。若可串行化，则进入写阶段，否则当前事务回退重试。

3) 写阶段：事务将更新操作应用到系统中，对数据库真正产生影响。

在检验阶段，设T_i为待检验事务，T_k为任意在T_i前检验的事务，事务T_i的有效性测试要求对任意事务T_k必须满足以下两个条件之一[39]：

- T_k在T_i开始之前已经执行结束。
- T_k所修改的数据项集与T_i所读数据项集不相交，并且T_k的写阶段在T_i开始检验阶段之前已结束。

在乐观并发控制的机制中，事务首先需要在私有的数据区内对数据的复本进行修改，这需要额外的耗时和存储空间，而且一旦有效性验证未通过，事务需要回滚重新执行，而重启的代价较大，因此乐观并发控制适合于写冲突较小的场景。

2.3 数据库管理系统中主键的维护

2.3.1 主键和主键的维护

键（Key）是标识一个数据值（Value）的关键字，通过键可以访问特定值，进而对值进行一系列操作。在数据结构中，键值技术的一个简单的应用是哈希表（Hash Table）[40]结构，通过散列函数将键映射到表中的一个地址来直接访问记录值，用于加快读取数据的速度。

在基于主键的数据库系统中，为了方便数据的存储和管理，通常采用基于哈希表、B 树等数据结构的存储引擎，而存储引擎即是这些数据结构在主存、磁盘或 SSD 等存储介质上的实现，数据库系统在存储引擎的基础上，根据不同存储引擎的特性封装了对数据增、删、改、查的接口。根据实现原理的不同，存储引擎可分为三种[41]：

- 哈希存储引擎：哈希存储引擎基于哈希表结构，它支持增、删、改和随机读取操作，但不支持顺序扫描。
- B 树存储引擎：B 树存储引擎基于 B 树或 B 树的变体结构，它不仅支持单条记录的增、删、读、改操作，还支持顺序扫描。
- LSM 树存储引擎：LSM 树存储引擎基于 LSM 树结构，与 B 树存储引擎一样支持增、删、改、随机读取和顺序扫描，同时，它可通过批量转储技术有效地规避了磁盘随机写入问题，但增加了读取操作的代价。

不同的存储引擎都需要通过键来操作数据，在数据库系统中，主键不仅是数据行某个方面的属性重要信息的载体，而且具有非常重要的作用：（1）主键唯一的标识一个数据单元（例如一个数据行），通过主键来判断两个数据单元是否为同一数据，对特定数据的读取和更新都是基于主键的。（2）主键将两张不同数据

表中的数据进行关联。主键在数据库系统中具有不可替代的重要地位。若没有主键，对数据的更新和删除将变得非常繁琐，主键代表了关系数据表的完整性。

在分布式可扩展数据库系统中，主键值不仅用于访问数据（包括随机访问和顺序扫描），而且在数据存储的底层，主键拥有更重要的功能，例如：根据主键范围对数据进行水平切分[42][43]，实现分布式存储的负载均衡等。

由于主键的特殊性，数据库的更新操作往往是对除主键之外的普通数据的修改，主键值一般只拥有可读属性。但是，在实际应用中，主键还承担着代表一行数据某个特殊含义的的功能，往往代表着实体一个非常重要的属性，它可能随着实体属性的变化而变化。在这个层次上，主键列和普通非主键列是同等的，意味着需要如同修改普通列一样修改主键值，修改主键值的操作就属于主键维护(Primary Key Management)的范畴。主键的维护主要是指对主键的生成、更新、删除的操作，主键在维护的过程中应满足一定的约束条件和完整性条件，例如每条记录必须具有主键，且不同行的主键值不能重复等。

2.3.2 数据库管理系统中主键维护的现状

本文调研了主流的传统关系数据库和几个开源的 NewSQL 数据库的主键维护功能的支持现状。其中，生成主键值是传统关系数据库常见的功能，它们对主键值的自动生成有不同的支持方式。自动递增的数值类型占用存储空间小，处理效率高，是系统生成主键的首选，主流的关系数据库都支持这一生成主键的方法，主要分为以下两种方式：

- 直接方式：在定义表结构时给主键列添加特殊标记，插入新数据行时直接生成自动递增的数值型主键值，例如 MySQL[44]、DB2 和 Microsoft SQL Server 等。
- 间接方式：使用数据库提供的其他功能生成递增的数值型数据，并赋值给主键，间接地实现生成主键的功能，如 PostgreSQL[45]、DB2 和 Oracle 等通过使用序列(Sequence)可生成自动递增的数值型数据。其中序列是一个用来生成一系列唯一数值数据的对象，支持独立事务，每次访问一个

序列，该序列的值按照一个预定的量增加或减少，事务的提交或回滚不影响序列值的变化。

VoltDB 和 OceanBase 等 NewSQL 数据库当前版本没有支持生成自增数值型主键的功能。而对于更新主键值，传统的关系数据中的 UPDATE 语句对所更新列是否是主键没有限制，即都支持更新主键的功能，而 NewSQL 数据库中只是部分支持。不同数据库对维护主键功能的支持程度如表 2.1 所示：

表 2.1　不同数据库系统对主键维护功能的支持情况统计

数据库名称	版本	生成主键	更新主键
Microsoft SQL server	SQL server 2005	√	√
Oracle	Oracle Database 11g	√	√
DB2	DB2 Express-C	√	√
MySQL	MySQL 5.7.15	√	√
PostgreSQL	PostgreSQL 9.5.4	√	√
VoltDB	VoltDB 6.6	×	√
OceanBase	OceanBase 0.4.2.17	×	×

由上表可知，传统关系数据库普遍都支持维护主键功能，而部分 NewSQL 数据库对维护主键功能的支持程度不高，这阻碍了应用系统从传统数据库向可扩展数据库的迁移，不利于该类新型数据库的推广和使用。

2.4 本章小结

本章介绍了可扩展数据库管理系统和主键维护的相关工作。首先介绍了可扩展数据库管理系统的可扩展性、基于 LSM 树的读写分离存储技术的基本原理以及可扩展的分布式关系数据库 OceanBase 的实现流程，然后概述了数据库系统中的事务处理技术，包括并发控制技术等。最后，介绍了主键概念和作用，以及传统关系数据库和主流可扩展数据库系统对维护主键功能的支持现状。

第三章 问题描述与分析

3.1 基本定义

关系模型是一种经典的表示数据和数据间联系的模型，它将现实世界的实体以及实体之间的联系表示为关系，适合描述结构化数据，该模型被大量现代的数据库系统所采用。关系模型相较于键值模型等在数据库系统中应用和发展较早，相关技术也相对成熟，逐渐成为数据库的行业标准。

定义3.1 关系(Relation)：在n个集合$D_1, D_2, \ldots, D_n$之上的关系R是一个n元组$(d_1, d_2, \ldots, d_n)$的集合，它满足$d_1 \in D_1$，$d_2 \in D_2$，$\ldots$，$d_n \in D_n$。

由关系的定义可知，关系是多个元组的集合，而元组包含了多个属性，其中键(key，又称为码)是属性集合的一个子集，构成键的属性值可以唯一地标识该关系的一个元组。组成键的属性被称为主属性，每个关系至少有一个键，根据需要可以指定多个键，每个可能的键称为候选键，在所有的候选键中选择一个候选键作为主键。在关系模型中，关系可以映射为一个二维表格，关系中的每个元组对应于表格的一行，因此，主键又可称为行主键(Row key)，而元组的某个属性对应表格的列。

定义3.2 主键(Primary Key)：唯一标识每一个元组，是关系中一个属性或多个属性组成的集合。定义关系R(K, A, B, ...)，可构造规模为n的元组集合：

$$(k_1, a_1, b_1, \ldots), (k_2, a_2, b_2, \ldots), \ldots, (k_n, a_n, b_n, \ldots)$$

其中k_i即为每个元组的主键，k_i可能为单个属性，也可能是多个属性的集合，$a_i, b_i, \ldots$为普通属性值，$(k_i, v_i) \in R, 1 \leq i \leq n$。

由主键的定义可知，主键唯一地标识着关系中一个元组，若关系中存在“两个”元组的主键值相同，则这“两个”元组必定是同一个元组，同一张表格中的所有数据必须满足主键唯一性约束条件。主键唯一性约束是数据库数据完整性约

束中的一个重要条件，它保证了数据的准确性和一致性。

定义3.3 主键唯一性约束：在同一个关系中，不存在任何两个元组在主键分量上具有相同的值。假设存在关系R(K, V)，其中K为关系R的主键，将关系R重命名为R1(K, V)和R2(K, V)，即：

$$\rho_{R1(K,V)}(R)$$

$$\rho_{R2(K,V)}(R)$$

则在关系R上构造所有元组对(k,v)，其中不可能存在两个元组对在K分量上的值相同，而在V分量的值不同，通过做笛卡尔乘积和选择操作，该定义可形式化的表示为：

$$\sigma_{R1.K=R2.K\ AND\ R1.V\neq R2.V}(R1 \times R2) = \emptyset$$

目前主流的关系数据库和 NewSQL 数据库都是基于关系模型的，关系模型是现代数据库管理系统的基础。

3.2 应用系统中主键维护的需求

3.2.1 需求描述

传统关系数据库的数据库功能已发展得相当完备，一般都全面地支持标准SQL[46]语言，包括对主键维护和管理的功能，而新型的 NewSQL 数据库由于发展较晚和分布式环境与单机环境的差异，它们在数据库功能的支持程度并不高，特别是缺乏对主键维护功能的支持。

维护主键的功能在数据库管理系统中非常具有必要性。一方面，大部分应用系统只是使用主键的可读属性，即主键值只是用于唯一标识一行数据，而不属于数据行对应实体的属性的一部分。例如：在某应用系统中设计一张用户 User 表，若新用户注册时系统自动分配一个唯一的标识符 user_id，则这个标识符可作为用户表的主键，但用户实体中并没有 user_id 这个属性，user_id 只有应用系统内部使用，该值只是为了方便应用系统识别不同的用户，此时主键不具有实际的含义，主键可以是不重复的任意值。因此，数据库系统应该提供一个自动生成唯一主键的功能。而另一方面，在某些应用系统中，主键值是实体的属性的一部分，

可能需要随着实体属性的变化而变化。例如：同样是设计用户 User 表，若新用户注册时要求用户人工输入一个用户名 user_name，并要求该用户名是惟一的，以便可以使用该用户名进行登录操作，为了简化实现，某些应用系统可能将 user_name 作为 User 表的主键，user_name 是用户实体的一个属性，用户可能有修改 user_name 的需求。因此，数据库系统应该提供修改主键值的功能。综上所述的分析和举例，可以看出，为应对多样化的用户需求，降低上层应用系统的实现复杂度，在数据库管理系统中实现自动生成主键值和修改主键值等通用功能是非常必要的。

维护主键是标准 SQL 中一项基本的功能，在可扩展数据库中，维护主键需求分为以下两个方面：

1) **生成主键**：在传统关系数据库中，应用最广泛的是自增数值型主键，因此，在可扩展数据库中，应提供自动生成自增数值型主键的功能。

2) **更新主键**：更新主键是指像修改普通列值一样方便地修改主键列值。

为了提高兼容性，要求可扩展数据库生成自增主键值和更新主键值的功能应与传统关系数据库的 SQL 接口类似。如在创建表格时，可通过关键词标记该表格的主键为自增数值类型，在插入新数据库行时，自动给该行分配一个唯一的自增的主键值；在更新数据行时，可以不区分主键列和非主键列进行任意修改。

3.2.2 外部解决方案

目前，大量应用系统都是基于传统关系数据库开发的，在应用系统从传统关系数据库向可扩展数据库迁移的过程，若可扩展数据库不提供维护主键的功能，可通过外部解决方案在一定程度上实现维护主键的需求。

外部程序通过使用数据库的事务功能，将一组数据库系统已提供的简单功能的接口封装为一个复杂的操作，间接实现特定的功能。例如：修改主键值时，可以外部程序中先查询出待更新的行数据，然后与新主键值一起拼接出更新后的行数据，最后将新数据行插入数据库，同时删除旧数据行，达到修改主键值的目的。外部实现方法主要存在以下几个问题：

1) 事务性：将操作转换为多次请求很难保证单次插入或更新操作的事务ACID 特性，实现完整的并发控制较为困难。

2) 执行效率：在上层进行操作转换，需要与数据库服务器产生多次网络通信和数据转输，执行效率往往不高。

3) 复杂性：对原有应用系统和中间件软件进行改造是一件高复杂性、高风险的任务，应该尽量避免之。

4) 通用性：外部解决方案都是针对特定的应用系统进行定制化开发，不具有通用性。

综上分析，维护主键是应用系统的合理需求，外部实现方法只是临时解决方案，其最终解决方案应该是在可扩展数据库系统中直接支持，其优点也是显而易见的：

- 数据库系统支持该通用型功能，可减少应用程序的开发难度和人力成本。
- 在数据库系统内部实现特殊机制，可为维护主键操作实现高效性和事务性提供了可能。
- 分布式的可扩展数据库是数据库系统的发展方向，数据库系统在实现分布式存储的同时，也应提供 SQL 功能和事务支持，这样可极大程度地减少应用程序从传统关系数据库系统迁移出来的阻碍，有利于可扩展数据库系统的推广使用。

3.3 可扩展数据库中维护主键的难点

生成和修改主键和更新主键虽然涉及对多个行的操作，但在传统集中式数据库系统中，这些操作都在单机上执行，在异常处理、事务性保证和并发控制等方面的实现相对简单，而在分布式的环境中，网络通信带来很多不确定因素，特别是读写分离的可扩展架构下，维护主键功能在实现时需要满足主键唯一性约束、保证一定的事务性和达到一定的性能要求。

（1）主键唯一性约束

生成和修改主键值首要原则是不能破坏主键唯一性约束的条件，这是数据完整性的基础。与集中式数据库不同，在读写分离的可扩展架构中，每张表格中的数据分布在增量数据节点和基准数据节点中，其中某个行的数据有可能全部存储于基线数据节点，有可能全部存储于增量数据节点，也有可能在增量数据节点和基线数据节点各存储一部分，这增加了检查主键重复的复杂性。

（2）事务性保证

事务处理技术中的并发控制技术在传统关系数据库应用体系已非常成熟，用于保证事务的 ACID 特性。但是在可扩展数据库系统中，出于对数据库性能的考虑，往往对事务性支持的程度不高。但是维护主键的操作在理论上只应该涉及到一行数据，因此应该增加特殊的机制以保证行级别的事务性，特别是保证生成主键值和修改主键值操作的原子性，以及并发执行时事务的一致性和隔离性。

（3）性能要求

在满足主键唯一性约束和事务性的前提下，应该使得修改主键值的操作尽可能的高效。其性能应该明显优于外部间接实现的主键维护功能的性能，并与修改非主键列值相比，其性能指标应该在一个可接受的范围内，且不影响原有系统其它操作的性能。

3.4 本章小结

本章首先对主键和主键唯一性约束做出了形式化的定义，然后分析了维护主键功能的必要性，并描述了应用系统中主键维护和管理的需求，以及系统外部间接实现维护主键功能的方法，分析比较了这些外部解决方案的优缺点，得出可扩展数据库系统应该提供主键维护功能的结论。最后分析了在可扩展数据库系统中实现主键维护功能的难点，包括主键唯一性约束、单行事务性保证和性能要求等。

第四章 先读后写更新模式

本文在介绍读写分离的可扩展数据库管理系统中实现主键维护的方案之前，将在本章描述一个数据库系统中的常见的先读后写的更新操作场景，然后针对该类场景提出其实现并发控制的方法，作为实现主键维护功能的理论基础。本章将分别介绍基于悲观并发控制协议和乐观并发控制协议的 PCC-WAR 模型和 OCC-WAR 模型，详细描述这两个模型的基本原理，论证其正确性并从理论上分析和比较两个模型的性能。最后，将提出优化方案，并分析其应用场景。

4.1 先读后写操作

4.1.1 场景描述

在数据库范畴内，更新(Update)指的是修改特定数据的值。传统关系数据库中更新数据的流程可分为读取、修改和写回三个步骤，即：首先从非易失性存储介质中读取待更新的数据并加载到主存，然后在主存中修改该数据，最后将新值写回至非易失性存储介质中。

而在读写分离的分布式架构下，更新操作需要首先从基线节点中读取待更新数据的基线数据，然后发送至增量节点，增量节点将基线数据和增量数据合并，并对合并后的数据进行修改，最后将修改后的数据存储在增量节点。在实际应用中存在一个特殊的场景：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修改数据。该类更新操作包含一个显示的查询数据库的过程，为了与普通更新操作区别，本文称之为显示的先读后写(Write After Read, WAR)的更新，简称为先读后写更新。

WAR 操作是数据库中一类特殊的长事务[47]，它包含了读取和写入两类操作，使用数据库事务封装的 SQL 语句描述，如 **SQL 1** 所示：

SQL 1 WAR 类型事务

```
1. BEGIN
2. SELECT <columns> FROM <table> WHERE <condition> FOR UPDATE
3. UPDATE[INSERT|DELETE] <table> SET <expression> WHERE <condition>
4. COMMIT
```

如 **SQL 1** 所描述的事务称之为 WAR 类型事务，该类事务可能包含多条读取语句和多条写入语句，但所有的读取语句都在写入语句之前。其中读取操作通常使用 SELECT ... FOR UPDATE 语句，在查询数据的同时锁定结果集，而写入操作可以使用任何对数据产生影响的语句，如 UPDATE、INSERT 和 DELETE 等。

本章接下来的内容将介绍在数据库系统内部执行 WAR 操作并发控制的方式及其应用，并针对某些特殊的场景进行流程优化。

4.1.2 基本定义和流程

为了便于描述，本章给出基准值、目标值和目标行的基本定义：

定义 4.1 基准值(Baseline Value)：定义基准值v_B为数据库中存储的某个已存在的值，由用户自定义的查询条件 Q 查询基线数据d_{static}和增量数据$d_{dynamic}$合并后获得，即：

$$v_B = Q(d_{static}) \cup Q(d_{dynamic})$$

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v_B不限为单个值，即有可能为值的集合。基准值v_B必须通过查询数据库才能获取，且可以同时被其它事务修改。

定义 4.2 目标值(Target Value)：定义目标值v_T是以基准值v_B为输入，经过自定义的目标表达式函数 F 运算后的输出值，即：

$$v_T = F(v_B)$$

目标值通常与基准值具有相同的含义，查询节点在本地将基准值的副本通过目标表达式函数转换成目标值（在使用完目标值后，往往需要将目标值更新至数据库中，成为新的基准值）。

定义 4.3　目标行(Target Row)：定义目标行r_T是目标值v_T以自定义关系 f 关联的数据行，即：

$$v_B \xrightarrow{f} r_T$$

目标行由目标基线数据s_T和目标增量数据d_T组成，即：

$$r_T = s_T \cup d_T$$

WAR 操作的最终目的是修改目标行，因此，需要通过目标值访问目标行，为了简化和加快对目标行的访问，一种特殊情况是目标值和目标行通过主键关系关联，即目标值一般是目标行的主键。对于某个特定类型的 WAR 操作，目标表达式函数 F 和关联关系 f 是固定不变的。因此，对于给定的 F 和 f，可以定义一类 WAR 操作。

在基于读写分离架构的可扩展数据库系统中，执行 WAR 操作需要查询节点、基线节点和增量节点相互协作，若不未考虑异常情况的处理，WAR 操作的执行流程如图 4.1 所示：

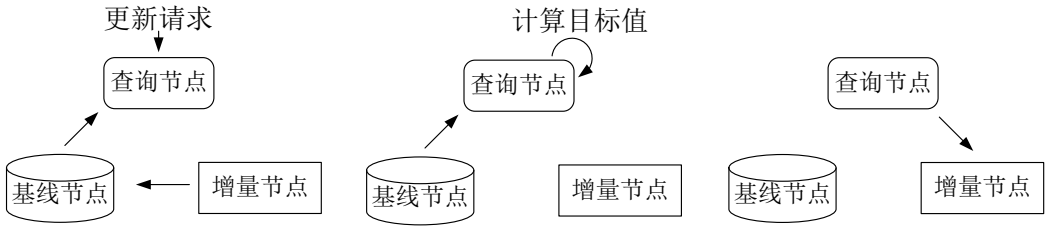


图 4.1　WAR 操作的执行流程

在上图所描述的流程中，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，已经获得了目标值和目标行的基准数据，增量节点真正更新目标数据行，同时，可能也会更新基准值，对应流程中 c 步骤。

基准值v_B记录了 WAR 操作执行前后的数据库的某个状态，它决定了待更新的数据行，以及本次请求是否可执行成功，同时，该更新请求执行完成后又可能会反过来决定基准值的新值v_B'，数据库从上一个状态进入下一个状态。图 4.2 描述了两个连续执行的 WAR 操作WAR_1和WAR_2，首先WAR_1读取了基准值v_B，然后更新目标行r_{T1}，基准值由v_B更新为v_B'，紧接着WAR_2读取出基准值的新值v_B'，更新了另一个目标行r_{T2}，最终基准值更新为v_B''，对基准值和目标行的更新依次交替进行下去。

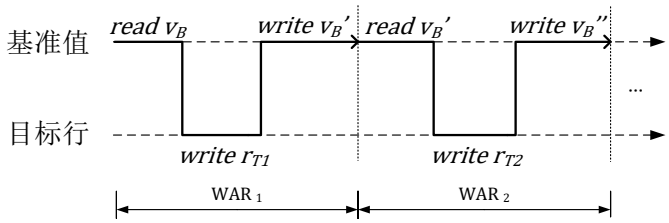


图 4.2　基准值状态的更新

若在写阶段需要更新基准值，则最简单的 WAR 操作至少包含了 1 个读操作和 2 个写操作，读操作和其中 1 个写操作是针对基准值的，为了保证数据库系统在执行 WAR 操作时从上一个一致性状态原子性地切换至下一个一致性的状态，需要增加并发控制机制，保证并发执行 WAR 操作不破坏事务的 ACID 特性。

在数据库系统中，并发控制的方法一般可分为两类，即悲观并控制控制协议和乐观并发控制协议，依据这两种协议可建立悲观并发控制的 WAR 模型（PCC-WAR）和乐观并发控制的 WAR 模型（OCC-WAR）两个模型。

4.2 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。

4.2.1 基本原理

PCC-WAR 模型基本原理是使用封锁技术，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它事务读取或修改，这是一种悲观的策略。若 WAR 操作需要读取多个基准值，则需要同时

申请多个基准值的锁，并按申请锁顺序的逆序释放锁。

查询节点查询基准值，必须读取基准值的增量数据，因此锁定增量数据，即可限制其它事务对基准值的读操作；在读写分离架构的可扩展数据库中，只有增量数据是可修改的，因此锁定增量数据，也可限制该其它事务对该基准值写操作。增量节点在事务提交后，释放基准值上的锁，此时整个 WAR 操作才算结束。

4.2.2 锁的选择原则

PCC-WAR 模型中的封锁机制针对的是基准值，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。就锁的权限而言，基准值上的锁有共享锁（S 锁）和排他锁（X 锁）两种选择，其选择的原则是：

- 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：若在写准备阶段只申请共享锁，存在两个事务同时更新相同目标行，这两个事务都执行到了写阶段，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提前申请排他锁。

- 共享锁：除上述情况外，可放宽锁的权限为共享锁，只禁止其它事务的写请求。

排他锁对资源的访问限制比共享锁更严格，一般而言，适用共享锁的场景也可以使用排他锁，但是排他锁对资源的独占性会引起其他事务不必要的阻塞和等待，导致系统的读写性能的降低。因此，在满足基本需求的前提下，应该选择代价较小的共享锁。

4.2.3 死锁的处理

使用封锁机制的 PCC-WAR 模型避免不了死锁的现象。例如存在包含 WAR 操作的两个事务 A 和 B，A 事务读取基准值 n，更新目标行 a，B 事务也读取基准值 n，更新目标行 b。在某一时刻，A 事务已经获得了基准值 n 的排他锁，开

始更新目标行 a，正准备申请目标行 a 的排他锁，而 B 事务正准备读取基准值 n。此时，若 B 是一个长事务，在此之前已经占有目标行 a 的共享锁，由于排他锁和共享锁是互斥的，则 A 事务会一直等待目标行 a 的排他锁，A 事务被阻塞，而由于 A 事务一直占有基准值 n 的排他锁，B 事务也在一直等待读取基准值 n，B 事务被阻塞。A 事务和 B 事务互相占有对方所需要的资源，都无法执行下去，从而发生死锁现象。

PCC-WAR 模型使用超时机制来处理死锁问题。超时机制是分布式系统的网络环境中处理请求异常的常用方法，PCC-WAR 模型在发出读取基准值的请求前设置一个超时时间，如果在超时结束以前，没有收到响应回复，则认为该操作处于死锁状态而放弃本次操作，并等待一定时间后发起重试操作。读取基准值的流程的超时设置如图 4.3 所示：

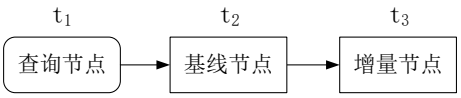


图 4.3 读取基准值的超时设置

读取基准值的流程涉及三个节点两次请求，需要两个网络请求超时时间t_1和t_2，以及一个锁等待超时时间t_3。考虑到网络传输的延时，一般设置$t_1 > t_2 > t_3$。

- t_1：查询节点查询基准值的超时时间；
- t_2：基线节点查询基准值增量数据的超时时间；
- t_3：增量节点申请基准值增量数据锁的超时时间。

超时机制的一个难点是确定一个合适的超时时间，若时间过短，则会误取消事务；若时间过长，则会延长死锁的持续时间，进而降低系统的性能。由于不同应用的差异较大，通常超时时间设置为一个比事务平均执行时间更长的时间[48]。特别地，在采用单节点写入模式的可扩展数据库，增量节点往往容易成为系统的性能瓶颈，应该尽量减少增量节点的资源使用量。因此，PCC-WAR 模型在应用时一般设置t_3=0，即若申请不到锁也立即返回，避免阻塞增量节点事务执行线程，

将等待和重试操作转移到查询节点或基线节点，分担增量节点的压力。

4.2.4 正确性分析

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，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机制。一个只查询一个基准值并且只更新一行目标行的 WAR 操作在执行过程中，一共需要申请 2 个锁，如图 4.4 所示：

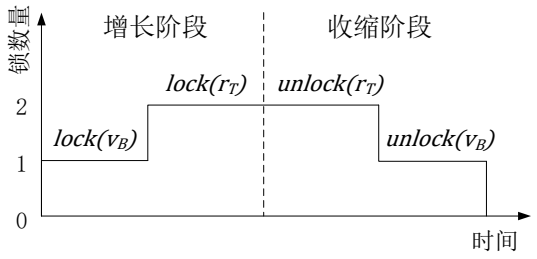


图 4.4　PCC-WAR 模型两阶段封锁的原理

在上图中，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是可以保证事务调度是冲突可串行化的[32]，因此，PCC-WAR 模型可以保证 WAR 操作的事务性。

4.3 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。

4.3.1 基本原理

OCC-WAR 模型的基本原理：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。若 WAR 操作需要读取多个基准值，则需要同时验证多个基准值的有效性。

乐观并发控制协议分为读阶段、有效性验证阶段和写阶段三个阶段。在 OCC-WAR 模型中，查询节点读取基准值，并对基准值的副本进行操作，计算出目标

值，增量节点将所有的更新操作都都追加到增量数据的未提交区域，对应“读阶段”；在提交事务前，进行有效性验证，来决定当前事务是否可以提交，对应“有效性验证阶段”；若有效性验证通过，则提交事务，将当前事务对数据库的修改操作应用到已提交区域，对应“写阶段”。

4.3.2 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它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它事务修改过（和读取过），验证在读阶段读取到的基准值副本在事务提交前是否依然有效的。而判断基准值是否被修改过（和读取过）方法一般有两种：1）每个事务记录读取的基准值的状态，2）记录每个修改（和读取）基准值的事务的顺序，如图 4.5 所示：

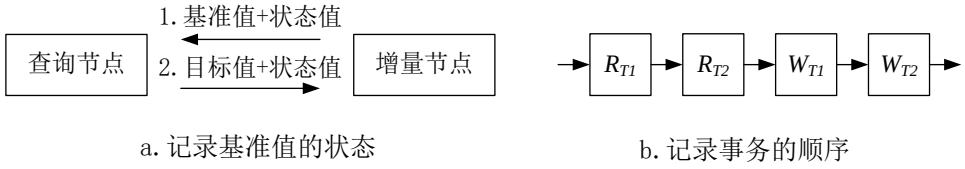


图 4.5　有效性验证的实现两个方法

第一种方法图 4.5 中图 a 所示。查询节点在读基准值时，同时获得该值的一个状态值，例如版本号或最后一次被修改（和读取）的时间戳等，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比对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，否则回滚事务。该方法实现简单，验证流程的效率高，但只适用于支持多版本的数据库系统，否则需要额外在空间中记录修改（和读取）的时间戳。

第二种方法如图 4.5 中图 b 所示。系统增加一个读写历史链表结构记录和维护每个基准值被不同事务修改（和读取）的顺序，例如R_{T1}、W_{T1}分别表示该基准值被事务T_1读取过和修改过。当查询节点在读阶段读取该基准值时，将事务号加入到基准值的读写历史链表中，在有效性验证阶段，判断该链表中同一个事务号后面是否存在其它事务的读或写操作，若不存在则验证通过。该方法需要增加额外的链表结构，实现较为复杂，并且需要考虑链表的并发修改和访问。

上述两种方法各有优缺点，分别适用于不同的场景。在读写分离架构可扩展数据库系统中，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，本文提出的 OCC-WAR 模型选用第一种方法进行有效性验证，即查询节点查询基准值的时间戳用于判断基准值的状态是否改变。

4.3.3 事务的重启

在 OCC-WAR 模型中，若有效性验证阶段验证失败，则需要进行重试操作。由于 WAR 操作由查询节点、基线节点和增量节点相互协作完成，其中查询节点是控制中心，因此重试操作只能由查询节点发起。查询节点可以设置一个超时时间或重试最大次数，在该超时时间或重试最大次数范围内进行有限的重试操作。

4.3.4 正确性分析

基于有效性检查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，在可扩展数据库系统中，时间戳值的对比和事务的提交可以看做是一个同时执行的瞬时行为，假设存在 a、b、c 和 d 四个事务对同一个基准值进行读取，如图 4.6 所示，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)。

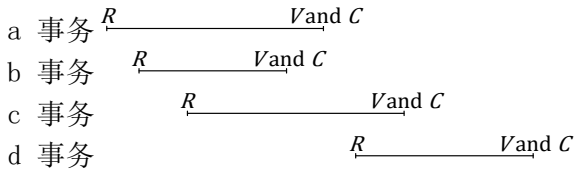


图 4.6　OCC-WAR 模型有效性验证举例

在上述图中，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，说明没有其它同时执行的事务与当前事务发生写写冲突（或读写冲突），当前事务可提交，只要存在任何其它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修改（和读取），都会导致有效性验证失败，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。

4.4 优化与应用

4.4.1 基于缓存的优化

缓存指的是复制远程的数据并存储于本地，根据局部性原理[49]，通过在本地保存一份数据的冗余副本，可以减少访问远程数据的次数，从而加速数据的访问。缓存技术最初是为了解决 CPU 时间和内存时间不匹配的问题而提出的，CPU 首先到高速缓存中读取数据，若没有命中才到内存中去读取，可以提升 CPU 的处理速度。

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。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，增加了事务并发控制的复杂性，且耗时较长，在某些特殊情况下，可以省略查询节点获取基准值的这一过程。例如，当基准值数据量较小，且对目标行的更新操作不需要使用其基线数据时，可将所有的基准值的缓存在增量节点的内存中，避免查询基线数据，从而达到简化流程和优化性能的目的，这是本节介绍的基于缓存的优化方法的基本思想。

（1）基本流程

基于缓存优化后的 WAR 操作的基本流程如图 4.7 中图 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。

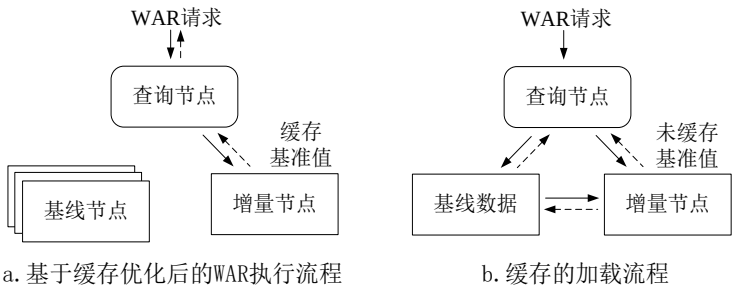


图 4.7　基于缓存优化的 WAR 执行流程和缓存加载流程

对比原始的 WAR 流程（见 4.1.2 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）缓存的加载

在初始情况下，增量节点的内存中没有基准值的缓存值，在某次 WAR 请求到达增量节点里，若未命中基准值的缓存，则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 WAR 请求和基准值一起发送至增量节点执行，如图 4.7 中图 b 所示，在本次 WAR 操作结束后，该基准值的缓存则一保存在增量节点的内存中，以供下一个 WAR 请求使用。

缓存的加载过程必须经过查询节点的原因是因为在基于读写分离的可扩展数据库系统中，往往存在多个基线节点，而只有查询节点缓存了基线数据在各基线节点的分布情况，增量节点无法确定向哪个基线节点请求基准值的基线数据，因此，增量节点在执行 WAR 请求时，若发现本地未缓存基准值，则直接返回查询节点错误信息，由查询节点读取基准值后再重试，这一方法的另一个优点是不会阻塞增量节点上宝贵的事务执行线程资源，保证其它事务的正常执行。

（3）缓存的维护

由于读写分离架构的特殊性，数据库中的每一个数据项都是由基线数据和增量数据两部分组成，这两部分融合后才能获取数据项的完整值，因此，在基于缓存优化的 WAR 操作中，对缓存维护变得简单，只需将基准值的完整值保存在增量节点即可，不需要设置特殊的存储空间来额外保存基准值的缓存，缓存的加载过程只是将基准值的基线数据和增量数据融合后再追加至其增量数据中。在每日合并后，增量数据会全部合并到基线数据中，此时，缓存的基准值重新变为空值，只需要重新加载一次基准值即可。

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。

（4）性能分析

基于缓存优化的 WAR 操作尽量避免了对基线数据的访问，因此，使用基于缓存的优化方法的 WAR 特殊场景需要满足以下两个基本条件：

a) 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。

b) 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。

经过缓存优化后，对基准值的操作限定于增量节点中，不仅简化了并发控制的流程，而且还带来了以下几个方面的性能提升：

1) 缩短了事务执行时间。在缓存命中的情况下，减少了向基线节点查询基准值耗时。

2) 未增加额外的空间代价。基准值的缓存即是基准值的增量数据。

综上所述，基于缓存的优化可以简化某些特殊情况的 WAR 操作的流程，并提升其性能。

4.4.2 WAR 模型的应用

WAR 操作代表了在数据库系统内部执行一类通用的操作，本文提出的 PCC-WAR 模型和 OCC-WAR 模型在数据库具有广泛的应用场景，例如生成自增主键、更新主键值、SELECT INTO 和部分含子查询的更新等，如表 4.1 所示。

表 4.1　WAR 的应用举例

应用	基准值v_B	目标值v_T	目标表达式 F	关联关系 f
自增主键	最大主键值	新主键值	v_T=v_B+1	主键
更新主键	原行	新主键值	UPDATE 输入	主键
SELECT INTO	来源行	新行	v_T=v_B	自身
子查询更新	子查询结果	待更新行主键	UPDATE 输入	主键

1) 生成自增数值主键：在插入新数据行时，系统自动根据当前表格中最大

的主键值自增得到新的主键值作为新数据行的主键，保持表格中的数据行的主键是连续自增的，需要记录表格当前最大的主键值作为基准值，在每次插入新行时使用并更新该基准值。

2) 更新主键值：将数据行的主键更新为新值，需要查询待更新行数据作为基准值，在插入新行后删除该基准值。

3) SELECT INTO：查询出某张表格的数据再插入到另一张表格中，此时查询的数据即为基准值，在插入到另一张表格中后，不需要更新基准值。

4) WHERE 条件中含简单子查询的更新：对于 WHERE 条件中含简单子查询的更新操作，需要首先查询出更新行的主键为作为基准值，更新目标行完成后不需要更新主键值。

综上可见，WAR 是数据库中的一种典型的功能实现的模型，它虽然是一类特殊的长事务，通过客户端进行长事务封装也可达到相似功能，但是相对于通用长事务功能，WAR 模型的优点有：

1) WAR 模型可以应用于某些数据库功能的实现，相对于客户端封装长事务执行有明显的性能优势，且可简化客户端的实现逻辑；

2) 通用的长事务一般只支持封锁的方式实现并发控制，而 WAR 模型提供更多的并发控制选择，如基于有效性验证的乐观并发控制等；

3) WAR 可以针对特殊场景进行流程优化，简化某些不必要的操作，从而达到优化性能的目的。

4.5 本章小结

本章总结并提出了基于读写分离的可扩展数据库中的一种先读后写的更新操作的抽象框架，然后介绍了分别基于悲观并发控制协议和乐观并发控制协议的 PCC-WAR 模型和 OCC-WAR 模型，最后，介绍了一个基于缓存优化 WAR 操作的方案以及 WAR 模型的几个典型应用。WAR 框架及其模型可以应用于主键维

护功能的设计与实现。

第五章 可扩展数据库中主键维护的实现

主键维护指的是生成主键和更新主键的操作。本文提出了针对读写分离架构的可扩展数据库中 WAR 操作的两个模型：PCC-WAR 模型和 OCC-WAR 模型，本章将在主键维护功能的设计与实现中应用这两个模型，提出多种实现方案，并进行分析对比。

5.1 主键的生成

5.1.1 设计概述

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的基础上自增一而生成的新主键值。因此，在计算新主键值之前，首先需要获得数据表中当前的最大主键值，一般有以下两种获得当前最大主键值的方法：

1) 扫描全表法。该方法通过扫描全表数据以获得表格中主键列的最大值。显然，当数据量较大时，该方法的操作效率非常低，且新主键可能与表格中删除行的主键重复。

2) 记录法。利用额外的数据表记录和维护表格的当前最大主键值。当新插入数据时，先读取该该的当前最大主键并加一后作为新主键再进行插入操作。该方法需要增加并发控制机制以避免数据的重复插入。

虽然扫描全表法可以实时获取表格当前的最大的主键值，但往往在扫描全表的过程中，需要锁定全表防止最大主键值变化，增加了实现的复杂度并降低了系统整体的性能，特别是在分布式环境下扫描全表的代价过大。因此，本文采用的是第二种方法来生成主键值。

在传统关系数据库系统中，通常分配有一个公共空间结构用于记录系统运行

相关的参数和指标，该公共空间结构称为系统表或内部表，类似的，可新增一张系统表，并命名为“__all_max_id”，用于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的数值，其表结构如表 5.1 所示：

表 5.1　生成主键的系统表__all_max_id 的表结构

字段名称	数据类型	主键	备注
table_id	INTEGER	是	NOT NULL
column_id	INTEGER	是	NOT NULL
max_id	INTEGER	否	NOT NULL

每次在该用户表中新增一行数据，系统表中字段“max_id”的数值就自增 1，并且该值只增加不减少。生成自增数值型主键的是一类典型的 WAR 操作，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 F 即是最大主键自增量 1 的操作，而查询条件 Q 指根据表名映射系统内外部的表 ID。

在不破坏主键唯一性约束条件且不考虑并发控制的情况下，插入新数据行并自动生成自增主键值的两个阶段分别是：

1） **写准备阶段**：查询节点接收到在某个表格中插入新行数据的请求，根据表格的 id 查询系统表__all_max_id 获取表格中当前最大的主键值，并在此基础上自增一，作为新行的主键值。查询节点根据查询条件向对应的基线节点发出查询请求，获得新行的基线数据，并随执行计划一起发送至增量节点。

2） **写阶段**：增量节点以新主键值为条件查询增量数据，并与新行基线数据（可能为空）融合，若该数据不为空，表明新行不存在，将新行插入至增量数据中，插入成功后再更新系统表中记录的最大主键值；否则，说明新主键与已有数据行的主键存在冲突，返回主键冲突的错误，然后查询节点从第 1）步开始进行重试操作，重新计算主键值。

由于生成主键功能需要读写同一个最大的主键值，冲突较大，不适合 OCC-

WAR 模型。因此，本章只应用 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，在插入数据成功后才更新记录的新最大主键值，保证每个主键值都被使用，若插入新行数据失败，则不影响记录的最大主键值。

5.1.2 基于 PCC-WAR 模型

基于 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，待事务提交后释放锁，在插入新主键前，根据新主键值查询基线数据和增量数据判断新主键是否已存在，在某些异常情况下，主键可能重复，则将最大主键值自增一发起重试。其具体执行流程在算法 1 中进行了详细描述：

算法 1 基于 PCC-WAR 模型的生成自增主键值算法

输入：不包含主键值的新行数据 new_row

```
1:  exclusive _lock(max_rowkey)
2:  读取 max_rowkey
3:  计算主键值 rowkey := max_rowkey + 1
4:  根据 rowkey 查询基线数据 static_data
5:  根据 rowkey 查询增量数据 dynamic_data
6:  if static_data + dynamic_data == NULL then
7:      插入新行(rowkey 和 new_row)
8:      更新最大主键 max_rowkey := rowkey
9:      同步操作日志并刷盘
10:     commit()
11:     unlock(max_rowkey)
12: else
13:     更新最大主键 max_rowkey := rowkey + 1
14:     retry()
15: end if
```

5.1.3 基于缓存的优化

生成自增数值主键操作中基准值是数据表中最大的主键值，这是一个数据量较小的基准值，且普通插入新行操作读取基准数据的目的只是用来验证主键是否

重复，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，这分别满足 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流程。

在增量节点缓存了最大主键值的情况下，基于缓存优化后 PCC-WAR 生成自增主键值的流程如算法 2 所述：

算法 2 基于缓存和 PCC-WAR 模型的生成自增主键值算法

输入：不包含主键值的新行数据

1: **exclusive_lock**(*max_rowkey*)
2: 读取 *max_rowkey*
3: 计算主键值 *rowkey* := *max_rowkey* + 1
4: 插入新行(*rowkey* 和 *new_row*)
5: 更新最大主键值 *max_rowkey* := *rowkey*
6: 同步操作日志并刷盘
7: commit()
8: **unlock**(*max_rowkey*)

当增量节点没有缓存最大主键值，则由查询节点新查询一次基线数据和增量数据，发送到增量节点，在插入新行完成后，更新最大主键值为新值，则自动缓存了最大主键值。

5.1.4 非连续自增主键

基于 PCC-WAR 模型生成自增主键值的方法在插入新数据行时保证了主键的连续有序自增，即确保每个自增生成的主键都分配给实现的数据行，不会浪费惟一的主键值资源，适合于要求主键值连续的应用场景。但一般情况下，当应用程序使用系统自动生成的自增主键值时，该主键往往只是形式主键，不具有实际的含义，对用户而言，主键是透明的，其存在的价值只是方便存取数据。此时，不需要保证主键值连续有序自增，基于此前提，本文提出了基于缓存生成非连续自增主键的方法，如算法 3 所示描述。

算法 3 基于缓存生成非连续自增主键值算法

输入：不包含主键值的新行数据

1: 更新并读取最大主键值 *max_rowkey* := *max_rowkey* + 1
2: 计算主键值 *rowkey* := *max_rowkey*
3: 插入新行(*rowkey* 和 *new_row*)
4: 同步操作日志并刷盘
5: commit()

在生成非连续自增主键的方法中，查询节点在查询当前最大主键值时即将该该值自增 1，且不再锁定该值，增量取得这一主键值用于插入新行。该方法可保证主键是自增，因此每个主键值都是唯一的，没有破坏主键惟一性约束，但是没有保证事务性，因为无论新数据行是否插入成功，最大主键值都自增 1。

5.1.5 分析与比较

本节描述了三个生成自增主键值的方法，分别是：基于 PCC-WAR 模型的方法、基于缓存优化的 PCC-WAR 模型方法和基于缓存非连续自增的方法，其中前两者保证了主键值的严格连续自增。

生成主键的需要读取和更新同一个基准值，冲突非常大，生成主键的功能无法做到真正的同时执行，且在没有经过缓存优化的方法中存在查询节点查询基线数据的一个过程，增加了其它事务的阻塞等待时间。因此基于 PCC-WAR 模型的方法能较差。而增加缓存后，其它事务虽然也需要等待执行，但阻塞时间减少，因此，对性能的提升较大。基于缓存生成非连续自增的主键进一步缩短了基准值的锁定时间，减少了事务的阻塞，因此，对性能的提升最大，但该方法不适用于需要主键连续的场景。

5.2 主键的更新

5.2.1 设计概述

在数据库系统中，使用索引结构可以快速地访问特定的数据，本文在 2.3.1

节介绍了三类存储引擎，即 B+树存储引擎、哈希存储引擎和 LSM 树存储引擎，在哈希结构中数据行的主键即是键，而在树形结构中，主键值决定了数据叶子结点的到根结点的路径。因此，在存储系统底层结构中不同的主键值对应不同的数据行，主键值的修改意味着在索引结构中删原行，并插入新行，而新行与原行除主键外其它数据完全相同，这是更新主键值的基本原理。

更新主键值本质是一个在增量数据中插入一行，同时删除一行的过程，其中一个必要步骤是将新主键和原行非主键列的值组合成新行，并且根据需要在待更新行的基础上计算新主键值，为了方便表述，本章中定义：

- 原行(Original Row)：待更新的行。
- 新行(New Row)：更新主键后生成的新的行。

更新主键是一类典型的 WAR 操作，原行对应 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 F 和查询条件 Q。

在不破坏主键唯一性约束条件，并且不考虑并发控制的情况下，更新主键的基本步骤对应 WAR 操作的两个阶段：

1） **写准备阶段**：查询节点接收到用户的更新请求（即一条 UPDATE 语句），通过解析该请求获得新主键值，若更新请求的新主键的更新表达式是诸如将主键值自增 1 或在其它普通列值的基础上做简单运算等形式，则先需要查询出原行的完整数据并计算出新主键值。查询节点根据查询条件向对应的基线节点发出查询请求，获得原行完整的基线数据（包含所有列的基线数据），如果新行在基线数据中存在，则同时也查询出新行的基线数据，和执行计划一起发送至增量节点。

2） **写阶段**：增量节点以新主键值为条件查询增量数据，与随执行计划发送来的新行基线数据（可能为空）融合，若该数据为空，在本机查询出原行的增量数据，并与对应的基线数据融合，获得原行的最新的完整的数据，

和新主键值组成新行数据，插入至增量数据中，并删除原行数据；否则，说明新行的主键值与数据表中已有的主键重复，返回主键冲突的错误。

分别应用 WAR 操作的 PCC 模型和 OCC 模型实现更新主键的并发控制。

5.2.2 基于 PCC-WAR 模型

（1）基本流程

基于 PCC-WAR 模型更新主键功能在查询原行数据前，将原行数据加锁，待更新主键事务提交后再释放锁，因为更新键最终需要删除原行，即修改基准值，因此选择排他锁，即在更新主键期间禁止其它 WAR 类型的事务读取和修改原行。其具体执行流程在算法 4 中进行了详细描述：

算法 4 基于 PCC-WAR 模型的更新主键算法

输入：原行主键值$K_{original}$，新行主键值表达式 F

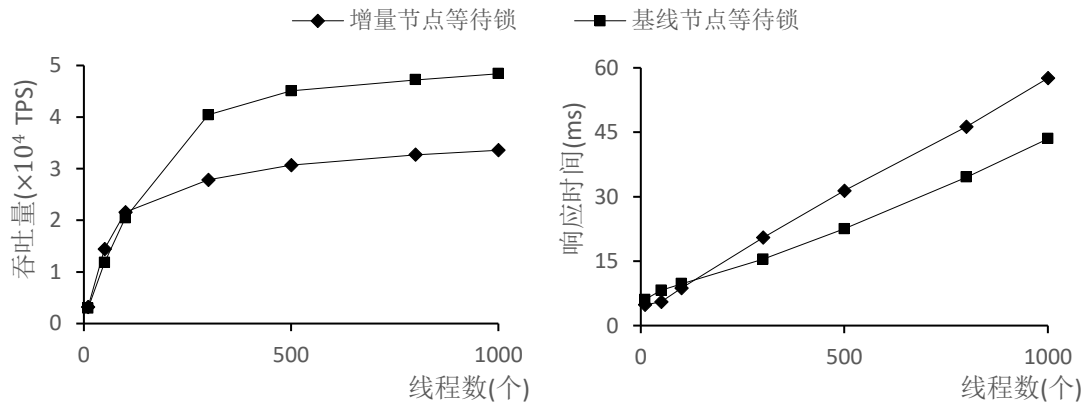
1: **exclusive_lock**(*original_row*)
2: 读取原行 *original_row*
3: 计算新主键值K_{new} := F(*original_row*)
4: 根据K_{new}和 $K_{original}$查询 *new_row_static_data* 和 *original_row_static_data*
5: 根据K_{new}和$K_{original}$查询 *new_row_dynamic_data* 和 *original_row_dynamic_data*
6: **if** *new_row_static_data* + *new_row_dynamic_data* == NULL **then**
7: 插入新行(K_{new}和 *original_row_static_data* + *original_row_dynamic_data*)
8: 标记删除旧行 *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**

（2）封锁的冲突调度

OceanBase 的内存事务引擎同时实现了 MVCC 和 2PL 的并发控制技术。内存表 MemTable 中按时间顺序保存每一次写操作的历史，在每日合并时将所有的修改历史合并到基线数据中，修改记录分为“已提交”和“未提交”两部分，读

和写事务分别读取和追加这两部分的记录，读取增量数据的操作不需要加共享锁，从而保证了不会出现读写冲突，因此，在读阶段对基准值所加的排他锁只会影响 WAR 类型事务，不会影响正常的读取操作。

当两个事务同时尝试更新同一行的主键值时，就发了写写冲突，其中一个先到的事务获得排他锁，另一个事务需要阻塞等待排他锁的释放。在 OceanBase 中，一次查询操作需要经过查询结点、基线节点和增量节点三个服务器，因此有三个等待锁选择，分别阻塞查询节点、基线节点的增量节点上的执行线程来等待锁，本文实现了在查询节点和增量节点上等待锁的两种方式，分别测试不同线程的 TPS 和平均响应时间，如图 5.1 所示。



a.不同锁等待方式的吞吐量　　b.不同锁等待方式的响应时间

图 5.1　在不同节点等待锁对更新主键吞吐量和响应时间的影响

由以上两图可知，在查询节点等待锁的事务并发执行的性能要优于在增量节点等待，这是因为 OceanBase 单节点写入模式的特殊性，所有写事务都集中到了增量节点，增量节点的 CPU 资源有限，如果阻塞过多事务执行线程，会影响增量节点的整体的性能，从上述折线图可以看到，在连接线程数较高的情况下，在增量节点等待锁的更新主键的 TPS 明显降低，而其平均响应时间较长。因此，本文采用在查询节点等待锁的方式。

在 WAR 的写准备阶段，查询节点发出查询原行数据的读请求，若不能立即获取排他锁，则等待一定时间后发起重试，在超时时间内反复尝试，直至查询成功，在相同的实验环境下，针对不同的锁等待超时时间，分别测试其与夭折事务

数目、TPS 和响应时间的关系，测试结果如图 5.2 和图 5.3 所示：

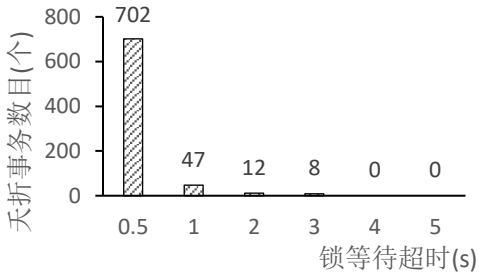
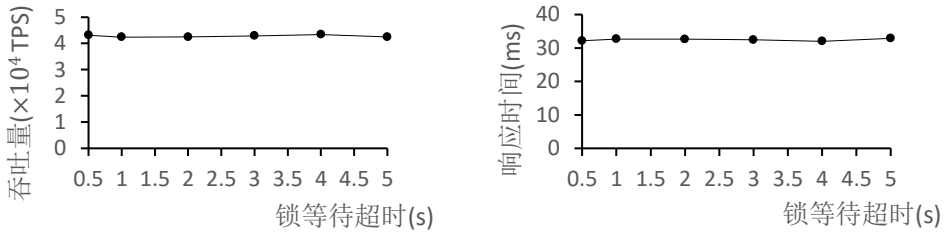


图 5.2　锁等待超时对更新主键的夭折事务数目的影响

由图 5.2 可知，锁超时时间越小，夭折事务数目越多，这是因为在存在较高冲突的情况下，查询节点在短时间内申请不到锁就不再重试，有更多的事务被放弃，在超时时间达到 3 秒时，夭折事务数目已经较少。



a.吞吐量与锁等待超时的关系　　b.响应时间与锁等待超时的关系

图 5.3　锁等待超时对更新主键吞吐量和响应时间的影响

由图 5.3 可知，锁的超时时间对系统的吞吐量和响应时间影响不大，因为超时时间越短，有更多的事务夭折，而超时时间越长，更多的事务发起重试，导致冲突可能越大，这是一个权衡取舍的问题。同时考虑到 OceanBase 的默认网络超时时间，本文设置的更新主键的锁默认超时时间为 3 秒。

5.2.3 基于 OCC-WAR 模型

（1）基本流程

基于 OCC-WAR 模型的更新主键功能在查询原行的增量数据直接读取，在事务提交前进行有效性验证，其具体的执行流程在算法 5 中进行了详细描述：

算法 5 基于 OCC-WAR 模型的更新主键算法

输入：原行主键值$K_{original}$，新行主键值表达式 F

```
1:  读取原行 original_row
2:  计算新主键值K_new := F(original_row)
3:  根据K_new和 K_original查询 new_row_static_data 和 original_row_static_data
4:  根据K_new和K_original查询 new_row_dynamic_data 和 original_row_dynamic_data
5:  if new_row_static_data + new_row_dynamic_data == NULL then
6:      插入新行(K_new和 original_row_static_data + original_row_dynamic_data)
7:      标记删除旧行 original_row
8:      if check_validation() == true then
9:          同步操作日志并刷盘
10:         commit()
11:     else
12:         rollback() and retry()
13:     end if
14: else
15:     abort()
16: end if
```

查询节点在读取原行的增量数据时，同时获取该增量数据的最近一次修改的时间戳，在写准备阶段完成后，将该时间戳和执行计划一起发送至增量节点，增量节点执行插入新行、删除原行的操作，在提交事务前增量节点检查原行的增量数据的修改时间戳是否发生变化，若两个时间戳数值不同，说明原行被其它事务修改过而失效，返回查询节点验证未通过错误；否则说明验证通过，提交事务。

（2）事务重启

若有效性验证不通过，则由查询节点发起重试操作，重新读取原行开始重启事务。查询节点可控制事务最大重启次数，在未超过最大重启次数之前，反复重启事务，直至执行成功。在相同的实验环境下，针对不同的最大重启次数，测试其与夭折事务数目、吞吐量和平均响应时间的关系，分别如图 5.4 和图 5.5 所示：

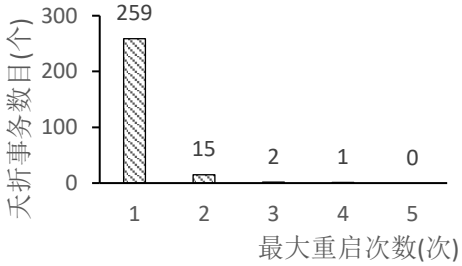
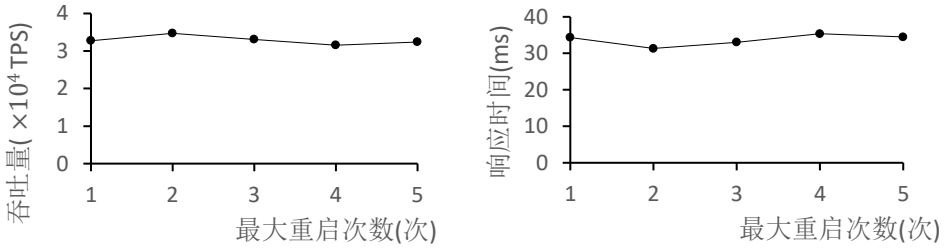


图 5.4　最大重启次数对更新主键夭折事务数目的影响

由图 5.4 可知，最大重启次数越小，夭折事务数目越多，这是因为在存在较高冲突的情况下，查询节点在有效性验证一直不通过的情况下，有更多的事务被放弃。在最大重启次数达到 3 次时，夭折事务数目已经较少。



a.吞吐量与最大重启次数的关系　　b.响应时间与最大重启次数的关系

图 5.5　最大重启次数对更新主键的吞吐量和响应时间的影响

由图 5.5 可知，最大启动次数对系统的吞吐量和响应时间影响不大，因为重启次数越少，有更多的事务夭折，而重启次数越多，更多的事务发起重试，导致冲突可能越大，这是也是一个权衡取舍的问题。综上所述，本文设置默认的最大重启次数为 3 次。

5.2.4 **分析和比较**

对于分别基于 PCC-WAR 模型和 OCC-WAR 模型的两个更新主键的方法，从理论上分析，在冲突较少的情况下，后者的性能应该优于前者，因为 PCC-WAR 模型封锁机制有一定开销，而 OCC-WAR 模型在低冲突的情况下的事务重启的代价可以忽略；在冲突较高的情况下，前者的性能应该优于后者，因为此时 PCC-WAR 模型的封锁机制开始发挥作用，但 OCC-WAR 模型的事务重启会越来越频繁，影响了事务的并发性能。

5.3 本章小结

本章详细介绍了本文提出的读写分离架构存储系统生成自增数值主键和更新主键的几种方法，分别是基于 PCC-WAR 模型生成主键、基于缓存优化和 PCC-WAR 模型生成主键、生成非连续自增主键、基于 PCC-WAR 模型更新主键、基于 OCC-WAR 模型更新主键等，分别对这些方式进行了比较分析。

第六章 实验和结果分析

本文进行了一系列的实验，用于测试读写分离的可扩展数据库中维护主键功能的正确性和性能。本章首先将介绍实验的环境设置，然后分别针对维护主键的正确性和性能两个方面进行了实验，并对实验结果进行分析和总结。

6.1 实验环境和数据集

本文设计的全部实验均在由 9 台具有相同软件和硬件配置的服务器主机组成的双集群环境中进行，这些主机的软件和硬件配置信息如表 6.1 所示。

表 6.1 服务器主机的软件和硬件配置

项目	配置说明
CPU	24 核，Intel(R) Xeon(R) CPU E5-2620
主频	2.00GHz
内存	128G
以太网	千兆网卡，BCM7719-Gigabit Ethernet
操作系统	CentOS release 6.5(Final) 64 位

本文提出的维护主键方法最终在读写分离的可扩展数据库系统 OceanBase 的开源版本(Version 0.4.2.17)上进行了实现，本章实验采用典型的主备(Master-Slave)双集群部署，其集群架构如图 6.1 所示：

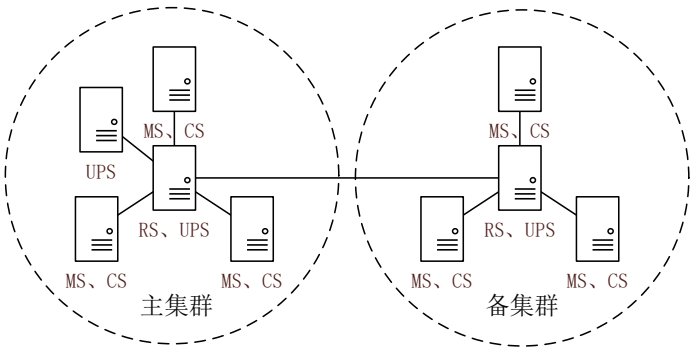


图 6.1 OceanBase 的双集群部署

在图 6.1 中，将 9 台可用的服务器分为 5 台和 4 台两组，其中前一组部署 1 个 RS、2 个 UPS、3 个 MS 和 3 个 CS 组成主集群，后一组部署 1 个 RS、1 个 UPS、3 个 MS 和 3 个 CS 组成备集群。

本章的实验分为功能测试和性能测试两个部分，因些分别使用两个不同的测试数据集。

对于功能测试，本章设计了一张较简单的测试表 test，其表结构如表 6.2 所示，初始数据中 value 字段值与主键值相同，更新操作同时更新 id 列和 value 列，可通过 value 字段的值判断该更新是否真正执行成功。测试生成主键的测试表初始数据为空，而测试更新主键的测试表初始数据为 100 万行，如表 6.3 所示。

表 6.2 功能测试数据表的表结构

字段	类型	主键	备注
id	INTEGER	是	NOT NULL
value	INTEGER	否	NOT NULL

表 6.3 功能测试的初始数据

id	value
1	1
2	2
……	……
1000000	1000000

对于性能测试，本章采用 sysbench[50]基准测试工具测试数据库事务处理性能。sysbench 是一个模块化的、跨平台、多线程的基准测试工具，可以用于测试各种不同系统参数下的数据库负载情况[50]，它可自由配置数据库连接并发数目，且模拟的表结构简单，适合测试简单查询和更新 SQL 的性能，非常符合测试主键维护的需求。sysbench 在测试数据库负载时提供默认的测试表 sbtest，其表结构均如表 6.4 所示。测试更新主键的性能时使用一张测试表，该表存储 100000 行数据。

表 6.4 性能测试数据表 sbtest 的表结构

字段名称	数据类型	主键	备注
id	INTEGER	是	NOT NULL
k	INTEGER	否	NOT NULL
c	VARCHAR(120)	否	NOT NULL
pad	VARCHAR(60)	否	NOT NULL

6.2 功能测试

主键维护最基本的功能需求是满足主键唯一性约束和单行事务性，因此，主键维护的功能测试指的验证本文实现的维护主键功能是否满足事务的 ACID 特性，即原子性、一致性、隔离性和持久性，同时验证维护主键的功能没有破坏主键唯一性约束。

6.2.1 主键唯一性约束

主键唯一性约束是数据库一致性的一部分，它要求数据表中不存在两个主键值相同的数据行，而在主键维护功能中，需要保证更新后的数据行或插入的新行不会覆盖原有的主键值相同的数据行，设计实验用多线程模拟多个用户同时向数据库发送更新请求，实验结果如表 6.5 所示：

表 6.5　主键唯一性验证的实验结果

操作	模拟用户数目	测试结果
插入新行	1	记录数与实际插入数目一致
	2	记录数与实际插入数目一致
	100	记录数与实际插入数目一致
将主键值更新为 1	1	更新失败
	2	全部失败
	100	全部失败
将主键值更新为 0	1	更新成功
	2	只有一个用户成功
	100	只有一个用户成功

对于生成主键功能，分别模拟不同数目的用户插入新数据行，并统计插入语句执行的数次，一定时间后，检查到数据库中表的总行数与插入语句的执行次数相同，说明没有数据库被覆盖，生成主键的功能满足主键唯一性的需求。

对于更新主键功能，设置了两个场景，第一个场景是模拟不同数目的用户将不同的主键值同时更新为一个表中已存在的数据行<id: 1, value: 1>的主键，此时更新的新主键值已存在，所有用户操作应该全部失败，再次查看数据原表，查看 id 为 1 的数据行的 value 字段依然为 1，说明用户可能破坏已有数据主键唯一性

约束的更新操作被拦截成功;第二个场景是模拟不同数目的用户将主键同时更新为一个表中不存在的主键值，此时，只有最先提交的事务没有主键冲突，因此，不论多少用户同时更新,只能有一个用户能更新功能,其它用户全部失败。由此，可以证明更新主键功能满足主键唯一性约束的需求。

6.2.2 事务特性

由于本文实现的维护主键功能针对的是单行的操作,因此事务性主要是指的插入一行数据或修改一行数据时的 ACID 特性。满足单行事务性的目的是在数据库出现异常的情况下，仍然保证数据的正确性，因此，设计实验直接修改源码，在主键维护执行的各个阶段，精确模拟异常场景，来验证单行事务性。

（1）原子性

原子性指维护主键过程中所有操作要么全部执行，要么全部不执行。设计实验：实验前，查询一次基准值和目标行的状态；实验中，在维护主键的执行过程的各个阶段模拟异常，主动取消操作；实验后，再次查询数据库基准值和目标行的状态。实验设置和结果如表 6.6 所示：

表 6.6 主键维护的原子性验证的实验结果

功能	取消时机	基准值状态			目标行状态		
		基准值	初始状态	最终状态	目标行	初始状态	最终状态
生成主键	插入新行前	最大主键值	1	1	新行	不存在	不存在
	更新最大主键前	最大主键值	1	1	新行	不存在	不存在
	事务提交前	最大主键值	1	1	新行	不存在	不存在
更新主键	插入新行前	原行	存在	存在	新行	不存在	不存在
	删除旧行前	原行	存在	存在	新行	不存在	不存在
	事务提交前	原行	存在	存在	新行	不存在	不存在

由上表可知，无论是更新主键，还是生成主键，分别在更新目标值前、更新

基准值前和事务提前取消操作，都会引起整个事务的回滚，基准值和目标行的状态不会改变，因此，维护主键功能满足原子性要求。

（2）隔离性

隔离性指并发事务之间不会相互影响。若事务在提交前对数据修改没有应用到数据库中，可保证事务之间是隔离的。设计实验：通过客户端 A 数据库，执行一个包含主键维护操作的长事务，在该事务提交前，在客户端 B 中查询数据表，验证未提交的事务对数据的修改是否可见。验证生成主键和更新主键隔离性的实验设置和结果分别如表 6.8 和表 6.8 所示：

表 6.7　生成主键的隔离性验证的实验结果

顺序	客户端 A	客户端 B	状态
1	SELECT * FROM test WHERE id=1		不存在
3	BEING		
4	INSERT INTO test (value) VALUES (1)		执行成功
5		SELECT * FROM test WHERE id=1	不存在
6	COMMIT		
8		SELECT * FROM test WHERE id=0	存在

由表 6.8 可知，在生成主键事务提交前，它中间状态对其它事务是不可见的，对数据库的修改操作只有在事务提交后才生效。因此，生成主键的功能满足隔离性要求。

表 6.8 更新主键的隔离性验证的实验结果

顺序	客户端 A	客户端 B	状态
1	SELECT * FROM test WHERE id=0		不存在
2	SELECT * FROM test WHERE id=1		存在
3	BEING		
4	UPDATE test SET id=0 WHERE id=1		执行成功
5		SELECT * FROM test WHERE id=0	不存在
6		SELECT * FROM test WHERE id=1	存在
7	COMMIT		
8		SELECT * FROM test WHERE id=0	存在
9		SELECT * FROM test WHERE id=1	不存在

由表 6.8 可知，在更新主键的事务提交前，它中间状态对其它事务是不可见的，对数据库的修改操作只有在事务提交后才生效。因此，更新主键的功能满足隔离性要求。

（3）一致性

一致性主要是指事务的执行满足数据完整性约束，主要是指主键唯一性约束条件，在 6.2.1 节已做验证。

（4）持久性

持久性是指事务一旦提交对数据库的影响是持久的。OceanBase 的高可用架构保证主增量节点宕机的情况下，备增量节点可自动接替主的工作。因此，设计实验执行一次维护主键的操作，分别在该事务提交前和提交后模拟主机宕机，再次查看数据库更新操作的结果是否存在，验证结果如表 6.9 所示。

表 6.9　主键维护的持久性验证的实验结果

操作	主增量节点宕机时机	查询数据库更新是否生效
生成主键	事务提交前	未生效
	事务提交后	已生效
更新主键	事务提交前	未生效
	事务提交后	已生效

由上表可知，主键维护事务提交后持久化到数据库中，对数据库产生的影响不因系统故障而丢失，符合持久性要求。

综上所述，本文在可扩展数据库系统中实现的维护主键的功能可保证单行事务性，且没有破坏主键唯一性约束。

6.3 性能测试

6.3.1 性能指标

维护主键是数据库 SQL 功能的一部分，因此可以使用 SQL 语句执行速度来描述维护主键功能的性能。通常使用系统吞吐量(Throughput)和用户平均响应时间(Average Response Time)来表示数据库管理系统处理 SQL 语句的性能。一般而言，吞吐量最大、平均响应时间越小，说明系统的性能越高。

（1）吞吐量

吞吐量指的是一定时间内数据库系统能够完成的读写请求总数，即单元时间内可完成的任务数，一般使用 TPS(Transaction per Second)或 QPS(Query per Second)表示数据库系统的吞吐量。TPS 或 QPS 值越高，说明数据库系统的吞吐量越大。

$$\text{吞吐量} = \frac{\text{系统处理请求总数}}{\text{完成处理的总时间}}$$

本实验采用 TPS 表示数据库系统更新主键功能的吞吐量。

（2）响应时间

响应时间是指用户从向数据库系统发出读写请求开始，到获得处理结果所需要的时间，即对请求作出响应所需要的时间，一般以毫秒(ms)或秒(s)表示。响应

时间包括网络传输时间和系统处理时间。

响应时间 = 网络传输时间 + 数据库系统处理时间

本实验采用 95%的语句的平均响应时间表示更新主键功能的平均响应时间，并以 ms 为单位。

6.3.2 生成主键的性能

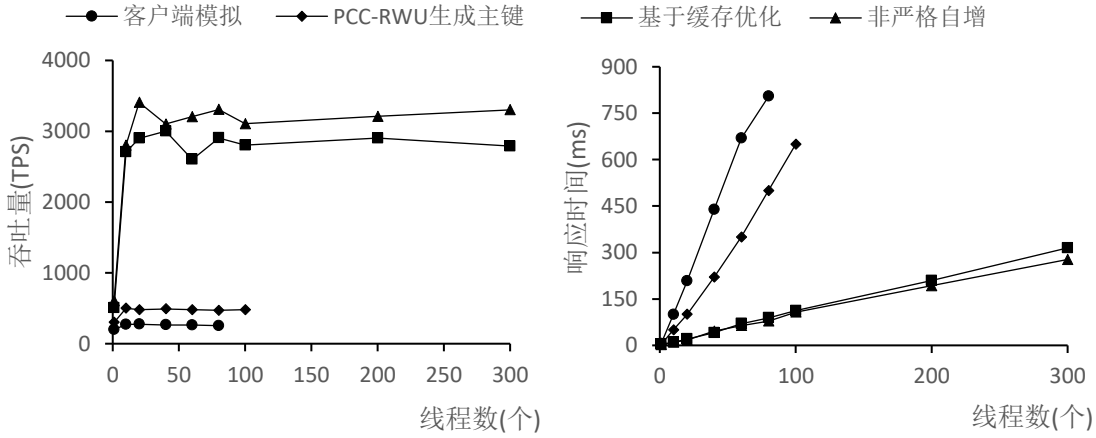
由于生成主键时需要对同一个基准值进行查询和更新操作，当多个线程同时往某张表格中插入数据时，必然产生很大的冲突，因此，不适合 OCC-PWU 模型。本文实现了基于 PCC-WAR 模型生成主键的功能，并在此基础上进行了基于缓存的优化，同时实验了基于缓存的非严格自增生成主键的功能，最后对比客户端模拟生成主键测试和对比这些实现方法的性能。

和客户端模拟更新主键方法类似，客户端模拟生成主键也是通过数据库事务功能封装一组操作，如 **SQL 2** 所示：

SQL 2 模拟生成主键

```
1. BEGIN
2. SELECT value FROM max_id WHERE table_id=1 FOR UPDATE
3. INSERT INTO sbtest (id, k, c, pad) VALUES (<id>, <k>, <c>, <pad>)
4. UPDATE max_id SET value=value+1 where table_id=1
5. COMMIT
```

在相同的实验环境中，分别测试客户端模拟生成主键、基于 PCC-WAR 模型生成主键、基于缓存优化和 PCC-WAR 模型生成主键和基于缓存优化非严格自增生成主键四个方法不同线程生成主键的吞吐量的响应时间，其中“客户端模拟”和“基于 PCC-WAR 模型”两种方式生成主键冲突过大且单个事务响应时间过长，在线程数分别达到 80 和 100 之后，由于过多事务超时导致 sysbench 断开连接而中断测试，因此没有数据。实验结果如图 6.2 所示：



a.生成主键的吞吐量　　b.生成主键的响应时间

图 6.2　生成主键的吞吐量和响应时间

由上述折线图可知，相对于客户端实现生成主键，基本的 PCC-WAR 模型生成主键对性能有一定的提升，但提升的幅度有限，大约只提升了的 1.5 倍，这是因为在高冲突情况下查询最大主键值和基线数据耗时过长，加剧了事务的冲突，从而生成主键事务的总体响应时间过长，导致过多的事务执行超时。而基于缓存优化的方案对性能提升了大约 6 倍，这是因为增加缓存后省略了最大主键和新行查询基线数据的过程，大幅度地减少了不必要的查询耗时，缓解了事务的冲突。基于缓存优化的非严格自增生成主键的方式对性能的提升更大，因为该方式提前释放了最大主键的锁，进一步地缓解了事务的冲突。但是由于生成主键功能的特殊性，多个线程不可避免地需要竞争最大主键值这一公共资源，而且线程越多，竞争越激烈，在多线程情况下，以上多种方式实现的生成主键的功能很快就达到了吞吐量的上限，线程数越多不但对吞吐量的增加没有帮助，而且反而会延长响应时间，加剧事务的冲突。

综上所述，本文实现的基于缓存的优化方案的方式很好地提升了生成主键的性能。

6.3.3 **更新主键的性能**

基于 PCC-WAR 模型和 OCC-WAR 模型的更新主键功能使用不同的并发处理机制，在不同的读写冲突情况下的性能表现也不同，因此，分别实验模拟低冲突和高冲突两种场景进行测试。sysbench 可以模拟对简单表格的单一 SQL 操作，

本实验设计了简单的单行更新语句 **SQL 3** 来测试更新主键功能的性能：

SQL 3 直接更新主键

```
1. UPDATE sbtest SET id=id WHERE id=<id>
```

上述 SQL 语句中设置“id=id”，这是为了避免高并发情况下主键重复错误的影响，将主键值更新为原主键值，该设置保证了每条 SQL 语句无论执行多少次，都不会产生主键重复的错误。

为了验证本文实现的更新主键的功能的高效性，在相同的实现环境中同时也测试了更新主普通列和客户端模拟更新主键的性能。更新主普通列指更新某行的非主键列的值，这是一般可扩展数据库都支持的功能；而客户端模拟是指使用长事务间接地实现更新主键行的功能，本实验设计一组 SQL 语句可达到修改主键列值的目的，如 **SQL 4** 所示：

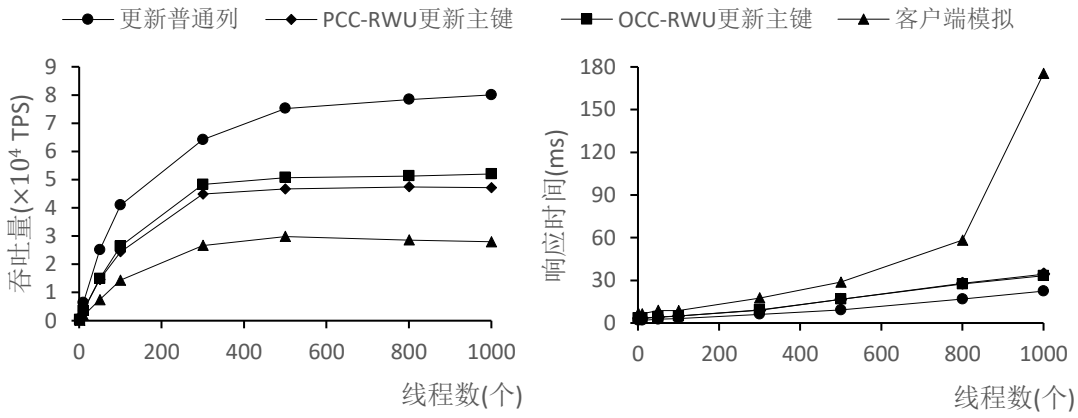
SQL 4 模拟更新主键

```
1. BEGIN
2. SELECT id, k, c, pad FROM sbtest WHERE id=<id> FOR UPDATE
3. DELETE FROM sbtest WHERE id=<id>
4. INSERT INTO sbtest (id, k, c, pad) VALUES (<id>, <k>, <c>, <pad>)
5. COMMIT
```

（1）低冲突情况

在 sysbench 中控制每个线程更新的数据行都不一样，可模拟一种低冲突的场景，该场景是低冲突的一种极端情况，即所有并发的事务都不存在冲突，可以用于测试更新主键功能的性能上限。作为对比，在该场景下同时也测试了更新普通列和在客户端模拟更新主键列的性能。

不同的连接线程的情况下，分别测试更新普通列、PCC-WAR 更新主键、OCC-WAR 更新主键和客户端模拟更新主键的性能，在低冲突场景中，它们的吞吐量和平均响应时间随线程数变化的曲线分别如图 6.3 所示。



a.低冲突时更新主键的吞吐量　　b.低冲突时更新主键的响应时间

图 6.3　低冲突时更新主键的吞吐量和响应时间

由图 6.3 可知，在当前实验环境中，本文实现的更新主键功能大约是更新普通列的 0.6 倍，以及客户端模拟更新主键的 2 倍；更新主键功能的平均响应时间略高于更新普通列的平均响应时间，并低于客户端模拟的平均响应时间，但随着并发线程数的增加，客户端模拟的平均响应时间快速增涨，远高于更新主键和更新普通列的平时响应时间。

由该实验数据也能看出，在低冲突的情况下，基于 PCC-WAR 和 OCC-WAR 两种模型的更新主键功能的性能相差不多，OCC-WAR 更新主键的性能略高于 PCC-WAR 更新主键。通过分析可知，OceanBase 的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，没有传统关系数据库的锁表维护开销，所以锁管理的代价非常低，对锁的申请和释放对整体的系统影响不大，因此低冲突情况下，相较于 OCC-WAR 更新主键，基于封锁协议的 PCC-WAR 更新主键方法仍表现出较好的性能。同时，本文实现的更新主键的功能性能大约是更新普通列性能的一半，也是客户端通过长事务间接实现更新主键功能的两倍，普通普通列包含一个更新操作，更新主键还包括一个查询操作，而长事务的方式包含了一个查询操作和两个更新操作，其性能理论上是 3:2:1 的关系，实验结果比较符合理论预期。

（2）高冲突情况

在低冲突的情况下，PCC-WAR 和 OCC-WAR 更新主键的性能差异不大，因

此设置高冲突的场景来测试和比较 PCC-WAR 更新主键和 OCC-WAR 更新主键的性能。

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，本次实验采用“1:1”的比例进行测试，即 n 个线程同时随机更新 n 行数据，这是一种冲突较高的场景。在不同的连接线程的情况下，分别测试 PCC-WAR 更新主键和 OCC-WAR 更新主键的性能，其吞吐量和平均响应时间随线程数变化的曲线分别图 6.4 所示。

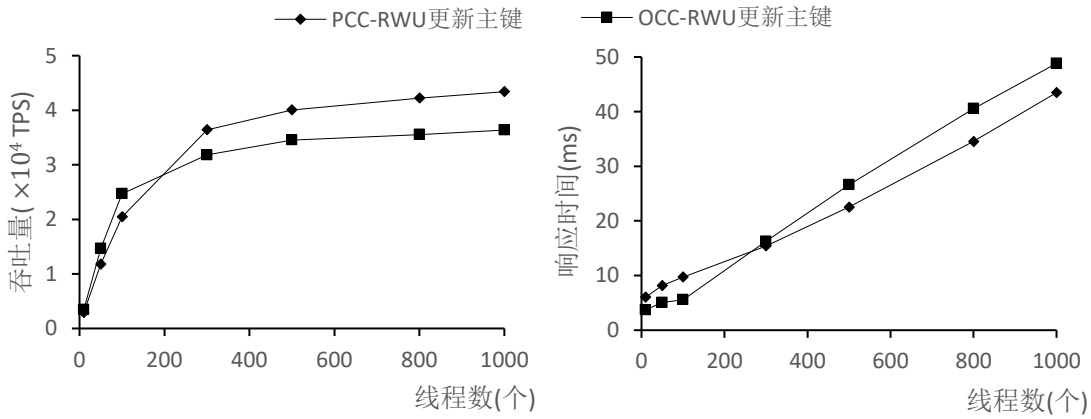


图 6.4　高冲突时更新主键的吞吐量和响应时间

由图 6.4 可知，在存在冲突的情况下，PCC-WAR 和 OCC-WAR 更新主键的性能均有所下降。具体的，在线程数较少的情况下，冲突相对较低，OCC-WAR 更新主键的性能高于 PCC-WAR 更新主键，而随着线程数的增加，冲突相对较高，OCC-WAR 更新主键事务重启开始增多，性能下降较大，因此 PCC-WAR 更新主键的性能优于 OCC-WAR 更新主键。

6.4 本章小结

本章设置了一系列实验，充分验证本文实现的读写分离的可扩展数据库系统中维护主键功能的正确性和高效性。其中维护主键的正确性包括满足事务性和主键惟一性约束两个方面。性能测试表明，本文实现的基于 PCC-WAR 模型和 OCC－WAR 模型的更新主键功能的性能是客户端通过长事务间接实现更新主键功能

的 2 倍，基于 PCC-WAR 模型生成主键功能的性能是客户端通过长事务间接实现生成主键功能的 1.5 倍，而基于缓存优化后的生成主键功能的性能进一步提升为原来的 6 倍。因此，本文成功地在可扩展数据库中实现了高效的主键维护的功能。

第七章 总结与展望

7.1 总结

近十年的互联网的格局发生了翻天覆地的变化，各种新型分布式数据库不断面世，基于读写分离的可扩展数据库管理系统便是其中之一。读写分离技术通过将数据划分为更新数据和基线数据，并且分别存在于不同的服务器节点中，从而将数据库的读和写操作分离到不同的节点执行。基于读写分离的可扩展数据库能够很好地解决海量数据存储和管理的问题，因而受到了越来越多的关注。

目前，基于读写分离的可扩展数据库系统对数据库功能的支持程度不高，特别是对主键的维护和管理相关的功能，很难满足应用系统的需求。本文针对基于读写分离的可扩展数据库中的主键维护功能，提出一类通用的先读后写操作的模型，作为主键维护功能的一种抽象描述，然后将其应用于主键维护功能的设计与实现中。本文的工作内容总结如下：

(1) 本文总结了一个先读后写 (WAR) 操作的模式，WAR 包含了读取和写入两类操作，即在更新数据前首先向数据库读取一次或多次数据，作为写入操作的输入，WAR 操作可以看作是数据库长事务的一个特例。本文分别基于悲观并发控制协议和乐观并发控制协议提出了 PCC-WAR 模型和 OCC-WAR 模型，它们适用于不同的场景，其中基于 PCC-WAR 模型适合读写冲突较大的场景，而 OCC-WAR 模型适合读写冲突较少的场景。进一步的，本文针对某些特殊场景对 WAR 框架提出了基于缓存的优化方案，该方案的基本思想是将数据缓存在增量数据中，避免了读取基线数据的过程，可以显著的提升 WAR 操作的性能。WAR 框架具有较好的通用性，可以应用于某些通用数据库功能的实现中，进而丰富新型的可扩展数据库系统的数据库功能。

(2) 本文提出了在基于读写分离架构的可扩展数据库中实现主键维护的功能

的多种方法。主键维护包括了主键生成和主键更新两个子功能，其中主键更新是指修改某个数据行的主键值，而主键生成是指在插入新数据行时系统自动分配一个惟一且自增的数值型主键。主键维护功能可以通过数据库的长事务功能封装多个读写操作间接地实现，但是这种外部的实现方法往往需要改造应用程序，增加应用程序的复杂度，且性能较差。针对这些问题，本文基于 WAR 的模型提出了在数据库系统内部直接支持主键维护功能的多种方法，这些方法可以保证主键更新和生成操作的事务特性，以及不破坏主键惟一性约束，并且可以实现较高的性能。

表 7.1　主键维护功能不同的实现方法的性能对比

功能	实现方法	性能（TPS 倍数）
主键生成	外部间接实现	1
	基于 PCC-WAR 模型	1.5
	基于缓存优化的 PCC-WAR 模型	6
主键更新	外部间接实现	1
	基于 PCC-WAR 模型	1.8
	基于 OCC-WAR 模型	1.9

(3) 本文在开源的可扩展关系数据库 OceanBase（版本 0.4.2.17）中实现了主键维护功能，并设计了一系列的功能测试和性能测试来验证主键维护功能的正确性和高性能。其中功能测试分别从主键惟一性约束和事务特性两个方面来验证维护主键功能的正确性，而性能测试的实验结果如表 7.1 所示，该结果表明在低冲突的情况下，基于 PCC-WAR 模型和 OCC-WAR 模型更新主键的性能分别是长事务间接更新主键的 1.8 倍和 1.9 倍，其中 PCC-WAR 模型的方法只是略低于 OCC-WAR 模型，这是因为在可扩展数据库中封锁的维护开销不大。基于 PCC-WAR 模型生成主键是长事务间接生成主键的 1.5 倍，而基于缓存优化的生成主键的性能提升较大，达到长事务间接生成主键的 6 倍。这些实验结果都证明了本文实现的维护主键功的高效性。

综上所述，本文提出的主键维护功能的实现方法在可扩展数据库管理系统中

得到了成功应用，满足正确性和性能要求，同时，本文提出先读后写操作的抽象模型，可为其它数据库的研究提供借鉴。

7.2 展望

本文对基于读写分离架构并且采用单节点写入模式的可扩展数据库中先读后写操作进行了研究，提出了在该场景中实现并发控制的方法，由于时间和能力有限，还存在一些问题没有完全解决，今后可以进一步研究的内容包括以下三个方面：

1) **数据库功能扩展**。本文提出的 PCC-WAR 模型和 OCC-WAR 模型，以及基于了缓存的优化方案具有通用性，但是在本文中只将其应用于维护主键功能的设计与实现，可将其应用于其它类似的数据库功能的实现中，例如 SELECT INTO 和带子查询的 UPDATE 等功能。

2) **针对不同数据库进行优化**。本文提出的维护主键功能的实现方案最终只在数据库 OceanBase 中进行了实现，而可扩展的数据库管理系统种类较多，其具体实现细节不同，因此可以对不同的数据库中的实现进行优化，比如锁的冲突调度和有效性验证方法实现等。

3) **多节点写入模式的扩展**。本文讨论的可扩展数据库管理系统使用的是单节点写入模型，对于多节点写入模型，由于涉及到分布式事务，需要引入分布式协议，情况较为复杂，多节点写入模型中的 WAR 模型值得进一步研究。

随着互联网大数据应用的进一步发展，基于读写分离技术的分布式可扩展数据库将会有更广阔的发展前景，应用系统也将对可扩展数据库的数据库功能提出越来越多的需求，本文提出的 PCC-WAR 模型和 OCC-WAR 模型可以广泛地应用到数据库功能完善的实践中去。

参考文献

[1] Bachman C W. The programmer as navigator[J]. Communications of the A CM, 1973, 16(11): 653-658.

[2] Kamfonas M. Recursive hierarchies: the relational taboo[J]. The Relational Journal, 1992, 27(10).

[3] Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377-387.

[4] Microsoft. Microsoft SQL Server. http://www.microsoft.com/zh-cn/server-cloud/products/sql-server/. [Online; accessed 12-August-2016].

[5] Oracle Corporation. Oracle. https://www.oracle.com/database/index.html. [Online; accessed 12-August-2016].

[6] IBM. DB2. https://www.ibm.com/analytics/cn/zh/technology/db2/. [Online; accessed 12-August-2016].

[7] Menon J, Pease D, Rees R. Distributed storage system for data-sharing among client computers running defferent operating system types: U.S. Patent Application 10/323,113[P]. 2002-12-18.

[8] Stonebraker M. SQL databases v. NoSQL databases[J]. Communications of the ACM, 2010, 53(4): 10-11.

[9] Amazon. Amazon SimpleDB. https://aws.amazon.com/cn/simpledb/. [Online; accessed 13-August-2016].

[10] Dirolf M, Chodorow K. MongoDB: the definitive guide[M]. O'Reilly, 2010.

[11] Redislabs. Redis. http://redis.io/. [Online; accessed 13-August-2016].

[12] Memcached. http://memcached.org/. [Online; accessed 14-August-2016].

[13] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed structured data storage system[C]//7th OSDI. 2006, 26: 305-314.

[14] George L. HBase: the definitive guide[M]. " O'Reilly Media, Inc.", 2011.

[15] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally distributed database[J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.

[16] Microsoft. Microsoft Azure. https://azure.microsoft.com/zh-cn/. [Online; accessed 13-August-2016].

[17] VoltDB. VoltDB. https://www.voltdb.com. [Online; accessed 13-August-2016].

[18] Alibaba. OceanBase. https://github.com/alibaba/oceanbase. [Online; accessed 13-August-2016].

[19] Buch V, Cheevers S. Database architecture: Federated vs. clustered[J]. http://www.oracle.com/technetwork/database/windows/clustercomp-134873.pdf.Oracle, February, 2002.

[20] 简朝阳. MySQL 性能调优与架构设计[M]. 电子工业出版社, 2009.

[21] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[22] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOC-S), 1992, 10(1): 26-52.

[23] Al-Houmaily Y J, Samaras G. Two-phase commit[M]//Encyclopedia of Database Systems. Springer US, 2009: 3204-3209.

[24] Lamport L. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.

[25] 阳振坤. OceanBase 关系数据库架构[J]. 华东师范大学学报: 自然科学版, 2014 (5): 141-148.

[26] 黄贵, 庄明强. OceanBase 分布式存储引擎[J]. 华东师范大学学报: 自然科

学版, 2014 (5): 164-172.

[27] 李凯, 韩富晟. OceanBase 内存事务引擎[J]. 华东师范大学学报: 自然科学版, 2014 (5): 149-163.

[28] Gray J, Reuter A. Transaction processing: concepts and techniques[M]. Elsevier, 1992.

[29] The Open Group. Distributed TP: The XA Specification. Free PDF-https://www2.opengroup.org/ogsys/catalog/C193, 1992.

[30] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom. 数据库系统实现[M]. 机械工业出版社, 2010: 204-205.

[31] 董钢. SQL Server 数据库封锁及死锁研究[J]. 计算机与现代化, 2005 (8): 52-55.

[32] Eswaran K P, Gray J N, Lorie R A, et al. The notions of consistency and predicate locks in a database system[J]. Communications of the ACM, 1976, 19(11): 624-633.

[33] Holt R C. Some deadlock properties of computer systems[J]. ACM Computing Surveys (CSUR), 1972, 4(3): 179-196.

[34] Bernstein P A, Goodman N. Timestamp-based algorithms for concurrency control in distributed database systems[C]//Proceedings of the sixth internati-onal conference on Very Large Data Bases-Volume 6. VLDB Endo-wment, 1980: 285-300.

[35] Shashi B, Patel R B, Mayank D. A Secure Time-Stamp Based Concurrency Control Protocol for Distributed Databases[J]. Journal of Computer Science, 2007, 3(7): 561-565.

[36] Ammann P, Jajodia S. A timestamp ordering algorithm for secure, single-version, multi-level databases[C]//Results of the IFIP WG 11.3 Workshop on Database Security V: Status and Prospects. North-Holland Publishing Co., 1

991: 191-202.

[37] Kung H T, Robinson J T. On optimistic methods for concurrency control [J]. ACM Transactions on Database Systems (TODS), 1981, 6(2): 213-226.

[38] 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. Springer Berlin Heidelberg, 2006: 796-805.

[39] Silberschatz A, Korth H F, Sudarshan S. Database system concepts[M]. New York: McGraw-Hill, 1997.

[40] Cormen T H, Leiserson C E, Rivest R L. 算法导论 (原书第 3 版) [M]. 机械工业出版社, 2013:142-150.

[41] 杨传辉.大规模分布式存储系统原理解析与架构实现[M].机械工业出版社, 2013:12-17.

[42] Chang S K, Cheng W H. A methodology for structured database decomposition[J]. IEEE Transactions on Software Engineering, 1980 (2): 205-218.

[43] Chang S K, Liu A C. File allocation in a distributed database[J]. International journal of computer & information sciences, 1982, 11(5): 325-340.

[44] Oracle Corporation and/or its affiliates. MySQL. https://www.mysql.com/. [Online; accessed 13-August-2016].

[45] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/. [Online; accessed 13-August-2016].

[46] Melton J. ANSI x3. 135-1992, American national standard, database language SQL[J]. American National Standards Institute, 1992.

[47] Gray J. The transaction concept: Virtues and limitations[C]//VLDB. 1981, 81: 144-154.

[48] Hofri M. On timeout for global deadlock detection in decentralized databa-

se systems[J]. Information Processing Letters, 1994, 51(6): 295-302.

[49] JL· 亨尼西 (美), Hennessy J L, DA· 帕特森 (美), 等. 计算机体系结构: 量化研究方法: 英文版·[M]. 机械工业出版社, 2003:44-45.

[50] Kopytov A. SysBench: a system performance benchmark[J]. URL: https://github.com/akopytov/sysbench, 2016.

致　谢

转眼间，两年半的研究生和学习生活马上就要结束了，而入学典礼仿佛就发生在昨天，那年夏天，初来华师的兴奋激动的场景犹历历在目。回忆起这两年多的点点滴滴，百感交集，非常庆幸自己来到了一个很好的学习环境，并且遇到了一群志同道合的同学和老师，也很欣慰自己一直没有松懈，在老师和同学的激励、指引和帮助下能够顺利地完成学业。

首先，最深的谢意献给我的导师周敏奇老师，周敏奇老师既平易近人又严谨治学，被同学们和其他老师亲切地称为小周老师，从入学前的第一次邮件沟通接触，到后来的相处，小周老师一直悉心引导我的学习和科研工作，给了我很多的关心和帮助。

同时，我要感谢钱卫宁教授在毕业论文上给予我的指导和帮助，他渊博的专业知识和严谨的工作风格，深深地熏陶着我。从论文成型到撰写完成，都离不开钱老师的全心指导，帮助我顺利地完成了毕业论文，让我感激不已。还要感谢课题组的宫学庆教授、张蓉教授、高明老师、张召老师、蔡鹏老师、胡卉芃老师、翁楚良老师，以及研究所里的王晓玲老师和金澈清老师，他们都在我两年多的研究生学习阶段给予了我很大的帮助。

其次，我要感谢实验室的郭进伟博士和朱涛博士，不论是平时在实验室的科研工作，还是本论文的实验和写作，他们都给予了我很多帮助，对本论文提出了许多有益的建议和意见。另外，我要感谢晁平复学长、高祎璠学姐、方祝和、于楷等同学在我入学之初对我生活和学习上的帮助，让我很快的适应了研究生的新生活；感谢课题组的周欢博士、张晨东学长、庞天泽学长、刘骁学长、李永峰学长和樊秋实学长对我科研和实验上的解惑；感谢实验室里同时入学的李宇明、钱招明、王雷、熊辉、余晟隽、祝君、周楠等同学一起对科研项目做出的努力，感谢储佳佳、王嘉豪、肖冰、张燕飞、张春熙、朱燕超、金天阳、黄建伟等同学在一起合作完成工作时对我的支持和帮助，其中要特别要感谢储佳佳对本文提出了

许多宝贵的修改意见，无私地帮助我完善论文。感谢实验室的所有同学创造了一个欢乐轻松实验室氛围，让我留下了许多美好的回忆。

最后，我要感谢我的父母，感谢我的哥哥嫂子，能够走到今天离不开他们的支持和鼓励，我希望能够顺利地完成学业，争取更大的成功，给他们一点欣慰。还要感谢华东师范大学，她在我人生中最无助最迷茫的时候宽容地接纳了我，给我的人生添上了浓重的一笔，让我有了一个更新更高的起点，能够更加自信地迎接未来的挑战！

谨以此文献向所有关注过和关注着我的人表示最衷心的感谢！

刘柏众

二零一六年十月十日

发表论文和科研情况

■ 已发表或录用的论文

[1] 祝君, **刘柏众**, 余晟隽, 等. 面向 OceanBase 的存储过程设计与实现[J] . 华东师范大学学报: 自然科学版, 2016 (5): 144-152.

[2] 储佳佳, 郭进伟, **刘柏众**, 等. 高可用数据库系统中的分布式一致性协议[J]. 华东师范大学学报: 自然科学版, 2016 (5): 1-9.

[3] 张晨东, 郭进伟, **刘柏众**, 等. 基于 Raft 一致性协议的高可用性实现[J]. 华东师范大学学报: 自然科学版, 2015 (5): 172-184.

■ 参与的科研课题

[1] 国家自然科学基金重点项目（U1401256），2014-2018，参与人

[2] 国家高技术研究发展计划（863 计划）课题，基于内存计算的数据管理系统研究与开发（2015AA015307），2015-2017，参与人