

本科生毕业论文

|  |  |
| --- | --- |
| 题目： | **一种在属性图数据库中实现事务的设计** |
|  | **A Design for Implementing Transactions in Property Graph Database** |

|  |  |
| --- | --- |
| 姓 名： | 宋怡馨 |
| 学 号： | 1800011747 |
| 院 系： | 信息科学技术学院 |
| 专 业： | 数据科学与大数据技术 |
| 导师姓名： | 陈向群 |

二〇二三年五月

北京大学本科毕业论文导师评阅表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 学生姓名 | 宋怡馨 | | 本科院系 | 信息科学技术学院 | 论文成绩（等级制） |  |
| 学生学号 | 1800011747 | | 本科专业 | 数据科学与大数据技术 |
| 导师姓名 | 陈向群 | | 导师单位/  所在学院 | 计算机学院 | 导师职称 | 教授 |
| 论文题目 | | 中文 | 一种在属性图数据库中实现事务的设计 | | | |
| 英文 | A Design for Implementing Transactions in Property Graph Database | | | |
| 导师评语  （包含对论文的性质、难度、分量、综合训练等是否符合培养目标的目的等评价）  导师签名：  年 月 日 | | | | | | |

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

**摘要**

关键词：关键词1，关键词2，关键词3……

**ABSTRACT**

[This is for abstract]

KEY WORDS: [Keyword 1, Keyword 2, Keyword 3, ……]

**目 录**

[第一章 引言 1](#_Toc134223255)

[1.1 研究背景与意义 1](#_Toc134223256)

[1.2 研究现状 2](#_Toc134223257)

[1.3 研究内容 3](#_Toc134223258)

[1.4 论文结构 4](#_Toc134223259)

[第二章 相关工作 6](#_Toc134223260)

[2.1 图数据库系统 6](#_Toc134223261)

[2.2 数据库事务 8](#_Toc134223262)

[2.3 多版本并发控制 10](#_Toc134223263)

[2.4 分布式ID 12](#_Toc134223264)

[2.5 分布式时钟 13](#_Toc134223265)

[2.6 Nebula Graph系统架构 14](#_Toc134223266)

[第三章 多版本并发控制实现 17](#_Toc134223267)

[3.1 整体结构 17](#_Toc134223268)

[3.2 多版本并发控制协议选择 18](#_Toc134223269)

[3.3 属性的多版本存储实现 20](#_Toc134223270)

[3.4 点-边关系的多版本存储实现 21](#_Toc134223271)

[3.5 可串行化隔离级别实现 22](#_Toc134223272)

[3.6 垃圾回收算法 23](#_Toc134223273)

[3.7 操作流程 24](#_Toc134223274)

[3.6.1 TX Manager接口总览 24](#_Toc134223275)

[3.6.2 开启事务 24](#_Toc134223276)

[3.6.3 读操作 25](#_Toc134223277)

[3.6.4 新建操作 26](#_Toc134223278)

[3.6.5 删除操作 27](#_Toc134223279)

[3.6.6 更新操作 27](#_Toc134223280)

[3.6.7 提交事务 28](#_Toc134223281)

[3.6.8 回滚事务 28](#_Toc134223282)

[第四章 持久化设计 31](#_Toc134223283)

[4.1 LRM系统设计 31](#_Toc134223284)

[4.2 Index Map持久化写入 31](#_Toc134223285)

[4.3 VE Map持久化写入 34](#_Toc134223286)

[4.4 重启恢复 34](#_Toc134223287)

[4.5 AOF重写 35](#_Toc134223288)

[第五章 实验与分析 37](#_Toc134223289)

[5.1 实验环境 38](#_Toc134223290)

[5.2 第一部分 38](#_Toc134223291)

[5.2.1 单事务CRUD 38](#_Toc134223292)

[5.2.2 多事务CRUD+RC 39](#_Toc134223293)

[5.2.3 多事务CRUD+RR 40](#_Toc134223294)

[5.2.4 事务回滚 41](#_Toc134223295)

[5.2.5 时序验证 45](#_Toc134223296)

[5.2.6 顶点锁 47](#_Toc134223297)

[5.3 第二部分 48](#_Toc134223298)

[5.3.1 Index AOF写入与读取 48](#_Toc134223299)

[5.3.2 VE AOF写入与读取 49](#_Toc134223300)

[5.3.3 AOF重写 50](#_Toc134223301)

[5.3.4 GC 52](#_Toc134223302)

[5.4 第三部分 53](#_Toc134223303)

[5.4.1 并发异常现象 53](#_Toc134223304)

[5.4.2 nGQL 56](#_Toc134223305)

[第六章 结论 58](#_Toc134223306)

[6.1 总结 58](#_Toc134223307)

[6.2 不足与展望 58](#_Toc134223308)

[参考文献 60](#_Toc134223309)

[附录 A 部分系统设计代码 62](#_Toc134223310)

[致谢 63](#_Toc134223311)

[北京大学学位论文原创性声明和使用授权说明 66](#_Toc134223312)

# 引言

## 研究背景与意义

互联网的高速发展产生了海量的数据，越来越丰富的应用场景也产生了多样化的数据，其中半结构化和非结构化的数据增长量可观，而传统的关系型数据库并不擅长存储和处理这两类数据。同时，现代社交关系网络以及衍生领域的发展，也使得数据间的关系的地位越来越重要，在此基础上图作为一种数据结构，衍生的图分析、图处理、图计算和图数据库系统也就获得了学术界和工业界的共同关注。

图是一种近十余年来在计算机以及数据科学等领域处于热门状态的数据结构，具有独特的机制，其中图的遍历以及对图的连结性的相关研究较多，由此诞生了图机器学习、图遍历算法以及图数据存储等分支领域。

对图数据的研究主要分为图计算以及图存储两大领域，其中图计算领域是将一种研究人类世界中的图的技术，会对它们进行描述、表征、分析以及计算，典型代表为图算法，例如搜索算法、寻路算法、图匹配算法、连通分支、子图、网络流等等，以及图神经网络和知识图谱等[1]。图存储领域更关注相连的数据在底层存储上的实现，诞生了图数据库系统，图数据库系统是指实现了OLTP和/或OLAP的存储系统，OLTP一般用于支持联机系统中的事务任务，OLAP一般用于执行复杂的分析和计算任务[2][3]。由于图数据库系统的独特关系存储机制，在学术界以及工业界都有热切的关注与应用。同时，由于图数据库是一个较新的领域，业界尚未建立统一的标准，系统发展存在诸多分支，因此也使得这一领域还有许多可以探索和实践的部分。

图数据库在现实中有广泛的应用领域，包括社交网络、推荐系统引擎、金融欺诈检查以及各类需要存储海量关系模型或需要执行深度关系查询的场景。不同于关系型数据库使用的关系模型，图数据库系统不需要预先定义关系模式（Schema），因此十分擅长存储非结构化数据。同时，图的顶点和边天然对应了数据的实体和关系这两个概念，因此用图来表征数据并在此基础上进行计算和分析操作是非常自然的——不需要像关系性数据库一样将数据的关系转换为外键，跨越多张表进行查询和就按——因此对于用户编写代码以及底层处理数据都是更方便的。

数据库事务是一个古老且持久的概念，能够在多种程度上维持数据一致性，是许多重要操作的底层基础。如今，随着互联网实时访问流量的增大以及全球化分布部署的特点愈发明显，对数据库事务的并发要求和可用性要求也越来越高，因此出现了包括写时复制（COW）、读-复制-更新（RCU）以及多版本并发控制（MVCC）等实现，其目的是让只读事务能够更快且尽可能不受加锁限制地获取数据，在读多写少场景十分有用[4][5]。

数据库事务的概念来源于交易，所以在与金钱有关的场景下，我们必须使用事务来保证正确性，而在其他对数据一致性有较高要求的场景下也需要数据库提供事务能力支持。设计优秀的数据库往往也带有完整的事务系统，能够按照用户需求提供不同一致性要求和效率的隔离级别满足不同场景需求，这样用户就不必在业务代码中过多涉及一致性考量，因此，好的数据库事务系统能够减小事务这个概念对业务代码的侵入性，同时提高用户编写代码的正确性。

随着数据存储量的快速增长以及跨国公司和业务的大量出现，分布式系统也在现实中越来越重要。但分布式系统会面临网络不稳定的客观因素，网络传输延迟甚至丢包会引发后续的多种分布式系统特有问题，因此，在分布式系统中重新实现以往的各种单体系统上的功能也仍然是研究和实践的重要关注点和难点，甚至衍生出了共识问题等相关领域。

综上，本文关注的分布式属性图数据库以及数据库事务，在现实中仍然有着重要的意义和使用场景需求。

【这里是一张图介绍为什么图数据库中也要事务】

## 研究现状

相比关系型数据库领域，NoSQL数据库领域是一个新兴的研究领域，而图数据库系统更是NoSQL数据库中几乎最新的一个领域，因此不论学术界还是工业界的发展都有大量的空白部分[6][7][8]。首先，图数据库系统缺乏统一标准，图数据库从语义上只需要遵守数学中的图概念，具体如何表征图、描述图以及存储图都缺乏统一标准，且对于图的顶点和边的细节也呈现出“各自为战”的现状：例如，一部分图数据库不支持顶点和边带有属性而另一部分允许，一部分图数据库不允许悬垂边存在而另一部分允许，一部分图数据库图模型为无向图而另一部分为有向图。

同时，关系型数据库查询语言基本兼容SQL标准，在基本的数据操作语言（DML）和数据定义语言（DDL）方面，主流关系型数据库相互兼容性很强，提供了相似度很高的API，但图数据库系统使用了完全不同的几种查询语言，包括但不限于Gremlin，Cypher，GraphQL以及本文使用的nGQL，一部分语言是声明式类SQL语言，而一部分语言是命令式的类似对象关系模型的查询语言，针对的图数据库类型也不同，这导致了在不同图数据库系统间迁移的学习成本较高，这一领域的通用性较低[9][10]。

此外，由于不同图数据库使用的底层存储大相径庭，对图的切分方式也不同，这导致了底层数据难以从一种图数据库中迁移到另一种中，缺乏统一标准造成了生态的割裂，也难以使用相同的数据集和操作集进行性能测试。

在不允许顶点和边带有属性的RDF图数据库系统领域，W3C提出了针对RDF图的标准，统一了RDF图数据库的数据结构、数据表示以及图查询语言[11]。但在允许边和顶点有属性的属性图数据库领域，仍然缺乏这样的统一标准。

对于数据库事务的研究持续已经有半个世纪之久，但基本只集中在关系型数据库领域，人们为关系型数据库实现了众多的细分事务隔离级别，将事务隔离级别从SQL-92标准的四个级别扩展到更具体的行为表现上[12]。对于非关系型数据库的事务研究处于长期的空白，一直以来NoSQL数据库更关注目标数据结构的快速存取而不太关注事务的完整性，大多采用设计幂等API并失败后重试的方式来完成事务的功能。但当涉及到金融场景等要求严格的场景时，缺乏事务支持的NoSQL数据库并不能完全胜任。

分布式系统领域同样是一个较新的研究领域，虽然针对分布式系统的理论研究早在半个世纪前就已有人提出，例如共识问题，但直到商业化的分布式数据库等软件开始得到广泛使用，这一领域才有了更加实际而快速的发展。二者的结合，分布式事务，是一个难题，其中的很多子问题被证明是NP完全问题，经典的CAP理论的提出更是对分布式系统的现实实现提出了很大挑战[13]。如今，分布式数据库实现的事务大多隔离级别较弱，只能保证一段时间后数据最终一致，实现了强一致的分布式数据库由于网络通信延时以及等待回复等原因，事务延迟较高，吞吐量较低，还可能比单机系统发生更严重的冲突问题，因此这一领域仍然没有一个很好的实现。

## 研究内容

本文基于一款国产开源分布式属性图数据库Nebula Graph，在其存储服务上进行改动，提出了一种使用多版本并发控制协议的事务实现方案[14]。Nebula Graph在键上最初实现中为数据库事务留出了时间戳字段，但在目前的最新大版本中取消了预留字段，说明其在短时间内是不会实现数据库事务的。因此，本文沿用其原有的实现方案，同样采用在键上增加时间戳的方式，基于多版本并发控制协议实现事务隔离，并类比关系型数据库，实现了常用的读已提交和可重复读隔离级别，为了完整性，还实现了兼容这两种隔离级别底层数据的可串行化隔离级别，为图数据库的日常使用提供了多级别的隔离支持。

本文的研究关键是将属性图的多版本存储根据访问特点拆分成属性的多版本存储和点边关系的多版本存储，并根据访问序列特征选择合适的版本存储方式。同时，建立一个完整的事务系统需要事务管理调度、内存版本索引以及持久化系统共同工作，在多种网络延迟情形以及事务回滚故障情形都要保证系统的宏观一致性。

对于数据库事务并发现象，不同的事务实现细节会导致不同的结果，即使是同一名词概念在不同的开源产品中也可能在某些场景下表现出不同的行为，因此对于并发冲突场景，本文针对自己的实现来具体分析，并设计实验加以验证。

## 论文结构

本文由六个章节组成，第一章为引言，1.1节介绍了本文研究工作的现实背景和意义，1.2节简要介绍了目前图数据库领域以及数据库事务领域的研究和产品现状，接着1.3节介绍本文研究工作的主要内容和重点，最后1.4节介绍文章的整体结构和组织。

第二章为相关工作，主要是对图数据库领域、数据库事务领域以及分布式数据库领域的相关实践的调研以及相关文献的综述。2.1节首先概述图数据库系统，介绍其分类以及特征，2.2节概述数据库事务以及并发冲突现象的研究成果，并定义相关概念，2.3节介绍多版本并发控制协议，概述了多版本并发控制协议的分类和各自的特征及适用场景，2.4节简要概述了现行通用的几种分布式ID生成算法，2.5节介绍了分布式系统的时钟问题，概述了几种分布式系统时间的解决方案，2.6节简要介绍了本文选择的开源图数据库Nebula Graph的系统架构和与本文实现相关的实现细节。

第三章为多版本并发控制的实现，是本文的关键部分。本章对多版本并发控制协议选型做了分析，将属性图的多版本存储拆分为两个部分并分别进行了存储方案的适配，接着实现了内存索引部分的垃圾回收算法，最后完整介绍了各事务操作的具体流程。

第四章为持久化设计，是本文实现完整的事务不可或缺的重要部分，设计了一种本地恢复管理系统，针对第三章中的两种内存索引操作进行了持久化操作，并设计了反向读入和恢复内存索引的机制以及为避免磁盘文件无限制增长使用的重写方法。

第五章为实验与分析，针对本文设计的事务系统以及持久化系统编写了测试样例进行实验，同时还对第二章提出的几个并发场景进行了实验，验证了本文设计的系统的隔离级别，并针对图数据库系统特有的图操作的底层算子进行了验证。

第六章为结论，总结了本文的研究工作，指出了本文可以改进的部分以及未来继续研究的方向和展望。

# 相关工作

## 图数据库系统

图数据库是一种NoSQL数据库，NoSQL数据库是与传统的关系型数据库相对的一个概念，即所有不属于关系型数据库的数据库都可以称为NoSQL。NoSQL主要包含四种类型，分别是键值数据库、文档数据库、列式数据库以及本文研究的图数据库。键值数据库是一种以键值对高效存储信息的数据库，一条值数据仅可通过它的键引用，一般仅提供Get、Put、Delete三种类型的接口，代表有Redis、LevelDB以及RocksDB[15][16]；文档数据库主要用于管理文档，存储的内容是半结构化的数据，不需要预先定义模式，且支持通过文档中不同字段引用数据，代表为MongoDB[17]；列式数据库按照列进行数据连续存储，因此可以省去关系型数据库中的列索引等要求，由于数据按列存储，因此大规模读取列的值以及压缩机制都很高效，适用于之前提到的OLAP场景，代表有HBase、Hive和BigTable[18]。图数据库以点和边作为基础存储单元，将实体表示为点，将实体之间的关系表示为边，因此可以高效地表示关系，并可以高效支持遍历操作，代表有Neo4j和Huge Graph[19][20]。在关系型数据库中，实体之间的关系需要依靠表之间的外键来表示，是一种较弱的关系模型；键值数据库与文档数据库都只支持按照键或其他元信息存取对应的数据，因此无法表示实体间的关系；列式数据库与关系型数据库表示关系的方式相似，也无法很直观的表示实体间的关系。因此，图数据库对关系表示的强支持是其区别于其他存储系统的显著特点，其可支持各种对实体间关系以及基于关系的遍历有强需求的场景，例如社交网络、推荐系统、金融风控、知识图谱等。

图数据库系统可以按照不同的方式分类。

按照底层存储方式可以分为原生图存储数据库以及非原生图存储数据库，原生图数存储数据库对数据的存储和对图的处理进行了设计优化，适配各类图算法以及图上的快速遍历，例如Neo4j，采用了无索引邻接(index-free adjacency)方式，逻辑上关联的节点在数据库中存在物理连接，使得图遍历过程中寻找边和对应的顶点能在常数时间内完成（对比关系型数据库，有索引情况下需要O(logn)时间，无索引情况下需要O(n)时间），在关系递归层数较深时有显著性能优势。非原生图数据库系统主要采用关系型数据库或键值数据库等方式直接存储图的点和边，在存储层上搭建一层语义层提供图操作，在多读查询等图特有情形的性能劣于前者，例如JanusGraph，其实现了计算引擎层，可以搭建在HBase等流行的分布式数据库上使用，具体的顶点和边的存储方式与采用的底层存储数据库有关[21]。

按照点和边是否可以携带属性可以将图数据库系统分为属性图(property graphs)数据库系统、超图(hypergraphs)数据库系统以及RDF三元组(RDF triples)图数据库系统。超图数据库系统的数据模型是超图，不同于一般的图的边只能连接两个顶点，超图的一条边可以连接多个顶点，适用于表示存在大量多对多关系的模型，例如M个人共同拥有N个物品，如果使用一般的图，需要M\*N条边表达“某人拥有某物”的关系，但如果使用超图，则只需一条边即可表示，因此适合于表示元信息场景。RDF三元组存储来源于语义网运动，将所有的数据都存储为“主语-谓语-宾语”三元组，用多个三元组来表达实体的属性以及实体之间的关系，可以将图模型以W3C制定的RDF标准进行模型化表示，且使用统一的图查询语言SPARQL，是图数据库领域中行业标准建设最好的一个领域，代表产品有RDF-3X、gStore和Virtuoso。但RDF三元组不支持针对图计算处理的优化，大多采用键值数据库作为三元组的底层存储，同时由于三元组的唯一性，也难以表示出现了多次的相同类型关系，因此总体上不如属性图数据库系统的性能和便利度。属性图数据库系统中的顶点和边都可以拥有属性，顶点可以拥有任意个标签用于标识顶点的类型，因此可以有效区分实体的属性和实体间的关系，并可以针对顶点和边进行存储优化。相比RDF图数据库系统，属性图数据库并未形成统一的业界标准，各种开源或商业产品对图的顶点、边、属性都有不同的规定，也使用不同的图查询语言：Neo4j允许点和边都有多个属性，每个顶点可以定义多个标签，同时属性是无schema要求的，将顶点、边、标签以及属性分别存储在不同位置，使用Cypher语言进行操作；JanusGraph同样允许顶点和边拥有属性，但还提供额外的用户自定义设置来对顶点和边进行限制，采用切边方式进行存储，每条边会被以其源顶点和目的顶点分别存储一份以便双向遍历，使用Gremlin语言进行操作；Nebula Graph允许顶点和边有属性，但对属性的要求是强schema类型，因此同一种类型的顶点或边拥有的属性字段相同，每个顶点可以拥有数个类型，采用切边的方式，基于强schema设计出了免序列化随机读取属性的方式，并将边与相邻的顶点存储在同一空间中，便于图遍历操作执行，使用nGQL语言进行操作。

本文选择基于开源图数据库Nebula Graph进行修改，原因如下：以业界流行的Neo4j为例，它是一个单体架构的图数据库，企业版本采用集群部署而非现行分布式部署，因此在扩展性上有一定欠缺。Nebula Graph在设计之初就考虑到水平扩展以及分布式部署，采用shared-nothing架构以及存算分离的方式，支持多机房多分片的部署，允许计算引擎和存储层以及底层存储的接口化拆装，对于海量数据存储有扩展优势。同时，Nebula Graph对属性值的编码使得在执行带有条件的图遍历语句时不必对全部属性进行反序列化就可读出，这可以节省可观的CPU以及语句执行时长。Nebula Graph底层虽然采用键值数据库作为实际存储，但对边进行了编码以及在顶点的block处存两份边的操作，使得“查找一个顶点经过某类型边后的相邻顶点”这样的操作仍然是常数时间内完成的，而查找一条边的相关顶点更只需将边的键解码后即可获得，是符合图原生存储的。但作为一款较新的开源产品，Nebula Graph目前仅支持边一致性事务，即插入或删除一条边时在两个邻接顶点处的存储状态一直，而对于普遍意义的事务仍不支持。作为一款支持OLTP在线联机操作的数据库，为其实现单机事务以及分布式事务就具有现实意义。

## 数据库事务

数据库事务是包含一个或多个数据库操作的序列，在事务结束后，其中的操作要么全都执行，要么全都不执行。事务的英文transaction的另一个意思为“交易”，这标识了数据库事务的来源：在古老的合同中涉及金钱交易时的场景，并由此产生了事务的四个性质：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），满足以上四点的事务也成为ACID事务[22][23]。

与ACID事务相对应的为BASE事务，BASE理论来源于分布式系统，是对CAP理论的扩展，表示基本可用（Basical Available）、软状态（Soft State）和最终一致性（Eventually Consistent）。与ACID事务的强一致性不同，BASE事务更注重高可用性，允许一定时间内的不一致，这个不一致的中间状态称为软状态，系统处于软状态时不会影响系统可用性。由于BASE事务的高可用性，如今的分布式系统大多实现BASE事务而非ACID事务。

对传统单机系统以及要求强一致性的分布式系统，ACID事务仍然是必要选项，下文将分别介绍ACID事务的四个特性。

原子性指一个事务在宏观来看就是一个不可分割的原子操作，其中包含的操作序列要么全部执行成功，要么全都执行失败。此处的成功执行是指数据库执行层面的成功，即一个操作执行完毕获得了返回结果（即使这个返回结果不符合预期），因此原子性并不能完全保证事务的一致性。

一致性指系统必须从一个一致性状态转移到另一个一致性状态，且这些状态必须是逻辑上正确的。例如，在简单的两人转账交易场景中，任何正确的一致性状态下，两人的账户金额总和是相同的。一致性一方面要求事务的执行是原子性的，另一方面还需要用户层面逻辑的保证，例如一个操作执行成功，但返回结果不符合预期，此时为了保证系统一致性，需要操作者手动回滚之前的所有操作。

隔离性指不同的事务互相之间不影响、互相无感知，即多个并发执行的事务，每个事务内部的操作不会影响其余事务的读写情况。但这个定义是针对最严格的隔离性所定义的，生产实践中，常常采用更宽松的隔离级别，允许不同事务之间的某些操作在特定情况下互相产生影响。隔离级别从低至高依此被分为读未提交（Read Uncommitted）、读已提交（Read Committed）、可重复读（Repeatable Read）和串行化（Serialization），隔离级别越高则隔离性越好，但同时系统效率越低[24]。即使在最低隔离级别也不允许一个未提交事务修改另一个未提交事务写过的数据，因此事务的隔离性保证了没有脏写现象。

RU允许事务读到其他事务还未提交的修改，而这个修改可能在之后被回滚，从而产生脏读现象。RC要求一个事物的修改内容在提交前对其他事务不可见，所以不会产生脏读，但事务对同一值的前后两次读取可能由于中途有其余事务修改且提交导致读取内容不一致，这被称为不可重复读。RR会使用快照等方式记录事务对值的第一次读取的内容，此后发生的读操作只会读这个快照，可以避免前后读取的不一致，但当执行范围查询操作时，可能发生其他事务插入一条新的记录，导致该事务后续读取读到新增行的问题，这被称为幻读，幻读问题需要使用范围锁来解决。串行化则将所有并发的事务都串行执行，解决上述的问题。

除了脏读、不可重复读、幻读之外，隔离级别不够还可能导致读偏斜（Read Skew）、写偏斜（Write Skew）以及丢失更新（Lost Updates）现象，它们是用户业务层面的不一致。读偏斜指事务的读结果违反业务要求的一致性，本质是因为业务一致性需要多条数据保证，而这些数据中存在状态的新旧区别，例如上述场景中事务B从x转账到y，若事务A在转账前读取了x的账户余额，在事务B提交后又读取了y的账户余额，就会发现x与y的总和违反了业务的一致性。写偏斜是由于事务中写操作的前提被破坏，一般发生在冲突事务的写前提和写目标不是同一数据的情况下，特别是事务的写前提和写目的恰好相反，例如事务A读取x的值并写入到y，事务B读取y的值并写入到x，最终结果可能产生x≠y的现象，与设想不符。丢失更新发生在写前提和写目的相同的情况下，对同一个目的的写入会被覆盖，例如事务A读取x并更新为x+20，事务B读取x并更新为x+10，若B先提交，则之后A的提交会覆盖B的更新，x最终只被更新为x+20，而非x+30。丢失更新与脏写的区别在于，脏写读到的值来源于另一个未提交的事务，是脏读的结果，而丢失更新独到的值是已经提交的结果。

持久性则要求事务一旦执行成功，其对数据库造成的影响将持久存在，即使数据库存储系统立即宕机也不会丢失这些修改。前三个性质主要定义了事务执行过程中的部分要求，而持久性是事务执行完成后的要求，因此一般会有独立的持久化系统对事务执行记录进行整理和落盘操作，一旦宕机发生还能借助持久化系统进行恢复。

事务最重要的是一致性，通过原子性的数据库操作，持久性带来的数据持久化和可恢复，搭配不同隔离级别，可以实现不同的一致性级别。不同的数据库事务实现在效率和一致性级别二者间做出了取舍，一般而言，一致性要求越高则效率越低，实际数据库系统往往提供多种隔离级别以满足用户的不同一致性要求，且默认状况下并不会选择一致性最好的隔离级别可串行化，而是在可重复读或读已提交隔离级别下实现事务。

## 多版本并发控制

并发场景指对相同数据的不同访问存在时间重叠的场景，在此场景下不同操作可能存在冲突。将操作分为读操作和写操作，写操作会修改数据本身，读操作不会，则存在读-读并发、读-写并发以及写-写并发场景，其中读-读并发不会引起冲突，而读-写并发与写-写并发会显著影响并发场景效率。

多版本并发控制，即Multiversion Concurrency Control，下文以MVCC代称，是一种通过存储同一数据的多个版本来实现数据库的并发控制的方式。传统数据库中，每一份数据只会存储一个版本，采用读写锁的方式进行并发控制，对同一个键的读以及写会互相阻塞，读操作结束后写操作才能开始，可能出现写饥饿现象。在存储了多个版本数据的情况下，读操作与写操作将不再互相阻塞，因为写操作可以创建一个仅自己可见的版本，而读操作只需读取可见的符合时间段的版本。对比传统基于读写锁的并发控制，MVCC以空间换时间的思路，以一定的多版本存储代价解决了读-写冲突，大幅提升了并发效率[25]。

使用MVCC技术可以构建出不同的隔离级别，例如将新写入但未提交的版本标记为私有版本即可实现读已提交（Read Committed，下称RC），在此基础上引入快照，保存某一时间点的版本数据，即可实现可重复读（Repeatable Read，下称RR）以及快照隔离[26]。对于写-写冲突，MVCC协议内实现了不同的解决思路，对写-写冲突的不同解决方式是各种MVCC的核心。

并发控制协议从大方向看可以分为乐观协议和悲观协议两类，在目前的单体数据库中大多采用悲观协议。悲观协议认为未来会出现较多的竞争和冲突，因此在可能发生冲突的阶段会预先排除冲突，确保一定能执行成功后才开始提交，根据具体实现可以细分为基于锁的和非锁的两类，基于锁的悲观协议又可分为两阶段协议及其他协议，例如TO和SGT。

基于锁的两阶段协议代表为2PL（Two-Phase Locking）以及其后续的各种优化协议，两阶段协议严格地从时间上划分出加锁阶段和释放锁阶段，释放锁后不允许再加锁，按照加锁以及释放锁地时机和类型又可以分为保守两阶段封锁协议（Conservative 2PL，C2PL）、严格两阶段封锁协议（Strict 2PL，S2PL）以及强两阶段封锁协议（Strong Strict 2PL，SS2PL），其中S2PL在实现单体可串行化中运用较多，在分布式场景下甚至需要SS2PL来保证可串行化。

基于锁的非两阶段协议以封锁树为代表，分为只写封锁树（Write-only Tree Locking，WTL）和读写封锁树（Read/Write Tree Locking，RWTL）以及后续的DAG封锁协议（DAG Locking Protocol），封锁树协议要求对数据的访问遵循一定的顺序例如树状访问，允许沿着树状路径访问时交替申请和释放锁，所以是非两阶段的协议。封锁树通过控制访问路径来尽可能避免死锁现象，例如，WTL是无死锁的协议。

非锁的悲观协议有两大代表，分别是时间戳排序（Timestamp Ordering，TO）和可串行化图检测器（Serialization Graph Tester，SGT）。TO协议不对操作加锁，对于存在冲突的操作，严格按照其开始时间的先后关系进行调度，因此如何产生一个准确且唯一的时间戳是这一协议的重要基础。SGT协议源于2PL类协议在实现串行化隔离级别时开销太大的问题，其会维护一张串行化图来分析并发冲突，在发生了写-读依赖、写-写依赖以及读-写反依赖的情况下在节点之间连上边，最后如果可以产生一个有向无环图则表示可串行化执行，否则会产生死锁。由于SGT没有管理锁的成本，因此性能会比传统基于锁的2PL协议更好，基于SGT可以实现工程级别性能更优的串行化快照隔离（Serializable Snapshot Isolation，SSI）。

与悲观协议相对，乐观协议认为未来只有很少甚至没有并发冲突和竞争，因此会尝试直接提交，遇到冲突时再进行回滚。将乐观协议和悲观协议的操作划分为有效性验证（V）、读（R）、计算（C）和写（W）四个阶段，则悲观协议地顺序为VRCW，乐观协议地顺序为RCVW，即乐观协议将可提交性的验证放在实际读取和计算后、写入数据库之前。乐观协议在执行有效性验证时同样需要一定的凭证，例如基于锁的检查（称为乐观锁）或基于时间戳排序，因此乐观协议并非完全无锁的。

乐观协议的代表为乐观并发控制（Optimistic Concurrency Control，OCC），在实际数据库中应用较少，更多存在于学术界论文的机制设计中，原因主要有以下两点。第一，乐观协议假设未来冲突很少，而一旦出现总流程持续时间较长的含有写操作地情形，就会较大地破坏这一假设，在如今的互联网时代并发冲突可能性越发密集，因此乐观协议在工程上实用性较低。第二，单体数据库主要以悲观协议为主，其中绝大部分是基于锁的2PL系列协议，因此为了兼容原有的系统在并发控制上的实现，许多新的数据库产品依然使用悲观协议实现并发控制[27]。

将以上的并发控制协议与多版本存储方式结合，就产生了不同种类的多版本并发协议，代表性的MVCC协议为MVTO、MVOCC、MV2PL以及SSI/SSN[5]。最后一种MVCC协议基于可串行化图，而前三种协议有一部分共同点，都需要使用时间戳或递增版本号标识数据的每个版本，且将同一数据的各个版本按照顺序依次串连成版本链。其中，MVTO在每个版本上维护了最近的读操作的版本号以及当前写操作的版本号，不允许两个及以上写操作同时进行；MVOCC只在个版本上维护当前写操作的版本号，当写操作要提交时，系统会给其一个新的时间戳，称为提交时间戳，用来安排串行化顺序，并会检查它的读集合是否被修改过，如果未被修改过则通过验证阶段可以提交写入；MV2PL在每个版本上维护了写者的版本号以及读者数量，使用读写锁控制读操作和写操作对数据的版本。

## 分布式ID

实现多版本并发控制需要一个具有偏序关系、递增趋势和唯一性的值来标识各事务，这个值称为事务ID。在单机系统中，可以基于服务器系统时钟生成唯一且递增的事务ID，或使用一个原子整型变量作为版本号分配给每一个事务线程。但在分布式系统中，若要实现分布式事务，需要生成全局唯一且具有偏序关系的事务ID，如果仍然采用服务器本地时钟生成时间戳，则可能存在两个线程同时到达不同的服务器并获得了相同的事务ID的问题。因此，分布式事务需要新的分布式ID生成算法支持，以下介绍几种现有分布式ID生成算法的实现。

UUID：UUID全称Universally Unique Identifier，即全局唯一标识符，是在每台机器上本地生成全局唯一ID的算法，生成效率很高，但UUID是随机的，无法做到递增趋势，可能暴露生成UUID使用的MAC地址或别的唯一本地标识符，同时16字节128位长度也不便于存储。

中心发号器：在分布式系统中增加一个中心组件，由这个组件本地生成事务ID，分布式服务器通过RPC与中心组件交互获取事务ID，此方式以Apache Zookeeper为代表[28]。这种方式的优势在于实现简单，可以直接使用单机ID生成算法，也保证了单调递增和全局唯一性，分布式服务器还可以通过批量获取ID的方式降低RPC次数，但中心节点的存在以及网络调用延时仍会成为系统并发瓶颈，以及出现单点问题，高并发场景下实用性较低。

本文使用的是雪花算法，这一算法名字来源于“世界上没有两片相同的雪花”，由“本地时间戳+集群ID+机器ID+机器内自增序列号”组成事务ID，由于本地时间戳放在最高位，因此整体呈现出递增趋势，同时因为集群ID+机器ID可以唯一标识每一台机器，所以也能保证事务ID的唯一性。雪花算法不依赖网络传输，生成性能高，查询效率也比UUID高，但强依赖机器的本地时钟，一旦出现时钟回拨现象将出现发号不可用问题。雪花算法可以生成递增趋势和唯一性的事务ID，但不能保证严格的递增——若事务a访问服务器A，事务b访问服务器B，但由于A的本地时钟慢于B，即使a晚于b到达各自的服务器，a的事务ID仍可能早于b的事务ID。使用非中心化发号器的分布式ID生成方案都需要解决系统本地时钟不一致的问题，因此引出了分布式时钟算法。

## 分布式时钟

分布式时钟是为了解决分布式系统时间一致性而产生的问题的一种解决方案，能够确定发生在不同分布式系统上的事件的先后关系。当两个事件发生在同一机器上时，可以确定其发生的先后顺序，但如果发生在不同机器上，由于机器本身时钟不同步，所以无法根据事件的时间戳确定先后顺序。

在分布式数据库中，由于各物理机的本地时钟不一致，基于本地时钟生成分布式ID的实现下，可能出现不同机器上的事务发生的先后顺序不可比较的问题，即使采用中心化授时机制，也会因为网络延迟出现本地时钟不一致。这会导致被分发到不同结点处理的事件的真实先后顺序与其ID体现的先后顺序不同。因此，对分布式系统找到分布式时钟算法来决定全局先后顺序，在许多情况下是对数据一致性以及冲突解决的保证。

经典分布式时钟实现分为物理时钟和逻辑时钟，物理时钟包括使用GPS和原子钟校准的时间系统True Time和对应的提交-等待协议，以及中心化授时系统TSO[29]；逻辑时钟包括对有因果的事件进行排序的Lamport时钟（LC），能够记录各结点逻辑时钟的向量时钟（VC），以及同时使用物理时钟和逻辑时钟以节省时钟存储空间的混合时钟（HLC）设计[30][31]。

在分布式系统中，多个结点上的数据副本可能会被不同访问请求同时更新，这就会产生数据冲突以及冲突解决需求，需要使用基于向量时钟的版本向量进行记录。版本向量的实现中，一条数据被服务处理时，会带上该结点的编号以及本地序列号，各结点的编号及序列号构成了数据上的向量时钟，通过这个向量时钟来判断冲突发生（在向量时钟算法中，没有因果关系的两个事件称为同时发生，而如果这两个事件是同一数据的不同副本，则说明发生了数据冲突），并按照一定方法解决数据冲突后写入合并后的版本。由于向量时钟的存储空间占用正比于节点数目，在分布式部署情况下是一个较大的开销，因此出现了与向量时钟功能类似但存储空间为常量的混合时钟，混合时钟可以被设计为一个64-bit的字段，与现有常用的时间戳大小相同。使用HLC可以确定任何有因果关系或发生在同一节点上的事件的先后关系，能够更好解决分布式一致性问题。

## Nebula Graph系统架构

Nebula Graph是一个分布式属性图数据库系统，它是强Schema的，要求属性必须以Schema形式被定义，且允许顶点和边都有自己的属性。它可以以集群方式部署在多台机器上，采用存算分离方式实现，包含三种服务，分别为Query Service、Storage Service以及Meta Service。Meta Service不仅承担图数据的meta信息存储和管理，例如顶点和边的schema信息以及用户鉴权和集群管理，同时会承担数据迁移指挥工作以及leader变更等运维操作。Query Service是无状态的查询计算引擎，彼此之间无通信关系，仅仅从Meta Service处获取meta信息并通过rpc方式与Storage Service交互，因此允许计算层被部署在云平台上。Storage Service是存储层，采用shared-nothing架构，底层物理存储采用可插拔实例，默认为基于RocksDB实现的KVStore，上层为Storage interface。存储层采用Raft协议来保证KVStore一致性，并使用图语义方法将计算层传来的图操作转换为底层KV操作。

Query Service与SQL执行引擎类似，接收查询语句nGQL，经过Parser、Execution Planner、Optimizer和Executer后转换为一系列具体图操作，并发送到存储层执行。【此处应有Query Engine的流程图】

分析器（Parser）基于flex和bison，将nGQL转为抽象语法树AST，执行计划器（Execution Planner）会对AST上的结点进行语义正确性校验，并将其转换为执行动作，这些执行动作是图API的包装，合在一起构成了一个执行计划，再经过优化器（Optimizer）后到达执行器（Executer），执行器会每一个执行计划中的动作逐个或并发执行，使用rpc客户端与Meta Service以及Storage Service进行通信

Storage Service分为三层，底层为本地存储引擎，只需要满足get、put、scan、delete等接口定义，所以允许使用者自行设计和选择物理存储引擎。第二层为共识层，使用Raft协议按照分片达成存储共识。最上层为存储接口层，提供了图操作API，这些API会在这层被翻译为对应的KV操作。

Nebula的顶点表示和边表示：Vertex key为Type+PartID+VertexID+TagID，对于无Tag的顶点，key为Type+PartID+VertexID，其中VertexID是字符串类型，使顶点可以有带具体含义的名字，顶点长度限制在创建图空间时指定。Edge key为Type+PartID+VertexID+EdgeType+Rank+VertexID，两个VertexID为边的两个端点。与一些图数据库不同，Nebula Graph的边没有独立的编号，查找一条边必须指定至少一个顶点，这是由于现实中图查询场景往往会从一个或一类顶点出发，查找指定的关系，因此提供从顶点访问关联边的操作可以满足绝大部分现实场景需求。这样编码的好处在于可以利用RocksDB的前缀范围扫描功能，通过Type区分顶点、边以及其他元素后，指定PartID，可以批量读取一个分片上的所有顶点或所有边，还可以批量读取一个顶点的所有出边、所有入边或所有关联边。

【此处应有边和顶点结构的表示图】

Nebula中使用Tag实现顶点的Schema，一个顶点可以关联多个Tag，每个Tag都有自己的属性定义，因此VertexID+TagID可以确定唯一的一组属性值，因此可以使用VertexID+TagID进行编码作为key，属性值编码作为value存储在KVStore中。同理，Edge Type可以用来定义边的Schema，Nebula中规定一条边只有一个Type，因此使用边的起点、终点、Type以及Rank值可以唯一确定一条边，所以使用SrcID+EdgeType+DstID+Rank的方式编码作为key，边的属性值编码作为value存储。

Nebula存储采用切边存储，每一条逻辑边会被存储两次，一次为SrcID+EdgeType+Rank+DstID，此时Type大于0，表示一条出边；一次为DstID+EdgeType+Rank+SrcID，此时Type小于0，表示一条入边。出边与SrcID对应的起点存储在同一个分片上，入边与DstID对应的终点存储在同一个分片上，因此可以方便地执行图查询语句，例如getNeighbors会获取一个顶点的所有关联边，由于这些边都存储在同一个分片上，且由于上文所述的编码方式，可以使用低代价的前缀扫描获得结果；同时，进行图遍历操作时，一条“入边－顶点－出边”路径上的三个元素都被存储在顶点所在的分片上，无需进行额外的关系索引既可以获得，因此Nebula Graph的存储层实现了图原生存储，大幅提升查询效率。

由于Nebula Graph采用了强Schema设计，因此不同于传统采用KVStore存储数据的系统，在读取和写入特定属性值时不必进行序列化和反序列化操作，这在带有where条件查询的语句中有着更大的CPU使用率以及延迟方面的优势。如果没有Schema预先定义属性值，读取特定的属性必须要先将value反序列化后再获取，即使实现了计算下推，将where子句传入存储层，也依然存在大量计算资源浪费，这在属性较多、语句范围查询涉及范围较大时尤其明显，例如图查询场景中常见的找到一个顶点的符合特定条件的出入边，若顶点的关联边较多，序列化与反序列化的消耗会比较明显。采用强Schema设计情况下，访问一个属性值时，会先读取Schema获得属性对应的偏移量以及属性的长度，直接从value中读取或写入相应字段到对应位置即可。对于变长类型例如字符串，Nebula使用定长指针+后置内容存储的方式，经过两次读取，使得通过Schema可以确定各类型属性的起始偏移量。

# 多版本并发控制实现

## 整体结构

本文基于分布式开源图数据库Nebula Graph进行了事务的实现，在其基础上主要修改了Graph Engine层的解析器以实现事务的开始（BeginTx）、提交（CommitTx）和回滚（AbortTx）操作的解析，在Storage层增加了Transaction Manager用于管理事务，Index Map和VE Map用于键的多版本管理，以及LRM Manager用于将操作持久化和重启恢复；为了生成唯一的事务ID，本文还修改了Storage服务与Meta服务的接口，并实现了基于SnowFlake算法生成的唯一ID。

对于MVCC的四个隔离等级，本文对读已提交（RC）、可重复读（RR）和可串行化（Serializable）进行了实现。读未提交（RU）的隔离级别会导致读到其他事务未提交的修改，在现实中很少使用，只需要使用写锁消除脏写现象即可，实现较为简单；可串行化则要求所有事务的执行序列都是可串行化的，是最高的隔离级别，对于写事务加排他锁，读事务加共享锁，本质上无需多版本支持。为了兼容另外两种隔离级别写入底层KVStore的带版本号的键值对，本文实现可串行化隔离级别时依然保留了事务的版本号，使用普通读写锁代替MVCC的锁进行加锁。

Nebula Graph是一个属性图系统，因此其多版本存储可以拆分为两个部分，分别是顶点连接的出边和入边的多版本关系，以及顶点或边与对应的属性之间的多版本关系。对于前者，只有边的插入和删除操作会影响其结果，因此选择差分版本形式存储相应的边操作序列，在读取时则按照顺序依次执行差分操作进行复原，不涉及与底层KV存储的交互，相对简单。对于后者，涉及到同一个顶点或边的键对应的属性不同，需要建立对同一键的多版本存储和索引机制来与底层KV存储交互，相对复杂。

本文建立多版本机制的内存索引参考了一个分布式强一致键值数据库etcd的实现思路，在内存中维护了每个键对应的版本号序列，实际存储中的键为原始键与版本号的拼接[32]。

Index Map是一个常驻内存中的哈希键值对，维护了点或边形成的键与各种对应的一系列版本号的关系。

VE Map包含出边哈希表和入边哈希表，记录了一个顶点对应的出边和入边的多版本关系，提供只需要访问内存索引就能快速查找顶点出入边的图遍历语句支持。

Transaction Manager是事务实现的关键，承担了事务注册、提交、回滚等功能，提供实务具体操作的API，并维护当前活跃事务集，为垃圾回收等功能提供支持。

由于Index Map和VE Map是内存中的结构，因此需要对其进行持久化，用来断电重启后恢复，于是此处参考了Redis的实现方式，设计了一个LRM Manager用来写入对内存索引的操作到磁盘文件中，以及在系统恢复时读取文件重建索引，同时还会定时进行重写，合并以及清理过时版本的操作记录，以避免磁盘体积无限制增长。

【此处为Index Map、Transaction Manager以及KV store的交互图。】

## 多版本并发控制协议选择

多版本并发控制协议存在多种选择，大体上分为MVTO、MVOCC、MV2PL以及基于依赖的有向无环图冲突分析。过往实验显示，MVOCC由于使用乐观锁机制，在读写比较大时写入冲突较低，因此性能良好；但在写操作占比逐渐增加的情况下，由于验证阶段不通过导致的事务重启次数较多，性能下降明显。MV2PL是基于悲观锁实现的两阶段提交协议，性能稳定，但现有分布式数据库中使用也不多。权衡了可能的使用场景以及为了在写操作占比较多时仍能有稳定的事务性能，本文选择基于MVTO作为多版本并发控制协议进行了实现。

选择MVTO作为MVCC协议，需要为每个多版本存储增加三个字段：txn-id、read-ts以及begin-ts，每个字段都是int64大小，占据8字节。其中，txn-id记录当前条目的写入者编号，如果没有写入者则该项为0，起到了写锁的作用；read-ts记录距离最近的读者编号，这可以保证“迟到的写者”无法更新新的读者读过的内容；begin-ts表明起始生效时间，当且仅当事务的版本号大于等于begin-ts才表明该版本对该事务可见。

在原始论文中，MVTO还需要end-ts用来表示版本结束时间，但由于实际操作中同一键的多版本都以链表形式连接，前一版本的end-ts等于后一版本的begin-ts，因此只需要存储起始时间，再与后续版本的起始时间进行比较即可找到事务对应的版本。

同时，原始论文中，写入新版本时，会对当前最新版本Bx加写锁，并生成一个带写锁的未设置begin-ts和read-ts的版本Bx+1，直到写事务提交时再设置Bx+1的begin-ts并释放这两个版本的写锁，这会使得比写事务开启更早的迟到的读事务无法访问到其本可以访问的最新版本Bx而Abort。因此本文将实现修改为：写操作将带有写锁且设置了begin-ts和read-ts的最新版本Bx+1添加到版本链后，释放Bx的写锁。一方面，由于版本链上的最新版本始终持有写锁，所以只要保证写操作校验时始终获取的是版本链末端的版本，即可以避免写-写冲突。举例说明，一种可能的并发场景时序为：(1)A获取版本链末端版本Bx，(2)B获取版本链末端版本Bx，(3)A获取Bx写锁并添加Bx+1版本后释放Bx写锁，(4)A获取Bx的写锁成功。为了避免这种并发场景出现，事务对当前获取的版本加锁时需要验证该版本仍然是队末版本。本文使用了一个全局哈希表，为每个key增加一个对应的int64原子变量，维护每一个key的最新版本号x，事务在获取老版本写锁前原子性比较持有的版本是否是x，如果是最新版本则更新x为自己的事务ID（上文中例子为x+1），这样就可以避免其他持有Bx版本的写者进行下一步。另一方面，本文对新版本read-ts的修改可以防止迟到的写者写入。举例说明，A和B是两个含有对同一键的写操作的事务，TA<TB，一种场景为B先写入key并提交，此时版本链上最新版本的read-ts=TB，因此迟来的A将无法再写这个key，只能回滚，这有效防止了丢失更新现象的发生。

传统读写锁实现中，访问共享区域变量需要互斥锁结构，用于保证同时只能有一个写者到达共享区域，但这样的加锁-解锁操作往往对性能损耗较高。因此，本文对写者加锁采用了无锁CAS方式，利用Atomic库提供的compare\_and\_exchange方法原子性交换数值，对版本的txn-id进行操作，节省了加互斥锁的开支。但同时，由于无锁CAS的本质是多个写线程争夺写入的权限，同样可能存在重复碰撞-等待-重启的现象，因此可以引入网络传输协议中的随机重启算法，降低访问同一个键的不同写者之间的冲突。【TODO 将直接abort改为随机等待重启，减少事务重启次数】

本文采用的MVCC协议可以支持读已提交、可重复读以及串行化三种数据库事务的常用隔离级别，对于2.2节部分提到的并发冲突问题，不同隔离级别的情况如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 读已提交RC | 可重复读RR | 可串行化Serializable |
| 脏写 | 无 | 无 | 无 |
| 脏读 | 无 | 无 | 无 |
| 不可重复读 | 有① | 无 | 无 |
| 幻读 | 有 | 无② | 无 |
| 读偏斜 | 有 | 无③ | 无 |
| 写偏斜 | 无④ | 无 | 无 |
| 丢失更新 | 无⑤ | 无 | 无 |

注1：在RC隔离级别下，本文实现的协议虽然不能完全避免不可重复读现象，但从理论上可以降低不可重复读现象的发生概率：MVTO协议要求写事务版本号大于当前最新版本的read-ts时才可更新版本，即一个迟到的写事务A是无法更新读事务B正在读的键。但如果写事务A启动时间晚于读事务B，A依然可以更新B正在读的键产生一个新的版本并提交，使得B随后读取到的最新版本是A更新后的版本。【TODO 修改代码使得满足RC读操作要求】

注2：在RR隔离级别下，本文不会出现幻读现象，由于本文实现的MVTO协议会为读事务生成唯一的Read View，因此后续其余事务的更新不会影响根据Read View进行的快照读。在MySQL场景中，RR隔离级别只能避免快照读的幻读，而无法避免当前读的幻读（例如update语句会进行当前读），但nGQL要求update时指定对应顶点或边的id，因此不存在写操作的范围读场景，也就不存在幻读问题。

注3：在RR隔离级别下，由于事务第一次执行读操作时会生成Read View，因此其他事务后提交的更新不会影响到当前事务读取，所以不会有读偏斜现象。

注4：在RC隔离级别下，本文实现的协议要求写者不能更新已经被更晚开启的读者读过的键，而写偏斜场景下的两个事务一定有先后之分，因此必定有一个事务在写入时不能成功提交，因此不会发生写偏斜现象。

注5：丢失更新场景下的两个事务写入的是同一个键，在本文实现的协议中，不允许两个事务对同一个键进行写入操作，同时拒绝了任何迟到的写者写入，因此无丢失更新问题发生。

## 属性的多版本存储实现

同一键对应的多版本以链表形式在内存中存储，其中按照存储的顺序、位置以及存储内容等方面又可以分为以下四类：Append-Only: O2N、Append-Only: N2O、时间遍历存储以及差分存储。Append-Only存储中，每个键指向的位置固定为最新或最旧版本，已存储的版本只会在垃圾清理情况下被移除，因此更新的代价相对较低；时间遍历存储将各键的最新版本保存在主表中，更旧的版本保存在额外的表中，由此可以扩大多版本存储的版本数量，但在每次插入新版本时都需要将当前版本从主版本换出，更新代价相对较高；差分存储链表头部记录一个基准版本，随后是按照顺序排列的版本操作，在更新的内容不多的情况下相对于前三种全量存储方式能明显降低版本存储体积，但当读操作发生时需要遍历版本连依次进行复原，时间代价上升。

本文选用了Append-Only：O2N的方式进行存储，版本头部全量存储较旧的版本，尾部存储较新的版本。由于本文存储的差分版本是键的差分版本，不包含属性值，因此每个版本的大小可控，所以选择了Append-Only方式；实现中使用STL List，List是一个双链表结构，因此可以直接调用rbegin方法获取最新版本，再通过反向迭代访问到之前的版本，这使得大部分操作访问版本链的时间都在常数时间内，而与版本链长度无关。

每一个键对应的多版本都是一个链表，而键与对应的版本链组成了一个键值对，所有的键值对组成了全局结构Index Map。对于属性值的读写请求，会先访问Index Map获取或插入对应的版本，然后拼接出带有版本号的键，再向底层KV存储层读取或写入新的键-值。

【此处有版本链示意图】

多版本链中每个版本都记录了该版本的版本号，并额外使用两个字段作为读写锁、一个字段标记起始时间。由于本文实现中属性的多版本是针对Index Map建立的多版本，所以多版本内容与具体属性值无关，而恰好与版本的起始时间相同，所以可以将起始时间与要存储的内容合并，使得每个版本都只包含3个int64大小的字段。

与通行做法一致，本文将删除操作实现为标记删除，即，删除一个版本并不会直接在存储中删掉，而是将对应版本标记为失效，由后续垃圾清理来完成回收。由于起始时间字段实际是创建该版本的事务的版本号，是一个正数，所以标记事务为删除时，直接对这一字段取相反数得到对应的负值表示删除。这一做法可以在节省删除标志位的同时，保留版本的原始时间用来进行事务的回滚。

综上，本文的版本结构如下：

【此处是版本的struct代码，struct Version {int64 txn-id; int64 read-ts; int64 begin-ts;};】

txn-id为int64类型的写锁，read-ts为int64类型的读锁，begin-ts同时代表版本号以及该版本的起始时间。

【此处是Index Map类的代码，unordered\_map <string key, List<struct Version> >以及对index map的读写操作API】

## 点-边关系的多版本存储实现

点-边关系的多版本存储实现采用差分版本存储，将对边的插入以及删除两种操作记录按照事务版本顺序记录为条目。采用差分版本存储可以有效降低图中超级顶点在多版本存储中的存储量膨胀问题。

在Nebula Graph设计中，顶点由TagID和VertexID唯一确定，边由SrcID、DstID、EdgeType以及Rank唯一确定（不支持多入多出边），因此顶点和边本身是不支持修改的。由于边的编码中已经包含了SrcID和DstID，故一条边对应的出顶点和入顶点也是一旦建立就不会改变的。所以点-边关系的多版本只需考虑一个顶点对应的出入边在不同时间的情况，因此可以只记录边的插入和删除操作。

点-边关系的版本结构如下：

【此处是点边关系的struct代码，struct VEVersion { int64 txn-id; int64 read-ts; int64 begin-ts; vector<Edge> insert-edges; vector<Edge> delete-edges};】

Txn-id、read-ts以及begin-ts的含义与上一小节中含义相同，insert-edges数组记录本次事务中插入的边，delete-edges数组记录本次事务中删除的边。

【此处是VE Map的代码，unordered\_map <string VertexID, List<struct VEVersion> > OutEdges，unordered\_map <string, List<struct VEVersion> > InEdges】

读写操作与上一小节中对属性的多版本实现相似，由于将同一顶点的出边和入边分开存储，所以对同一顶点的出边的并发写入会被禁止，对同一顶点的入边的并发写入也会被禁止，而允许对同一顶点的入边和出边同时写入——这不会导致对同一条边的并发修改，是无冲突的。

每个事务内部会记录其插入和删除的所有边，用来在事务提交时生成持久化记录或在事务回滚时找到对应的VE Map修改并进行回滚。在本文设计中，使用一个动态数组维护每条边的编号，并使用cpp bitset库，用两个bitset进行记录，分别称为insertBitset和deleteBitset。

插入一条边时，将insertBitset对应位置1，删除一条边时，将deleteBitset对应位置0，总添加边集addBitset = insertBitset & (~deleteBitset)，总移除边集removeBitset = deleteBitset & (~insertBitset)。

## 可串行化隔离级别实现

MVCC协议本身不支持可串行化隔离级别，因为可串行化隔离级别需要使用锁机制配合实现，但为了本文系统的完整性，也实现了可串行化隔离级别。可串行化隔离级别不需要生成多版本存储，但由于持久化系统的存在，同样需要使用带有版本号的版本进行持久化日志写入，所以在实现可串行化隔离级别时仍然使用了3.2~3.4的设计。

对于可串行化，在使用锁的情况下可能出现死锁情况，例如，事务A先读x再写y，事务B先读y再写x，则不论使用互斥锁还是读写锁，如果A和B都完成了读操作，则会导致死锁。针对这个问题，一种做法是使用冲突可串行图检测技术生成事务执行序列，这会逐步建立一个有向无环的可串行化图，但本文为了实现简便，使用了非阻塞的读写锁代替，如果事务在执行加锁操作时不能获取锁，立即返回失败并回滚。

为保持兼容性，对应的内存索引结构不变，但实际实现中将版本链预设为只能存一个版本，即所有的读写操作都只访问第一个版本。使用pthread库中的读写锁，系统额外维护一个全局锁表，其中记录了每个key对应的读写锁，所有的读操作会使用pthread\_rwlock\_tryrdlock获取读锁，写操作会使用pthread\_rwlock\_trywrlock获取写锁，如果获取锁失败则会立即返回，并回滚整个事务。

## 垃圾回收算法

多版本存储会随着事务的增多产生更多的过时版本，因此需要进行垃圾回收。主流的GC操作分为Tuple-Level GC以及Transaction-Level GC，前者又分为后台线程回收以及事务内合作回收。后台线程回收机制下，系统会定时启动GC线程，按照一定的算法进行标记-回收，把过时的以及被删除的版本删除，缩短版本链的实际长度；事务内合作回收机制只适用于传统Append Only：O2N存储方式，在从头遍历版本链时可以同时回收靠前的失效版本，此方法可以避免GC线程标记回收时的“Stop the World”现象，但如果一个版本链一直没有被任何后续事务访问，则它的失效版本不会被回收，所以此方法需要与定时后台GC相结合；事务级别GC同样采用后台线程扫描，但以事务为级别进行回收，会将一个失效事务生成的所有版本都一同回收（一个事务被认为失效当且仅当其产生的所有版本都不再对活跃事务集中的事务可见），在关系型数据库中由于局部性原理，同一事务产生的版本通常存储在相邻位置，因此可以提高回收效率，缺点在于此法需要维护每个事务的读写集合，带来额外的开销。

本文采用了Tuple-Level后台线程回收机制，定时扫描Index Map和VE Map中的每条版本链，使用cpp future库并发地回收过期和被删除的版本。考虑到各版本的活跃版本链相对均匀，本文实现中使用了粗粒度的锁，对整个内存索引结构加锁，其中事务线程为读者，GC线程为写者，当且仅当所有版本链都回收完成时才解开写锁，允许后续事务的访问。出于代码简洁性，此处使用shared\_ptr模拟了defer语句进行锁的释放，不必在各个出错分支处手动解锁。

GC进行时，对Index Map和VE Map加写锁，获取当前最小活跃事务TID记为minTID，然后遍历每个key的版本链。对每个版本链，找到minTID可访问的最新版本，则比它更旧的版本可以全部删除，因为任何活跃事务都不会访问到这些版本。同时，GC时会遍历整个版本链，将事务回滚过程中标记删除的失效版本清除。GC结束时，会生成单version Delete操作的AOF entry，并批量写入AOF Manager，最后释放写锁。

## 操作流程

### 3.6.1 TX Manager接口总览

TX Manager是本文的事务系统直接与上层交互的模块，提供了事务管理和操作的一系列API，包括：事务管理的开启事务（BeginTX）、提交事务（CommitTX）、回滚事务（AbortTX）以及事务操作的插入操作（Create）、读操作（Read）、更新操作（Update）、删除操作（Delete），其中后四个API又被称为CRUD API。

TX Manager向下层与持久化系统、索引系统以及底层KV Store存储三个部分进行交互，在本文实现中，与持久化系统的交互只在CommitTX和AbortTX阶段发生，而与索引系统的交互以及向底层KV Store的写入是实时发生的，因此在CRUD API中，TX Manager会按照“读写索引系统——读写底层存储——本地记录持久化和回滚所需数据”三个主要阶段进行操作。

对于BeginTX接口，其只需要在事务管理系统中注册事务，因此与其他三个部分不存在交互。

对于CommitTX接口，由于CRUD操作已经实时写入底层存储，所以该接口分为“恢复索引系统——写持久化系统——事务管理系统更新”三个主要阶段。

对于AbortTX接口，调用该接口的事务产生的一切影响都应该被消除，因此不会产生持久化记录，但需要反向操作消除底层数据库中的记录（例如对于Create操作需要反向写入Delete），所以该接口分为“恢复索引系统——反向写入底层存储——事务管理系统更新”三个主要阶段。

### 3.6.2 开启事务

多版本存储系统中，使用BeginTX关键字手动开启一个新事务；为了保持版本的一致性，对于不在事务中的语句，按照每条语句自动开启一个事务的方式实现，保证每一个写入的值都带有多版本版本号，否则在事务开启情况下将不能读到缺失版本号的版本。

【此处是一个示意图，混杂了手动开启和自动开启的数条语句的事务TID】

开启事务需要为事务分配一个Transaction ID（以下简称TID），由于Nebula Graph支持分布式存储，所以出于实现的简洁性以及对分布式事务的扩展性支持，本文使用SnowFlake算法生成在每台机器上单调递增且在全局唯一的版本号作为TID。

【以下是本文的SnowFlake算法实现的TID的结构示意图，以及对于分布式场景下不同机器生成的TID的比较方式】

获得TID后，事务会向全局Transaction Manager注册，后者维护了活跃事务集以及全局索引系统中各个键的访问事务集。

### 3.6.3 读操作

读操作包括各类查询语句，例如查找一个顶点的某种类型的出边和入边，查找一个顶点或一条边的属性值。在MVTO协议下，读操作虽然不会影响底层KV store的数据，但会影响Index Map或VE Map中对应版本的read-ts项。

对于查找一个顶点的某种类型出边和入边的读操作，原生实现中利用了底层Rocks DB的前缀扫描功能，可以将SrcID/DstID、EdgeType、Rank等条件依次编码成前缀进行查找。但在多版本存储的实现下，底层存储中会存在大量有相同SrcID、DstID、EdgeType以及Rank的边，它们是同一条边在不同时间点生成的多版本属性的键，这会大大增加前缀扫描的时间和数据传输量，且还需要在获得边集后按照当前事务TID与每一条多版本边的版本号作比较，最终才能筛选出本次读操作的正确结果，整体流程是冗长且冗余量大的。

为了解决多版本实现引入的这一问题，同时能够优化这一类读操作使其不依赖底层存储提供的前缀扫描功能，本文在内存中维护了一个VE Map用来快速查找点的关联边，再在获得关联边后按照查询条件筛选出结果。这一实现的背后原理是，批量查询顶点的关联边与边的属性无关，而边本身的类型以及顺序等元信息是编码在边的唯一标识key中，而非写在底层数据库中的，所以可以直接将其维护在内存中，减少对底层存储的访问次数。

对于查找一个顶点或一条边的属性值，需要按照MVTO协议的要求访问Index Map中对应key的版本链，找到与当前事务匹配的版本后，更新该版本的read-ts为最新的读者TID，然后返回这一版本号，再根据版本号拼接多版本属性的键，从KV store中拿到对应的属性值。

在RC隔离级别下，读操作总是访问键值的最新公开版本，即使这个版本是由一个晚于当前事务时间戳的写事务创建提交的，因为按照语义要求，RC隔离级别总能读到所有的已提交版本（即公开版本）。在RR隔离级别下，读事务在第一次读操作时会生成逻辑意义上的ReadView，晚于ReadView生成时间的其他事务提交的内容是不可见的，因此本文中RR隔离下的读操作只会访问begin-ts小于当前事务的最新公开版本以实现可重复读。

对于可串行化隔离级别，Read操作先获取key对应的读锁，再访问Index Map找到key对应的版本链，并访问第一个版本，如果是一个可以读取的版本，则返回版本号，否则返回失败。

### 3.6.4 新建操作

新建操作为Insert Vertex以及Insert Edge，可以选择IF\_NOT\_EXISTS选项。IF\_NOT\_EXISTS的语义为只有当不存在这一顶点或边时才插入，否则不执行任何操作，是一种保证幂等性的操作。一般而言，设置了IF\_NOT\_EXISTS选项会耗费更长时间，因为系统需要首先进行一次读操作。

当设定了IF\_NOT\_EXISTS选项时，会先查询Index Map中是否有对应的key，如果查到了则不会进行后续操作，否则操作同未设定IF\_NOT\_EXISTS选项时的情况。

当未设定IF\_NOT\_EXISTS选项时，此时的写入操作是覆盖写。覆盖写的语义意味着在相应的事务提交后的所有对这一key的读取操作都不能访问到之前的版本，而在相应事务提交前，本次CREATE操作对其他事务是不可见的。但MVCC设计的目标之一就是允许迟到的读者读到原来的版本，因此在MVCC场景下，覆盖写的语义细化为——比使用了覆盖写的事务更古老的事务仍然可以访问到之前的版本，而比其创建时间更晚的事务则不能访问到之前的版本。

在除了本文选择的多版本实现方式以实现覆盖写之外，另一种常见的实现方式是使用RCU机制。RCU，即Read-Copy-Update，对于读操作不需要加锁，而对于写操作会复制一份原来的版本后在新的私有版本上更新，最后再替换原始的版本，替换时机可以选择版本上的读锁都被释放后进行替换。但由于RCU生成的版本链是一个事务私有的版本，并不允许其他事务访问，因此难以支持RU这样的需要读取未提交版本的隔离级别，同时如果事务提交时公有版本链上还有之前的事务的读锁，也无法完成版本链的替换工作，因此会阻塞事务提交，所以本文没有采用RCU方式。

在本文设计中，对边的插入操作需要同时记录在VE Map中。执行Insert Edge会在VE Map对应的Vertex中增加一条含有insert-edges的版本记录，并在事务内记录插入的Edge，使得提交或回滚时可以找到对应的Edge版本。

在可串行化隔离级别下，新建操作不会遍历版本链，而是获取写锁后找到第一个版本，修改版本号为当前事务的版本号。

### 3.6.5 删除操作

删除操作为Delete Vertex和Delete Edge，其中Delete Vertex可以设置WITH EDGE选项来删除顶点对应的所有出入边，效果等同于增加多次Delete Edge操作。同CREATE操作类似，删除操作要保证多版本下的读取一致性，即在删除操作对应的事务提交前的所有其他事务都能访问到这一key，提交后开启时间更晚的事务将无法访问这一key。删除操作会在版本链上新增一个版本代表此次操作，同时向底层存储删除对应的键值对。

在本文设计中，对边的删除操作需要同时记录在VE Map中。执行Delete Edge会在VE Map对应的Vertex中增加一条含有delete-edges的版本记录，并在事务内记录删除的Edge，使得提交或回滚时可以找到对应的Edge版本。

在可串行化隔离级别下，新建操作不会遍历版本链，而是获取写锁后找到第一个版本，修改版本号为0代表已删除。

### 3.6.6 更新操作

更新操作为Update Vertex和Update Edge，只能更改顶点或边的属性值，而不允许更改顶点或边本身。在多版本实现下，更新操作等同于新增一个版本并写入key对应的版本链后方，是一个Append-Only过程。如果Index Map中未能查到对应的key版本，认为这是一次错误的操作请求，返回错误信息；否则按照MVTO协议的要求获取最新版本号，访问KV store获得原始属性后，在Index Map中写入新版本，并将拼接出的新key和更新后的属性写入KV store。

UPDATE操作不影响点和边的关系，因此无需修改VE Map。

一条事务可能存在多次对同一个key进行的写操作，例如在第一条语句内设置key A的a属性，在第二条语句内设置key A的b属性。在非RU隔离级别下，其他事务不能读取本事务未提交的操作，因此可以生成两个version相同的版本挂在版本链上，直到事务提交时合并相同version的版本修改；也可以要求事务更新操作总是在原版本上更改，使得每条版本链中任意时刻最多只有一个相同的版本。本文在实现中采用了后者，这样可以保持版本链的单调性，保证一个事务对应一个版本。

在可串行化隔离级别下，更新操作与新建操作行为一致，获取写锁后直接修改第一个版本的版本号为最新。

### 3.6.7 提交事务

一条手动开启的事务使用CommitTX关键字或一条自动生成事务的语句正常结束时，会触发Transaction Manager的提交操作，提交操作主要分为三个阶段：修改Index Map和VE Map的对应版本为事务完成状态，生成读写操作的AOF entry并写入，以及完成Transaction Manager中的事务相关状态更新。

1. 修改Index Map和VE Map：在向版本链中写入新版本时，对新写入的版本设置了当前事务TID的写锁，这两个写锁需要在提交阶段释放。对于前述的CREATE和DELETE操作，会用UndoWriteSets中的新版本链代替原有版本链。
2. 生成AOF entry：每个事务都记录了读集合与写集合，以此生成Modify、Add和Delete三种类型的对于内存索引的操作记录，再批量写入AOF buffer中，实现Index Map和VE Map的持久化。
3. 事务状态更新：已提交的事务会从活跃事务集中移除，而对于Transaction Manager中维护的读事务集合队列，会进行标记删除。如果当前事务是某一key的读事务集合队列队首，表明这是现存访问该key的最古老事务，不会受到其他事务回滚的影响，因此可以直接删除。

【此处应为流程图】

### 3.6.8 回滚事务

在事务执行过程中，如果因为系统执行出错或多版本冲突而导致执行某一步失败时，会触发回滚机制；对手动开启的事务，也支持使用AbortTX关键字主动放弃事务触发回滚。回滚的事务不会产生AOF entry，但需要恢复Index Map以及VE Map为原始状态，并撤销对底层KV store的数据修改，因此回滚操作也主要分为三个阶段：恢复Index Map和VE Map，反向写入数据以抵消对底层KV store的影响，完成Transaction Manager中的事务相关状态更新。

1. 恢复Index Map和VE Map

分析上述读写操作对Map的影响，依次按照事务内部记录的读集合和写集合进行回滚。对写集合中的每一项，针对增加版本的操作记录，找到对应的版本后在版本链中标记删除，并释放原来版本的写锁。对读集合中的每一项，修改Index Map中相应版本的read-ts为撤销本次事务后应有的值：遍历Transaction Manager维护的读事务集合队列，计算回滚事务之后访问当前key的所有读事务的最大值，并将read-ts更新为这个最大值，再生成一条Modify类型的AOF entry写入AOF中。

【举例说明：1、3、8三个事务依次读同一key，若都提交成功则会生成M1，M2，M5三条AOF，若3回滚，TID=3<BehindMax=8，只有1和8生成了M1和M5，为保证一致性，delta=AbortDelta=2，需要生成一条M2；1、8、3三个事务依次读，生成M1和M7，若8回滚，则read-ts变为3，TID=8>BehindMax=3，delta=BehindMax-(TID-AbortDelta)=3-(8-7)=2，需要生成M2这条AOF；3、8、1三个事务依次读，生成M3和M5，若8回滚，read-ts变为3，此时TID=8>(TID-AbortDelta)=8-5=3>BehindMax=1，无需操作；】

读事务回滚的read-ts变化计算方法如下。

使用三个量tx.TID、BehindMax、BeforeTs来分类讨论，tx为当前回滚事务，BehindMax是当前事务之后的所有事务的最大TID，BeforeTs为当前事务读取前的read-ts值，BeforeTs=tx.TID-tx.delta，则读集合中记录的事务总是满足tx.TID>BeforeTs，所以有以下三种情况：

(1) BeforeTs>BehindMax 此时无需进行额外操作；

(2) tx.TID>BehindMax>BeforeTs 此时delta=BehindMax-BeforeTs，生成一条M{delta}的AOF entry；

(3) BehindMax>tx.TID 此时delta=tx.delta，生成一条M{delta}的AOF entry。

对于同一事务多次读取同一版本的键的情况，假设事务为2和3，读取序列为R2R3R2，则只在第一次读取时将事务ID记录在全局读集合内，例如此时应该记录为{2,3}。可以证明只记录首次读取行为能够满足read-ts的回滚和索引重建要求：一个待回滚事务的读行为记录一次或多次的情况下，其回滚后的read-ts值都是整个读集合中除去这一事务后的最大TID；事务后续对同一版本的键的读取不会导致read-ts修改，因此从索引重建和生成AOF条目的角度看待，后续的读取行为是没有影响的。

【此处是上面举例的图示】

1. 反向写入数据

由于在事务产生的写操作的过程中，已经将数据写入了底层KV store，因此为了撤销这些操作的影响，本文采用反向写入方式，即对put操作反向写入同一key的remove操作，对remove操作则根据事务内记录的操作集合反向写入同一key的put操作，以达到恢复的效果。

1. 事务状态更新

从活跃事务集中移除回滚的事务，并在读事务集合队列中标记删除对应的事务TID，因为该事务对read-ts的影响已经在第1步中回滚了，此后可以忽略。

# 持久化设计

## LRM系统设计

LRM系统，即本地恢复管理（local recovery manager）系统，其核心是将读写操作对Index Map和VE Map的修改进行编码以及落盘，用来保证故障发生后的修改操作恢复。LRM系统的核心为AOF Manager，即仅追加文件（append-only file）管理器，AOF是一个存储在磁盘上的文件，只允许以追加方式写入数据，其记录了对内存中的MVCC索引的变更信息。对每一条写入AOF的信息进行编码，称为AOF entry，一个entry内包含了索引的键、修改操作以及修改的值等信息，AOF entry进行序列化后直接调用文件系统API写入磁盘。在本文的实现中，LRM系统包含了一个AOF Buffer，其会按照记录的条数以及时间定时写入磁盘（例如每1000条操作记录或每1秒进行依次落盘）。由于使用了后文叙述的无锁队列实现，即使只使用一个buffer，在将buffer内容写入磁盘文件的同时也支持继续写入AOF Buffer，因此本文的实现在高并发场景下并不会因为磁盘IO导致阻塞。

在本文实现中，仅在事务Commit或Abort时才写入AOF，为了提高性能，此处将同一事务的所有读写操作批量编码后再传给AOF Manager，即每次调用WriteAOF函数时都会传入这一事务的所有读写操作。事务Abort时需要进行回滚，按照第三章介绍的回滚事务时的具体操作流程生成相应的AOF entry后传给AOF Manager。

由于AOF的append-only性质，原始文件会随系统运行而越来越大，因此需要定时进行重写。AOF文件可重写的原理为，其实际用处为重启后恢复内存中的Index Map和VE Map结构，恢复每一个key的版本链，因此并不关心过程中各事务读写key的顺序和时间；同时，在GC定期运行的情况下，Index Map会实际删去众多的版本甚至key，因此如果遇见Delete操作，则可以将之前对同一{key, version}的所有AOF条目都舍弃。重写机制实际上是进行了一次AOF文件的读入和Index Map、VE Map的重建，然后将重建后的Index Map和VE Map以Add操作的方式写入新的AOF文件后替换旧文件。

## Index Map持久化写入

持久化写入的核心实现为WriteAOF，是将每个事务的操作集合批量写入磁盘文件进行存储的操作。

首先讨论编码方式：将Index Map相关的三种操作分别记为修改（modify）、增加（add）、删除（delete）（下文使用M、A和D分别代表修改、增加和删除三个操作），则A需要记录操作的key和version，D只需记录操作的key，M需要记录操作的key、version以及变更值delta。也即是说，M包含operation、key、version、delta四个域，A包含operation、key、version三个域，D只包含operation、key两个域。此处将分别列举出三种操作可能出现的场景。【此处是一个表格表示Index Map AOF的三种编码】

1. Modify：在本文实现中，M操作仅会修改指定版本的read-ts字段，因此只会出现在含有读操作的事务提交或回滚时，且不需要额外记录修改的字段名称。每条M-AOF记录只指定一个version。
2. Add：A操作会在Index Map上插入新的版本，因此出现在UPDATE、CREATE这两种包含写操作的事务提交时。由于添加新版本必须分别指定每一个版本的版本号，所以每条A-AOF记录只指定一个version。
3. Delete：Delete操作可能出现在两种场景下：其一是写操作删除了一个键或使用覆盖写的方式新增了一个键，此时是批量删除一个key的所有版本，每条D-AOF记录指定一个key的所有版本；其二是事务回滚时其中的写操作对应的版本被标记删除以及GC遍历版本链时对过期版本的删除，由于事务回滚时的写操作还未生成AOF，而GC移除过期版本时不必依次生成删除日志，只需要先生成一条删除全部版本链的AOF，再依次生成保留下来的版本的A操作日志即可。

其次讨论核心AOF Buffer的读写实现。该场景是一个MPSC（multi-producer-single-consumer）场景，不同的事务可以并发调用WriteAOF方法来写入自己的操作集，而仅有一个消费线程负责将AOF Buffer中的内容写入磁盘。因此，一种实现方式是使用rust语言内置的mpsc::channel或mpsc::sync\_channel类型，使用FFI方式封装出接口供CPP程序调用，具体实现思路如下：

【此处是一个流程图，描述rust使用这样的channel时的流程】

在这种实现之外，重新审视这一场景，可能的并发问题出现在两个事务的读写事务集有重合部分时，以下进行分类讨论。

定义事务A与事务B为相关联事务，当且仅当A与B的持续时间有重合且A与B存在对同一key的访问。用TA表示A的TID，TB表示B的TID。

1. 读-读并发

场景(1)：不失一般性，设原时间戳为O，且O<TA<TB，A事务先于B事务读到指定key的version，则AOF entry为(M, key, ver, TA-O)与(M, key, ver, TB-TA)，合并重写后的AOF entry为(M, key, ver, TB-O)，因此顺序无影响。

场景(2)：O<TA<TB，B事务先于A读，则仅有B事务产生一条AOF entry (M, key, ver, TB-O)，此时在AOF处无并发。

1. 写-读并发

不失一般性，假设事务A对key进行写入操作，事务B对key进行读取操作。若A先到达key的链表处，由于非删除的写操作总是基于最新版本进行修改，因此会锁定当前最新版本，这将导致随后到来的事务B对最新版本的读失败，B将不会Commit，因此无AOF处的并发。对于删除操作，A产生AOF entry (D, key, 0)，B产生一条AOF entry (M, key, ver, delta)，只要在读取AOF文件重建Index Map时忽略对不存在的version进行修改的操作即可保证结果一致性，因此也不会冲突。

1. 读-写并发

不失一般性，假设事务A对key进行写入操作，事务B对key进行读取操作。若O<TA<TB，B事务先进行读操作，修改最新版本的read-ts为TB，随后A事务到来找到最新版本ver，根据MVTO协议要求，由于TA<ver.read-ts，所以A不能修改当前key，需要换一个更大的时间戳，A将不会Commit，无AOF处的并发。若O<TB<TA，将产生(M, key, ver, TB-O) 和 (A, key, newVer) 两条AOF entry，对应在Index Map复原操作上为对{key, ver}的修改和在key链表上增加newVer，其相对先后顺序不会影响最后的结果，因此也没有AOF处的并发影响。对于删除操作的讨论同上一条，不会发生冲突。

1. 写-写并发

在MVTO要求下，对同一key有写操作的相关联事务最多只能有一个Commit，因此不存在A与B为相关联事务且都修改了同一key的情形，该场景下无AOF并发冲突。

综上所述，在AOF entry写入场景中，相关联事务并不会产生并发冲突，任意的写入序列以任意顺序串行写入后，最终能复原出相同的Index Map。因此，另一种实现方式是使用基于CPP原生实现的并发安全队列，忽略不同事务对AOF的写入顺序直接串行化。本文选择了开源库MoodyCamel::ConcurrentQueue（github.com/cameron314/concurrentqueue），使用ConcurrentQueue可以构造一个多生产者多消费者模型，并支持批量写入和批量读出。在此种实现方式下，每个事务提交时会将自己的所有读写AOF entry批量入队，负责落盘的线程则定期将队列中所有元素出队后追加写入磁盘文件。

对于可串行化隔离级别，为了保证其生成的AOF文件与其他隔离级别兼容，Read操作不生成AOF entry，Create生成(D, key)和(A, key, version)两条AOF entry，Update生成(A, key, version)，Delete生成(D, key)即可。读入AOF文件恢复Index Map时，忽略M操作，并只在版本链的第一个版本上进行修改操作。

## VE Map持久化写入

VE Map以差分版本链形式记录了顶点与其出入边在不同版本上的关系，在事务提交时会根据事务内部的insert-edges队列以及delete-edges队列生成对应的AOF条目。

仅有Insert Edge以及Delete Edge两种写操作会生成VE Map的AOF条目，在本文使用的MVTO协议实现中，写-写并发操作是被禁止的，所以含有相同源顶点或含有相同目的顶点的边的并发修改也是被禁止的，即任意并发到达的AOF条目包含的边集无重叠，所以可以按照任意顺序串行写入AOF文件。此处使用和上一小节相同的ConcurrentQueue，在另一个AOF文件中写入VE Map的操作记录。【此处是一个表格表示VE Map的两种编码方式】由于本文对点边关系维护了一份出边索引和一份入边索引，因此需要在VEAOF中标记出边的方向或将出边与入边的AOF存在两份文件中。为了减少读写磁盘次数，本文选择在VEAOF结构中增加一个字段表示边的方向，而由于nebula graph中的边的编码方式决定了当Edge Rank>0时为出边，Edge Rank<0时为入边，因此后期可以优化节省这一字段，改为读取时从边信息中获取出入边类型。

【TODO 此处可以优化，将边进行编码，多维护一个索引，使得条目内部能够使用位图的方式存储edges】

## 重启恢复

重启恢复的目的是根据磁盘中的AOF文件，恢复Index Map和VE Map结构，二者步骤相似，下文以恢复Index Map为例进行说明。目前只提供了在storage服务启动时自动读取磁盘AOF文件进行Index Map恢复的功能，未来也可以提供手动调用的API。

首先，调用ReadAOF将磁盘中的AOF文件转化为AOF entry序列，再对AOF entry进行解码，获得每一条entry对应的Index Map的改动，在此基础上对Index Map进行相应的modify、add以及delete操作。

在恢复Index Map结构前，storage系统是不可用的，状态status为false，直到完成了复原后才将状态status修改为true，允许事务执行。

出现Delete操作时，由于恢复过程是单线程的，且希望重建的Index Map内存占用尽可能小，因此对特定version的删除不再采用标记删除而是直接删除；对于指向key的整体删除，为了减少map中的键值对重复插入和删除的性能消耗，采取只删除key指向的链表的方式，在恢复过程中保留key不变。因此，当重建完成后，新的Index Map结构可能存在多余的key——一个key的所有版本都已经被删除了，此时应该遍历Index Map，移除所有指向空链表的key。

对于可串行化隔离级别，读取VE Map不会生成AOF条目，每次写入或删除边的事务结束后都会产生删除操作删除当前版本链上所有版本的AOF entry以及插入新版本操作，新版本会将可串行化状态下的所有边集合写在insert-edges中。在读取VEAOF文件恢复点边关系索引时，忽略修改read-ts的Modify操作，对于Add和Delete操作直接在当前版本上更新边集合。

## AOF重写

AOF重写的目的是压缩AOF文件大小，并精简操作步骤，使得重启恢复时只做必要的操作，节省恢复时间。

对于Index Map，一次Delete操作会导致之前的所有修改被清空，而多次Modify操作也可合并成一个，因此其AOF文件重写可以有效压缩体积，达到减小文件体积和内存索引复杂度的结果。

对于点边关系索引VE Map，本文维护了顶点对应的出入边集合，采用了差分版本存储的方式存储点边关系，每次读取时都需要顺序从头开始读，因此减少版本链的长度有利于读取和写入性能。但在点边关系中，Add和Delete操作都只会影响多版本中的值，不会影响索引中的键，因此无法按照Index Map的方式进行合并压缩。

但对于任一差分版本链上的公共版本，如果该版本的创建时间已经不在活跃事务集内，则说明任何之后的事务都不会读取到该版本之前的版本，因此可以将这个版本之前的版本与其合并，达到减小版本链长度和目的。在4.4节重启恢复过程中，由于当前活跃事务集为空，因此所有的版本都可以被合并，这将使得重启恢复后的VE Map中每一个顶点都只包含一个版本。由于本文实现的GC算法也可以合并过时版本，所以在此处直接调用GC即可。【TODO优化是重建索引时进行合并】

AOF重写采用定时执行与文件大小相结合的方式，达到设定事件或者AOF文件体积达到设定大小后就会自动进行，也可以手动调用（类似于数据压缩）。在重写过程中产生的新的AOF写入请求会被写入一个暂存文件，待重写结束后将暂存文件的内容添加在新AOF文件之后，再将新文件重命名后替代旧文件。

AOF重写分为两个主要阶段：读入和写回，其中读入阶段的操作与上一小节重启恢复相同，会在内存中重建不包含冗余key的精简索引，此后遍历内存索引，依次生成Add、Modify和Delete操作的AOF entry并写入新文件，写入完成后将新文件重命名为原始文件名以达到覆盖效果。

但由于读入文件重建索引阶段会将系统设置为不可用，这可能造成类似GC的“Stop the World”现象，所以此处有一些小的改动，提高了系统的可用性。本文参考了经典的内存键值数据库Redis的持久化设计思路，在AOF重写开始时将新产生的AOF条目写入另一个临时文件中，并使用一个临时索引读入原始文件重建索引；当索引重建完毕时，使用一个互斥锁，加锁设置系统为不可用状态，读入临时文件中的AOF条目在新的索引上，并用新的索引替换原索引，然后将临时文件中的数据添加至重写后的AOF文件中并替换原AOF文件，解锁恢复系统可用。这样的实现中，系统不可用时间等于重建索引时间内的AOF条目读入+文件追加写操作，在AOF文件较大时能显著缩短不可用时间。

Modify操作的合并：对于一个版本，若其read-ts不等于begin-ts，说明这个版本在生成后被后来的事务读过，因此拆分成Add{begin-ts}和Modify{read-ts – begin-ts}两条记录依次写入，这就起到了合并多次Modify操作的效果。

Delete操作的合并：对于同一个键，一次Delete操作将会清空整个版本链，因此之前对这一个键的所有修改操作的AOF都可以被省略。

# 实验与分析

本部分对前面两章设计的MVCC改动进行了实现，并编写一系列样例进行实验验证来证明本系统设计的正确性，实验样例分为三个部分。

实验样例的第一部分包括在读已提交和可重复读两种不同隔离级别下的基础CRUD操作，因为本文基于的MVTO协议原始并不支持读未提交隔离级别——MVCC要求读操作总是读到合适的指定版本，而读未提交总是读最新版本，而本文的实现中使用了RCU机制，此时的最新版本是其余事务线程私有内容，不支持被读取。实际上，在MVCC基础上，推迟带有脏读的事务的提交直到其读集合中的所有版本都被提交可以避免脏读的发生，等效于读已提交隔离级别。对于可串行化隔离级别，本文使用了非阻塞读写锁的方式，因此会导致事务中获取锁不成功就立即回滚，也不适合本部分的验证。因此，本部分只覆盖了RC和RR两个隔离级别的实验验证，涵盖了TX Manager提供的7个API的使用。

实验样例的第二部分包含对持久化系统的验证以及对垃圾回收算法实现的验证，持久化系统验证包括AOF写入、AOF读取以及AOF重写。

实验样例的第三部分包含nGQL图遍历语句的实验支持以及第二章讨论的并发场景下的异常现象验证，使用实验结果来证明本文理论分析部分的正确性。

在实验部分，为了观察实验结果，本文还实现了一系列辅助Debug API，包含PrintIndex、PrintVE、PrintRS、Min、Max等，它们分别对应着打印前文系统设计中的内存结构以及获取当前活跃事务集范围等操作。这一系列API在三个部分实验中都被使用，不再单独说明。

表5.1 本章实验名称及编号

|  |  |  |  |
| --- | --- | --- | --- |
| 实验部分 | 实验名称 | 实验编号 | 实验指令 |
| 第一部分 | 单事务CRUD | 1、2 | Create、Read、Update、Delete（以下称为CRUD）；  BeginTX、Commit、Abort |
| 多事务CRUD（两种隔离级别） | 3、4 |
| 事务回滚 | 5~9 |
| 时序验证 | 10、11 |
| 顶点锁 | 12 |
| 第二部分 | Index AOF写入与读取 | 1~6 | CRUD；  Rebuilt、Rewrite、GC |
| VE AOF写入与读取 | 7、8 |
| AOF重写 | 9、10、11 |
| GC | 12、13 |
| 第三部分 | 并发异常现象 | 1、2、3 | CRUD |
| nGQL | 4 | GetNeighbors |

## 实验环境

实验代码使用cpp编写，编译器为gcc 9.4.0，在wsl2：ubuntu-20.04环境下使用c++17标准编译和运行，编译工具为Cmake 3.16.3和GNU Make 4.2.1，所有的实验用例通过bash输入和解析。

实验使用的底层KVStore为Facebook开源RocksDB的本地编译版，版本为8.1.1，源代码的GitHub地址为<https://github.com/facebook/rocksdb/tree/v8.1.1>，在上述环境下编译为动态库使用。

## 第一部分

第一部分为基础CRUD API实验，包含在不同隔离级别下对本文实现的MVCC CRUD行为验证以及事务回滚的验证。同时，这一部分还对前文提出的顶点锁进行了实验验证。

这一部分涉及的API为Register、Commit、Abort、Read、Create、Update、Delete。

为了验证不同隔离级别下的读写行为，本部分为不同隔离级别分别编译了可执行文件产物，因此最终产物为RC、RR和Serializable三个，在本部分的一些实验中会分别列出三种隔离级别下的指令执行结果。

### 5.2.1 单事务CRUD

第一个实验验证注册事务、提交事务和回滚事务在全局TX Manager中的表现，以下是操作序列，括号内为输出结果。Min输出当前最小活跃事务TID，Max输出当前最大活跃事务TID，Reg [TID]表示注册事务TID。

Reg 1；Min（1）；Max（1）；Reg 5；Reg 2；Min（1）；Max（5）；Commit 1；Min（2）；Max（5）；Abort 2；Reg 3；Min（3）；Max（5）；Commit 5；Commit 3。

因此，Commit和Abort都会将当前事务移出活跃事务集。

第二个实验验证单个事物的CRUD操作正确性。以下是事务1的一个访问序列，操作的结果写在操作指令右侧，写操作的执行状态用true和false表示，读操作返回读到的值，如果没有独到则用/表示。

Create (key,value,ifNotExist) 表示创建(key,value)，若第三个参数为true则key存在时不会覆盖写入新的值。Update (key,newValue) 表示将key的值改为newValue。Delete (key)会删除key以及对应的值。Read (key) 返回事务读到的key对应的值，如果读取失败或没有可读取的值则返回空。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg 1 | true | Update (v1,111) | true | Read (v1) | 1234 |
| Create (v1,123) | true | Read (v1) | 111 | Create (v1,2345) | true |
| Create (v2,234) | true | Delete (v1) | true | Read (v1) | 2345 |
| Read (v1) | 123 | Read (v1) | / | Create (v1,3456,true) | true |
| Read (v2) | 234 | Create (v1,1234) | true | Read (v1) | 2345 |

可见，单事务情况下，能读取到最新的写入操作情况，而如果指定了IfNotExist，将不会发生覆盖写，否则默认的Create操作是覆盖写操作。

### 5.2.2 多事务CRUD+RC

实验三验证了RC隔离级别下Create和Read的表现情况。可以看出，在读已提交隔离级别下，事务3发生了不可重复读现象，两次Read的值不同。

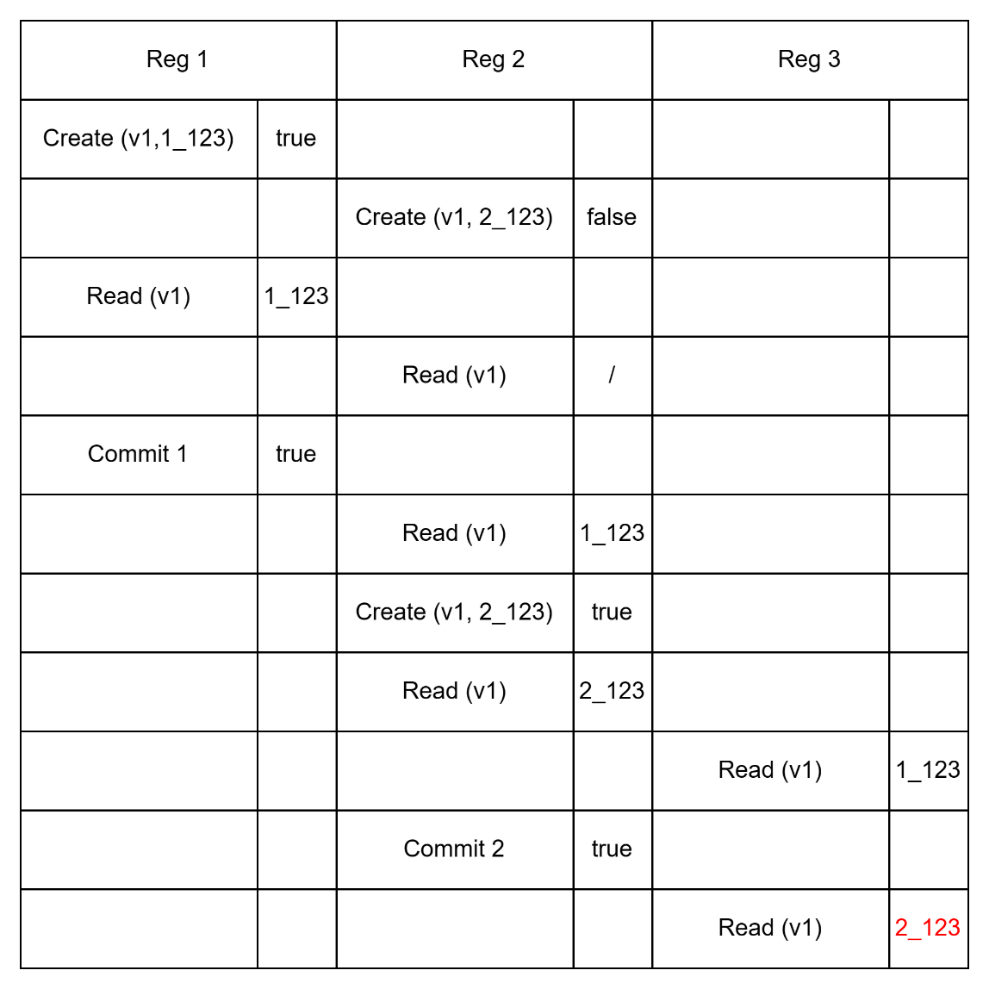


图5.1 第一部分-Create-RC

实验四验证了RC隔离级别下Update和Delete操作的表现，可以发现没有提交的事务的修改不会被其他事务读到，而一旦提交了，对应的更新和删除操作的修改就会被读到了。事务4发生了两次不可重复读，分别读到了事务2更新的值和事务3删除后的空值。

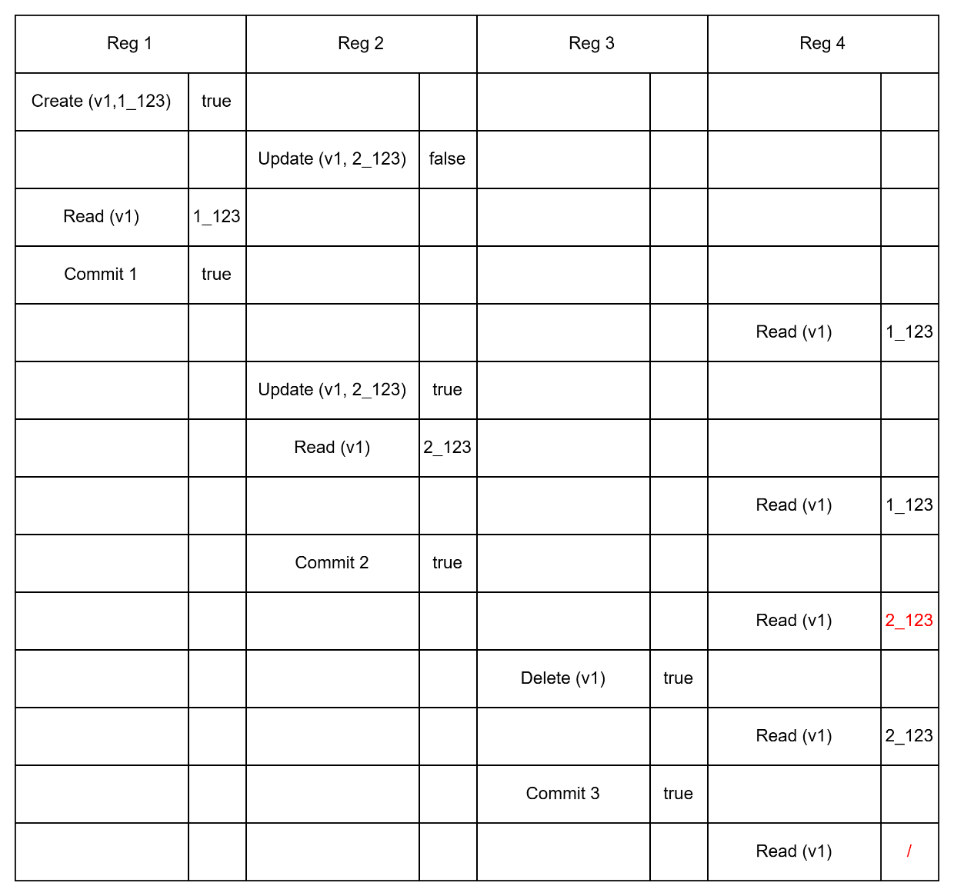


图5.2 第一部分-Update和Delete-RC

### 5.2.3 多事务CRUD+RR

实验三验证了RR隔离级别下Create和Read的表现情况。可以看出，在可重复读隔离级别下，事务3没有发生不可重复读现象，两次Read的值是相同的。

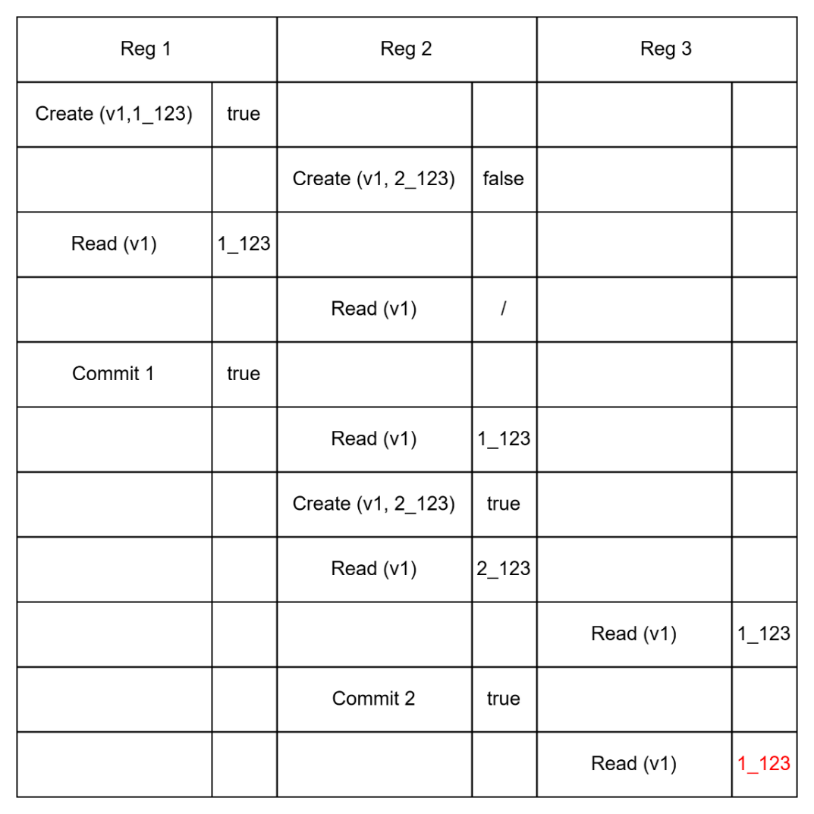


图5.3 第一部分-Create-RR

实验四验证了RR隔离级别下Update和Delete操作的表现，由于本文实现的RR只会读取开启时间小于等于自身事务ID的版本，因此后续的更新和删除操作不会影响可重复读的值。下图中事务2更新了v1的值，事务3删除了v1，但读事务4的Read View建立在事务1提交后事务2更新前，所以读取的值与第一次读取保持不变。



图5.4 第一部分-Update和Delete-RR

### 5.2.4 事务回滚

事务回滚包含了单事务回滚和多事务回滚的实验，其中单事务回滚又包括了Create/Update操作的回滚和Delete操作的回滚，以及Read操作导致的版本read-ts变更的回滚实验验证。

实验五验证了含有Create和Update操作的事务回滚情况。PrintIndex会打印内存中的Index Map情况。事务2在创建v2和更新v1的值后回滚，之后事务3仍然可以正常修改v1的值，事务4尝试读v2的值但未空，即回滚后的事务对全局没有宏观影响。

①为{v1:[{write-lock:0, read-ts:3, begin-ts:1}, {write-lock:0, read-ts:2, begin-ts:-2}]},

v2:[{write-lock:0, read-ts:2, begin-ts:-2}]}

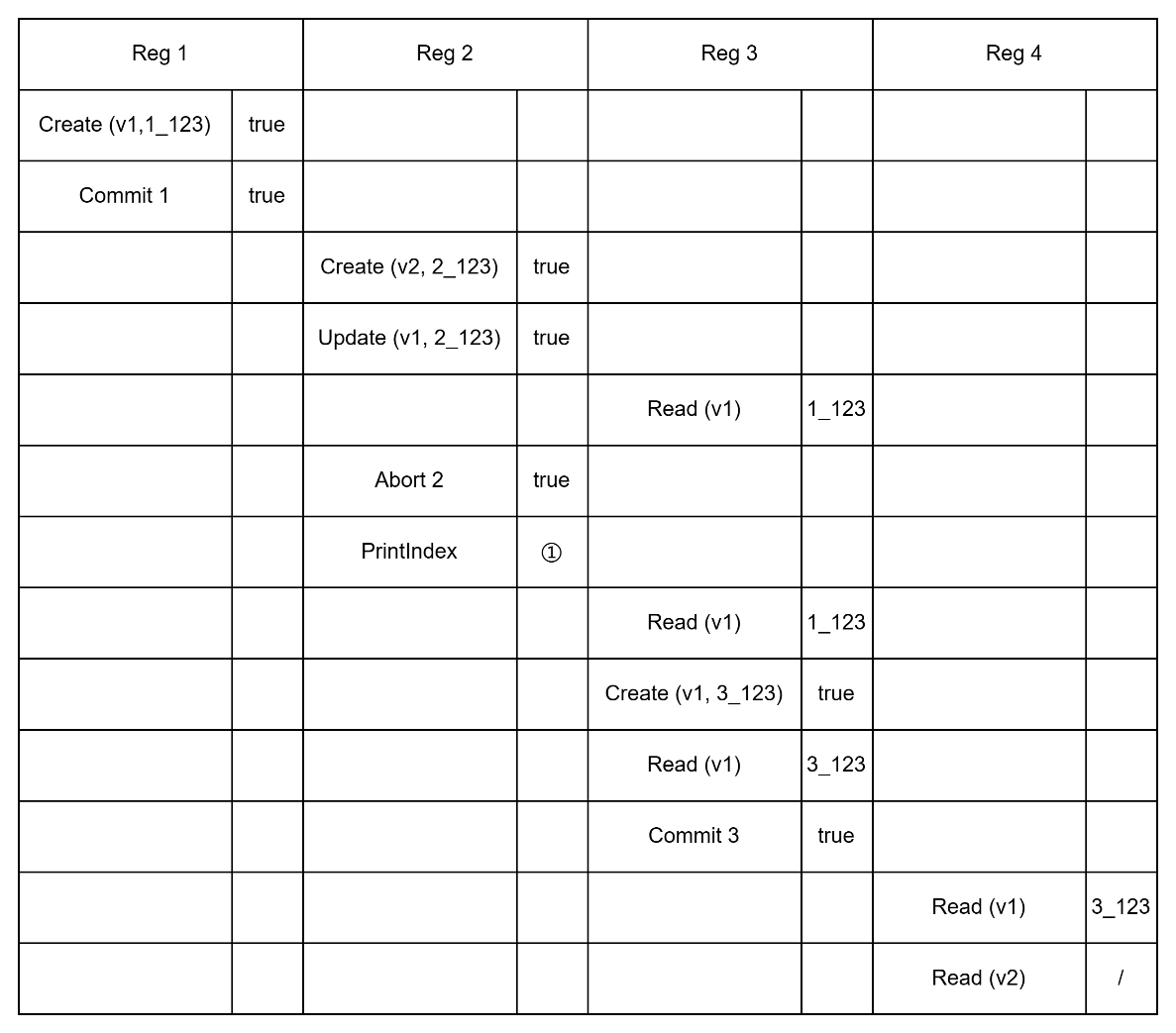


图5.5 第一部分-Create和Update回滚

实验六验证了含有Delete操作的事务回滚情况。事务2在删除v1后回滚，之后事务3仍然可以正常读和修改v1的值，即回滚后的事务对全局没有宏观影响。

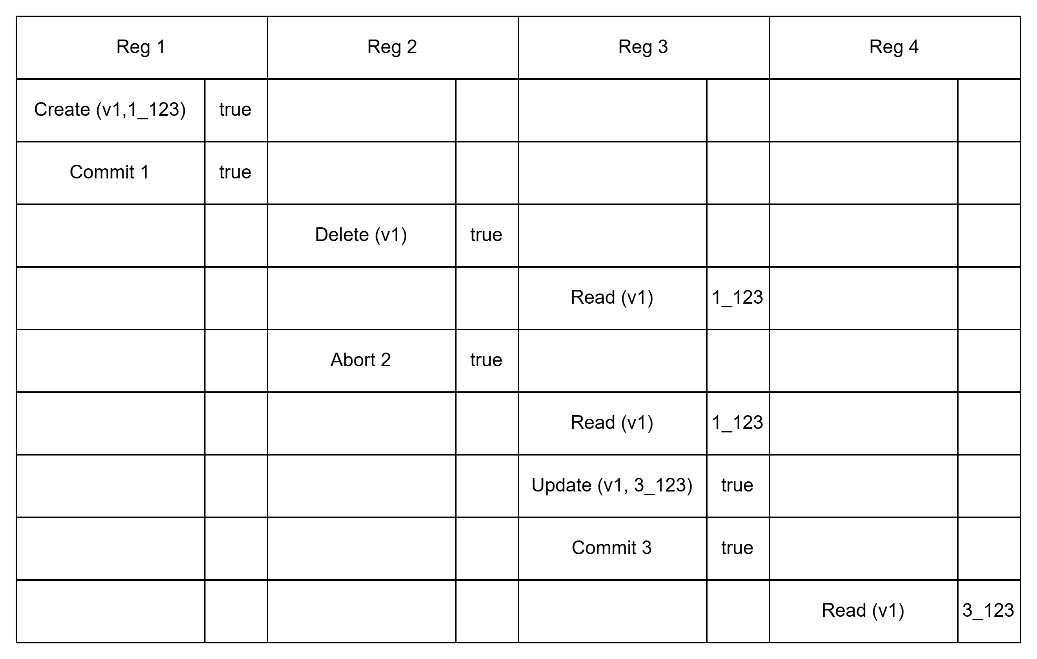


图5.6 第一部分-Delete回滚

实验七验证了含有Read操作的事务回滚情况——旧事务回滚。事务1创建了v1和v2两个key，事务2和3都是只读事务，按照r2(v1)、r3(v1)、r3(v2)、r2(v2)的顺序读取，此后事务2回滚，此时由于v1和v2的最新读取事务时间戳都是3，所以read-ts不会回退。而之后再让事务4读取v1，发现v1的read-ts更新为4，而v2的保持3不变。

①为{v1:[{write-lock:0, read-ts:3, begin-ts:1}], v2: [{write-lock:0, read-ts:3, begin-ts:1}]}

②为{v1:[{write-lock:0, read-ts:3, begin-ts:1}], v2: [{write-lock:0, read-ts:3, begin-ts:1}]}

③为{v1:[{write-lock:0, read-ts:4, begin-ts:1}], v2: [{write-lock:0, read-ts:3, begin-ts:1}]}

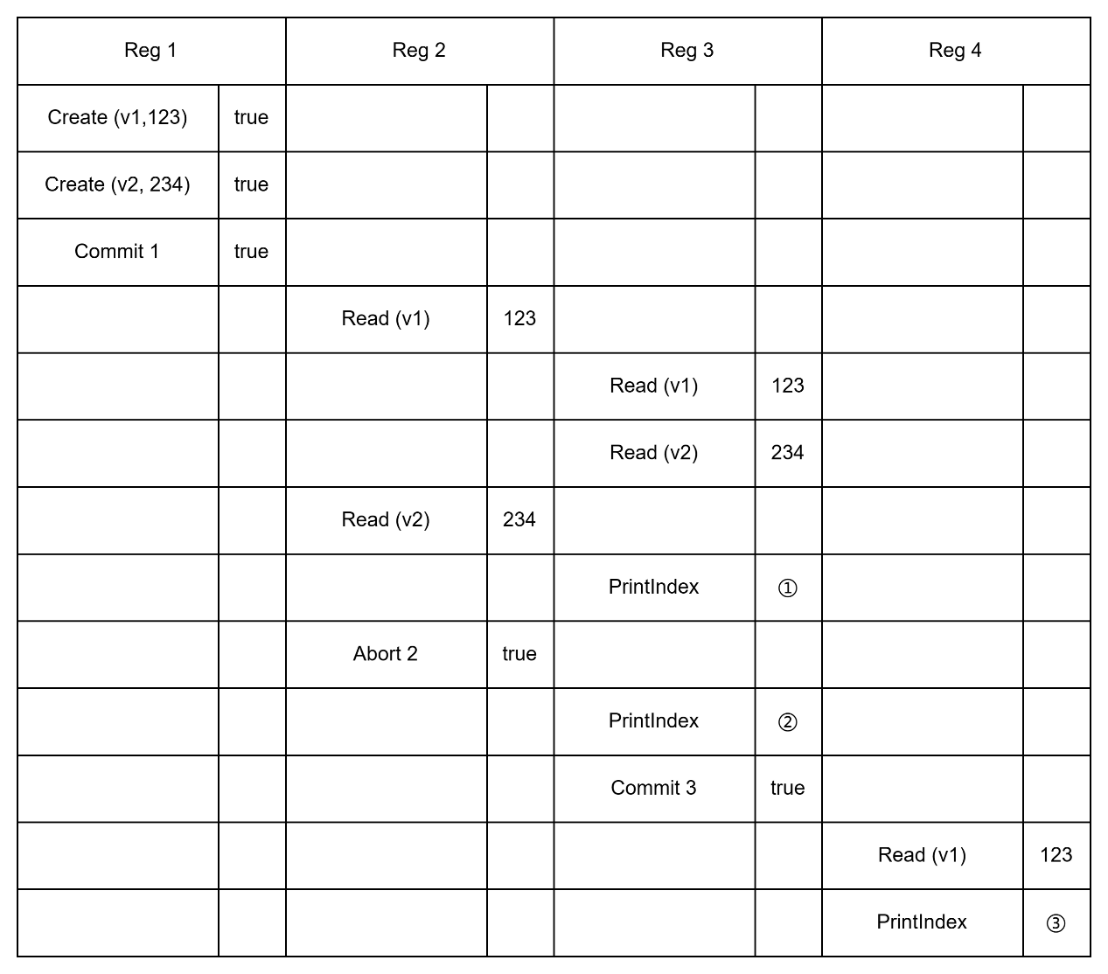


图5.7 第一部分-Read回滚-旧事务回滚

实验八验证了含有Read操作的事务回滚情况——新事务回滚。事务1创建了v1和v2两个key，事务2和3都是只读事务，按照r2(v1)、r3(v1)、r3(v2)、r2(v2)的顺序读取，此后事务3回滚，此时v1和v2的最新读时间戳回滚为2。之后事务4读v1，会将v1的时间戳更新到4，v2保持不变。

①为{v1:[{write-lock:0, read-ts:3, begin-ts:1}], v2: [{write-lock:0, read-ts:3, begin-ts:1}]}

②为{v1:[{write-lock:0, read-ts:2, begin-ts:1}], v2: [{write-lock:0, read-ts:2, begin-ts:1}]}

③为{v1:[{write-lock:0, read-ts:4, begin-ts:1}], v2: [{write-lock:0, read-ts:2, begin-ts:1}]}

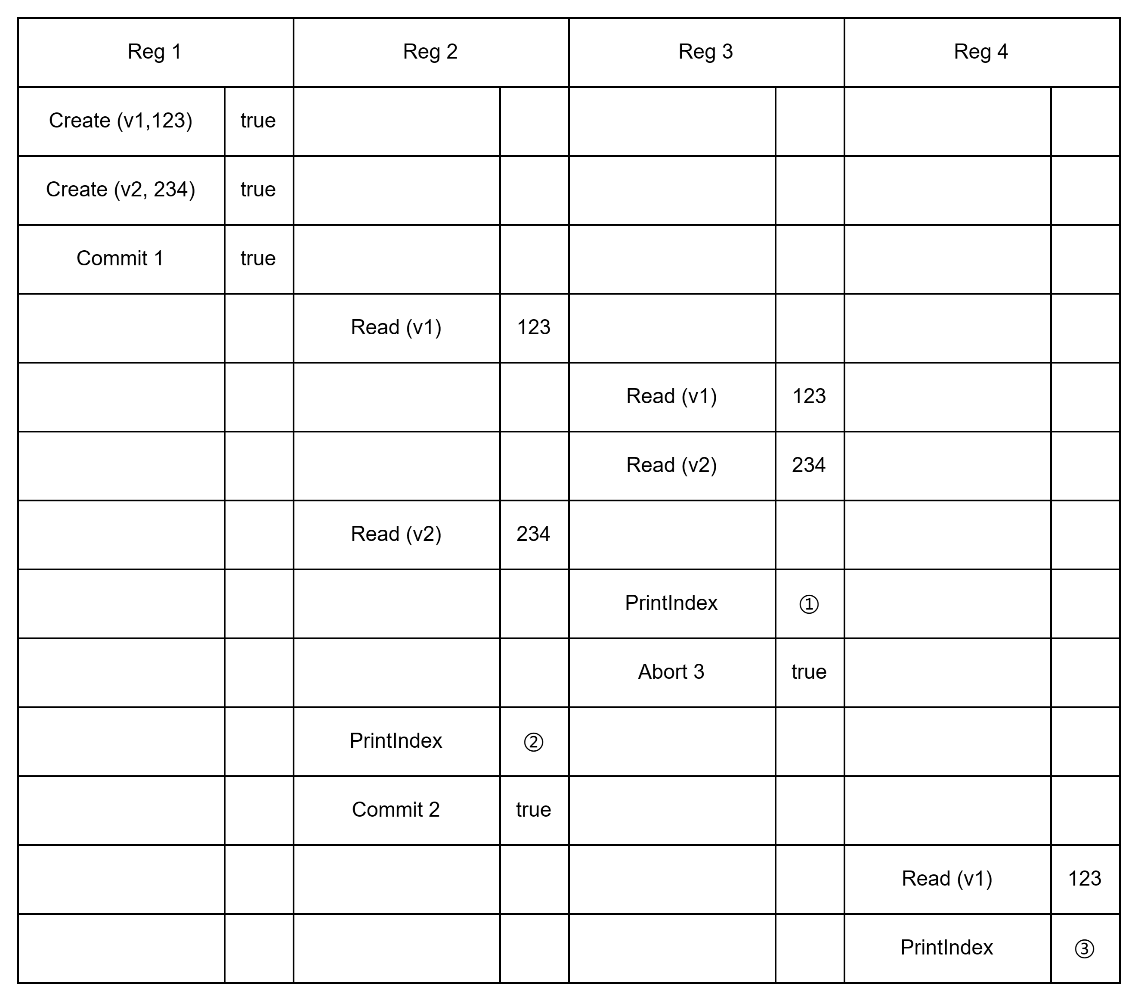


图5.8 第一部分-Read回滚-新事务回滚

实验九是多事务混合回滚的实验，事务行为：事务1创建v1并提交，事务2更新v1并创建v2后提交，事务3读取v1后更新v1并删除v2但回滚，事务4读取v1和v2并创建v3后回滚，事务5读取v1、v2、v3后提交。PrintRS为打印全局的读集合，会将读了每个key的事务按照读顺序打印出来，用于验证事务回滚时必须要在确保其他事务提交或回滚不再用到当前事务的TID后才能从读集合中删除。

①为{(v1,2):(3,false), (v2, 2): null}

②为{v1: [{write-lock:0, read-ts:1, begin-ts:1}, {write-lock:0, read-ts:3, begin-ts:2}, {write-lock:3, read-ts:3, begin-ts:3}], v2: [{write-lock:0, read-ts:2, begin-ts:2}, {write-lock:3, read-ts:3, begin-ts:3], v3: [{write-lock:0, read-ts:4, begin-ts:-4]}

③为{(v1,2):(5,false), (v2, 2): (5,false)}

④为{v1: [{write-lock:0, read-ts:1, begin-ts:1}, {write-lock:0, read-ts:5, begin-ts:2}, {write-lock:0, read-ts:3, begin-ts:-3}], v2: [{write-lock:0, read-ts:5, begin-ts:2}, {write-lock:3, read-ts:3, begin-ts:-3], v3: [{write-lock:0, read-ts:4, begin-ts:-4]}

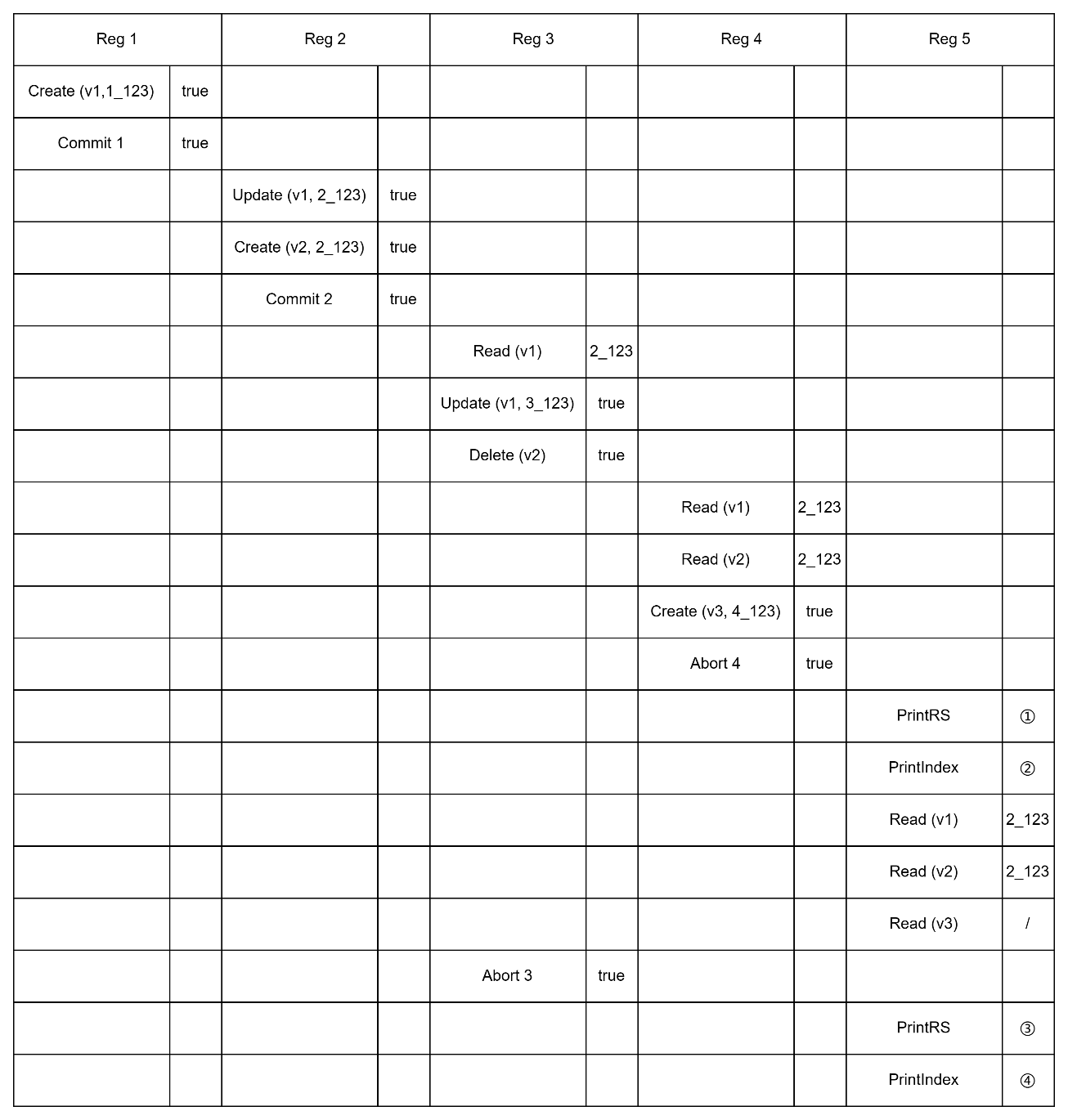


图5.9 第一部分-多事务回滚

### 5.2.5 时序验证

在读写并发场景下，存在“迟到的读者”（read too late）与“迟到的写者”（write too late）两种情况。迟到的读者指一个开启得更早的读事务晚于开启得更晚的写事务到达，在未使用MVCC情况下，由于它会读到一个本不应该读到的新值，所以会被拒绝读取，要求读事务换一个更大的TID再来读取；但在多版本场景下，允许其读到符合自身ID的老版本。迟到的写者指一个开启得更早的写事务晚于开启得更晚的读事务到达，由于写总是在最新版本上更新，因此按照MVTO要求应该拒绝写入，要求写事务换一个更大的ID后再写入。

实验十为迟到读者，事务2为读事务，事务3为写事务。事务3先更新v1的值，但之后事务2仍然可以读到v1原来的值。

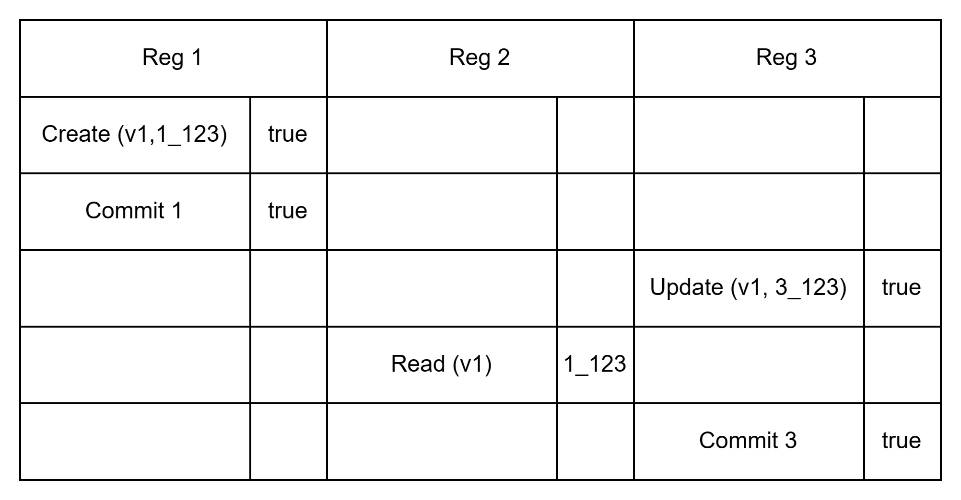


图5.10 第一部分-迟到的读

实验十一为迟到写者，事务2为写事务，事务3为读事务。事务3先读了v1的值，因此之后事务2对v1的增删改操作都无法成功，直到事务3回滚后，事务2才能成功更新v1的值。

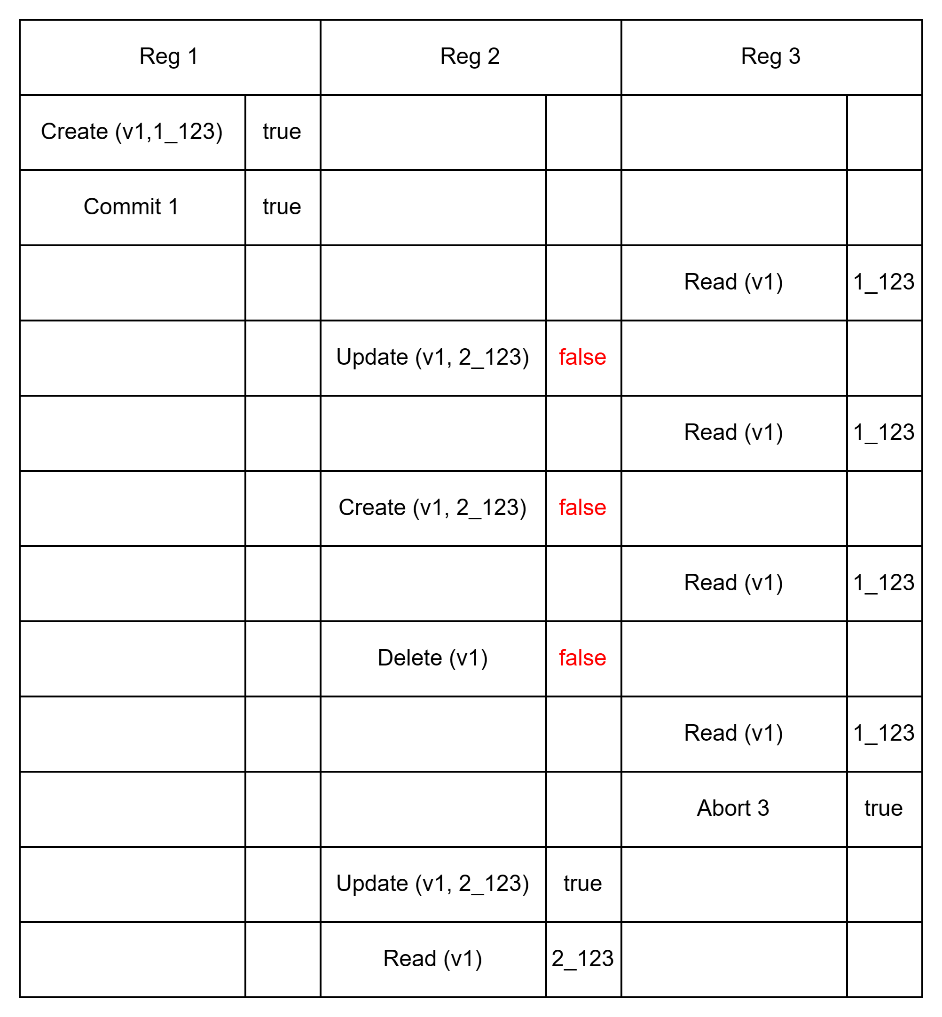


图5.11 第一部分-迟到的写

### 5.2.6 顶点锁

在新建或删除一条边时，不仅会在Index Map中写入对应的键值版本属性关系，还会在保存点边多版本关系的VE Map中写入这条边的修改记录。本文实现了边写入的顶点锁，修改边时会锁定这条边的两个端点对应的版本链，防止对同一顶点的关联边的写-写并发。

实验十二使用Create和Delete操作验证顶点锁。事务1插入了一条从v0到v1的有向边，锁定顶点v0的出边集和顶点v1的入边集，因此事务2插入v0到v2的边以及v3到v1的边会失败，但插入v2到v0的边成功。事务1提交后释放了顶点锁，后两次插入也能成功，此时事务2持有了v0、v2、v3的出边顶点锁和v0、v1、v2的入边顶点锁，因此事务3删除事务1建立的v0到v1的边会失败，直到事务2提交释放锁后才能成功删除。

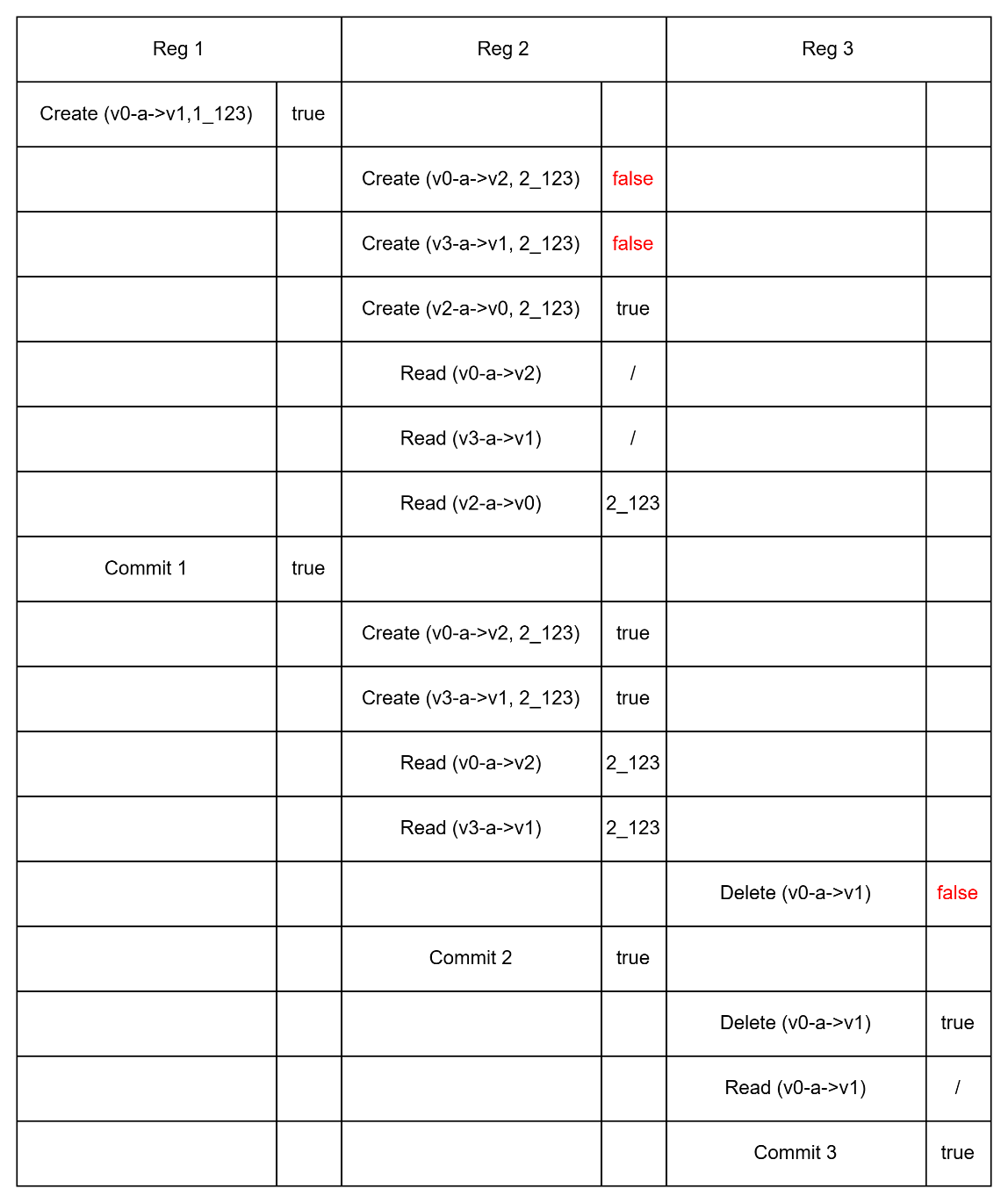


图5.12 第一部分-边写入顶点锁

## 第二部分

第二部分是针对持久化系统以及垃圾回收机制进行的实验，包括对属性多版本存储索引Index Map对应的持久化文件Index AOF的写入与读取，对点边关系多版本存储索引VE Map对应的持久化文件VE AOF的写入与读取，对AOF文件进行重写的实验验证以及手动调用GC的实验验证。

这一部分涉及的API为Rebuilt、Rewrite、GC以及构造数据使用的CRUD API。

### 5.3.1 Index AOF写入与读取

实验一构造了包含事务注册、提交、回滚以及CRUD操作，最后生成AOF文件。实验二基于生成的AOF文件进行Index AOF重建操作。下图序列生成的AOF文件为：

A(v1,1) A(v2,1) A(v1,2) M(v2,1,1) M(v1,1,2) A(v2,3) D(v2) M(v1,2,3) D(v1) A(v3,6)

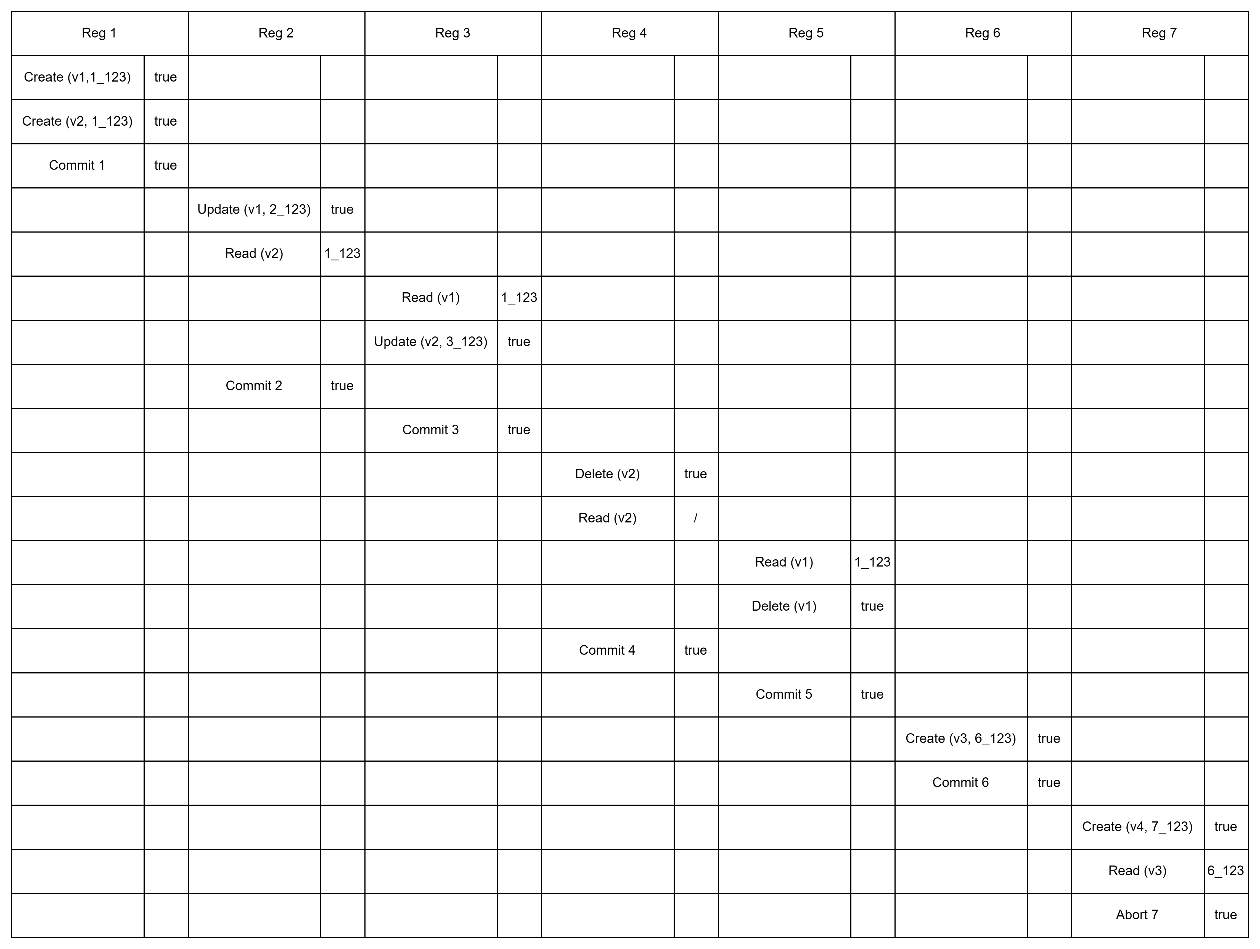


图5.13 第二部分-IndexAOF写入

实验二将上面AOF文件使用rebuilt复原，然后调用PrintIndex打印Index Map，再注册事务10，依次读取v1~v4的值。可以发现上一批操作过程中将v1和v2删除了，v4的写入事务回滚，只有事务6写入的v3提交成功，因此只有v3可以读出值。结果如下：

Rebuilt，Reg 10，PrintIndex => {v3:[{write-lock:0,read-ts:6,begin-ts:6}],v2:[],v1:[]}，Read(v4) => / ，Read(v1) => / ，Read(v2) => / ，Read(v3) => 6\_123。

实验三与第一部分实验七验证单事务回滚-Read-旧事务回滚的操作序列一致，目的是验证由读操作的事务回滚后其对read-ts影响在AOF文件中的实验验证。实验四读取了实验三产生的AOF文件并恢复Index Map。实验五与第一部分实验八验证单事务回滚-Read-新事务回滚的操作序列一致，实验六读取实验五产生的AOF文件并恢复Index Map。

实验三产生的AOF文件为：A(v2,1) A(v1,1) M(v2,1,2) M(v1,1,1) M(v1,1,1)。其中前两项由初始事务1产生，第三项和第四项是事务3读取v1和v2后提交时产生，第五项为事务2回滚时产生的补偿操作，保证依据AOF复原后的v1的read-ts为3。

实验五产生的AOF文件为：A(v2,1) A(v1,1) M(v2,1,1) M(v1,1,1)。其中前两项由初始事务1产生，第三项为事务3回滚时对v2产生的补偿操作，第四项为事务2读取v1后提交时产生。由于事务2读取v2时v2的read-ts已经为3，所以事务2不会产生对v2的修改记录。

### 5.3.2 VE AOF写入与读取

实验七使用Create和Delete构造了VE AOF的写入数据，实验八使用这个数据进行内存索引恢复。下图序列产生的VE AOF为：

OutEdges: {v1: [{begin-ts:1, insert-edges:{v1-a->v2,v1-a->v4}, delete-edges:{v1-a->v4}}, {begin-ts:2, insert-edges:{}, delete-edges:{v1-a->v2}}], v3:[{begin-ts:1, insert-edges:{v3-a->v4}, delete-edges:{}}, {begin-ts:2, insert-edges:{v3-a->v2}, delete-edges:{}}]}

InEdges: {v2: [{begin-ts:1, insert-edges:{v1-a->v2}, delete-edges:{}}, {begin-ts:2, insert-edges:{v3-a->v2},delete-edges:{v1-a->v2}}], v4:[{begin-ts:1, insert-edges:{v1-a->v4,v3-a->v4}, delete-edges:{v1-a->v4}}]}

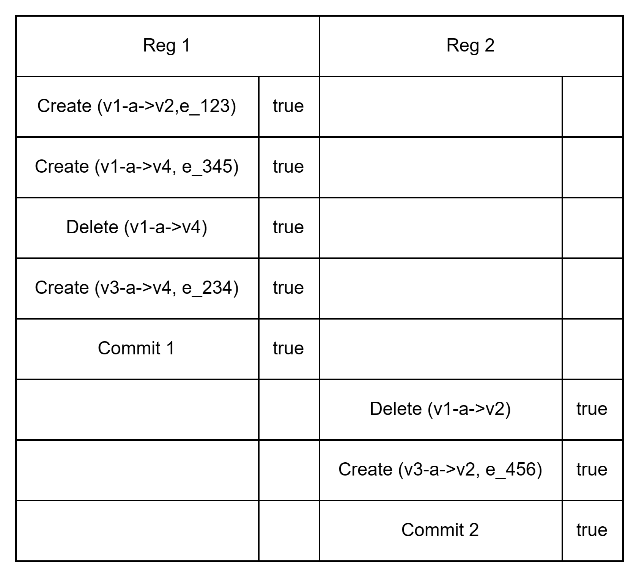


图5.14 第二部分-VEAOF写入

实验八将上面的VEAOF复原后，开启事务13并读取相关边，结果如下：

Rebuilt，Reg 13，PrintVE =>②，Read(v1-a->v2) => / ，Read(v3-a->v2) => e\_456，Read(v1-a->v4) => /。

②为OutEdges: {v1: [{begin-ts:1, insert-edges:{v1-a->v2}, delete-edges:{ }}, {begin-ts:2, insert-edges:{}, delete-edges:{v1-a->v2}}], v3:[{begin-ts:1, insert-edges:{v3-a->v4}, delete-edges:{}}, {begin-ts:2, insert-edges:{v3-a->v2}, delete-edges:{}}]}

InEdges: {v2: [{begin-ts:1, insert-edges:{v1-a->v2}, delete-edges:{}}, {begin-ts:2, insert-edges:{v3-a->v2},delete-edges:{v1-a->v2}}], v4:[{begin-ts:1, insert-edges:{v3-a->v4}, delete-edges:{}}]}

可以验证恢复操作成功恢复了上一个实验中事务2创建和删除边的行为，同时合并了同一个事务产生的版本内对同一条边的插入和删除操作（事务1产生的v1-a->v4的插入和删除行为被合并了）。

### 5.3.3 AOF重写

AOF重写会合并Delete操作之前的操作，以及将Read操作带来多次的read-ts变化合并。实验九验证Delete操作的重写，实验十验证Create和Update操作的重写，实验十一验证多次Read操作的重写。

实验九使用了删除Delete操作，下图产生的AOF序列为：A(v1,1) D(v1) A(v1,3)。

重写后遇见删除操作D时会自动合并之前的A，故重写后的AOF文件为：A(v1,3)。

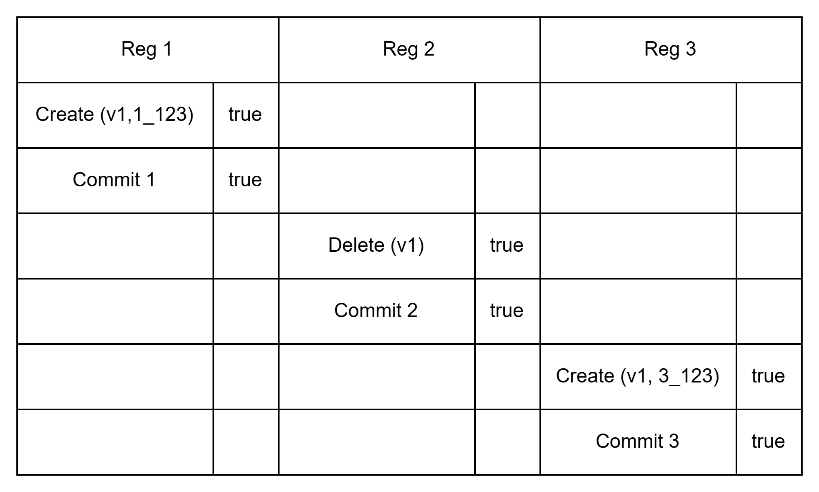


图5.16 第二部分-AOF重写-Delete

实验十使用了覆盖写Create和Update操作，下图产生的AOF序列为：A(v1,1) A(v2,1) D(v1) A(v1,2) A(v2,2)。

重写后覆盖写操作产生的D会覆盖之前同一个键的所有操作，得到的新AOF文件为： A(v2,1) A(v1,2) A(v2,2)。

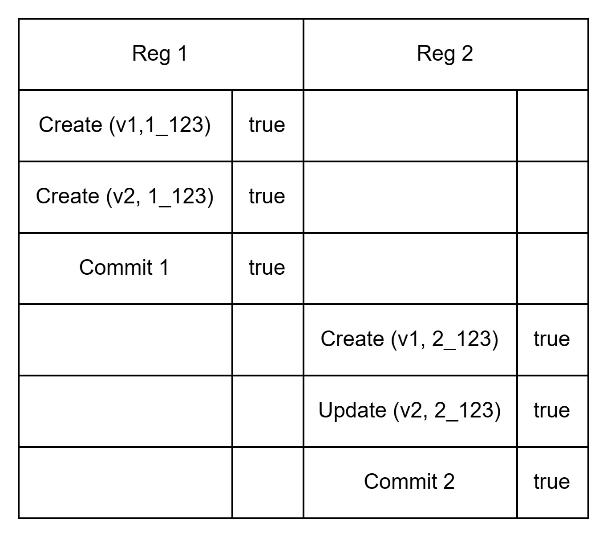


图5.16 第二部分-AOF重写-Create和Update

实验十一使用了覆盖写Create和多次Read操作，下图产生的AOF序列为：

A(v1,1) A(v2,1) D(v1) A(v1,2) M(v2,1,1) M(v1,2,1) M(v2,1,1)。

重写后覆盖写操作产生的D会覆盖之前同一个键的所有操作，同时针对v2的版本1的read-ts的两次改动会被合并，得到的新AOF文件为：A(v2,1) A(v1,2) M(v2,1,2) M(v1,2,1)。

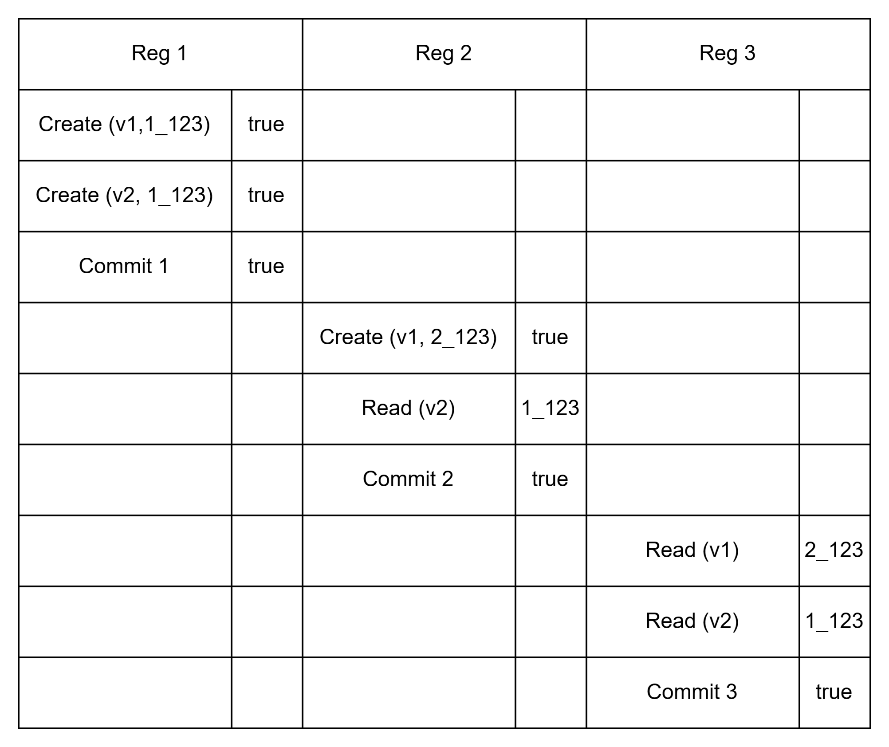


图5.17 第二部分-AOF重写-Read

### 5.3.4 GC

GC会清除内存索引中的软删除版本以及移除不可能再被访问到的过时版本。软删除的版本是由事务回滚导致的，如果当前活跃事务集为空，则GC操作会合并版本链到只剩余一个公开版本为止。

实验十二针对Index Map调用GC前后的变化做了验证。可以看出GC移除了软删除的版本2和过期了的版本1.

①为{v1:[{write-lock:0, read-ts:1, begin-ts:1}, {write-lock:0, read-ts:2, begin-ts: -2}, {write-lock:0, read-ts: 3, begin-ts:3}]}

②为{v1:[{write-lock:0, read-ts:3, begin-ts:3}]}

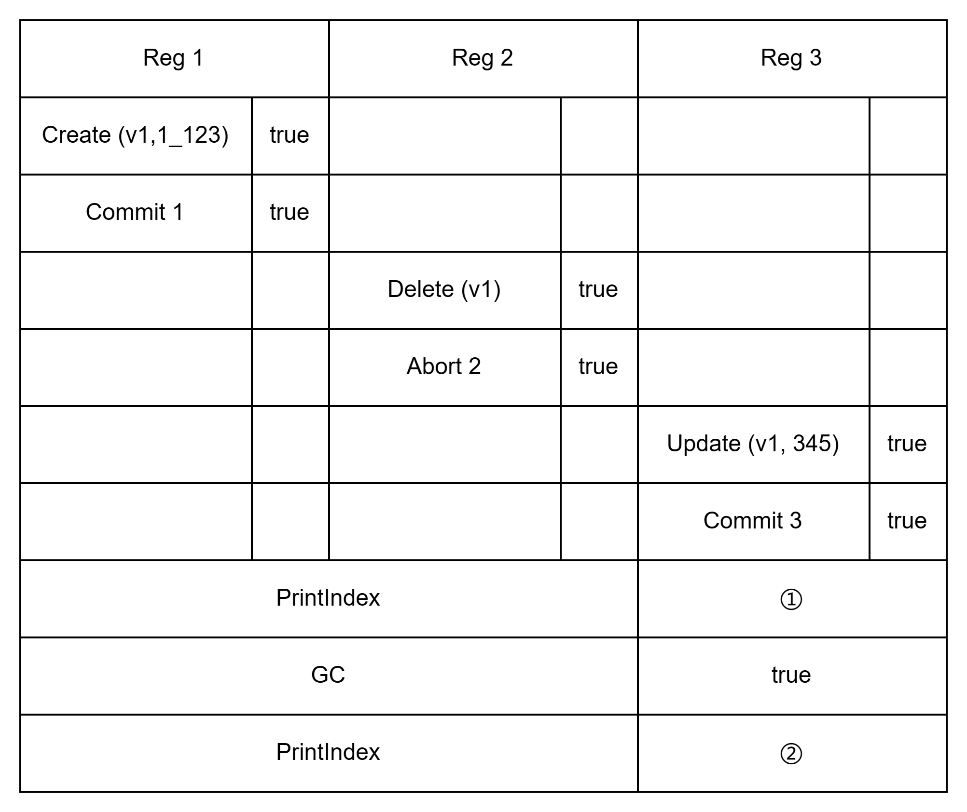


图5.18 第二部分-GC-IndexAOF

实验十三针对VE Map调用GC前后的变化做了验证。GC操作会遍历差分版本链并合并当前边集情况直到活跃事务集能访问到的最小公开版本。可以看出GC操作合并了不活跃事务集的重复删除和插入操作。

①为OutEdges: {v0: [{begin-ts:1, insert-edges:{v0-a->v1}, delete-edges:{}}, {begin-ts:2, insert-edges:{}, delete-edges:{v0-a->v1}}, {begin-ts:3, insert-edges:{v0-a->v1}, delete-edges:{}}], v1: [{begin-ts:1, insert-edges:{v1-a->v2}, delete-edges:{}}], v2: [{begin-ts: 2, insert-edges:{v2-a->v3},delete-edges:{}},{begin-ts:3, insert-edges:{}, delete-edges:{v2-a->v3}}]}。

②为OutEdges: {v0: [{begin-ts:3, insert-edges:{v0-a->v1}, delete-edges:{}}], v1: [{begin-ts:1, insert-edges:{v1-a->v2}, delete-edges:{}}] }

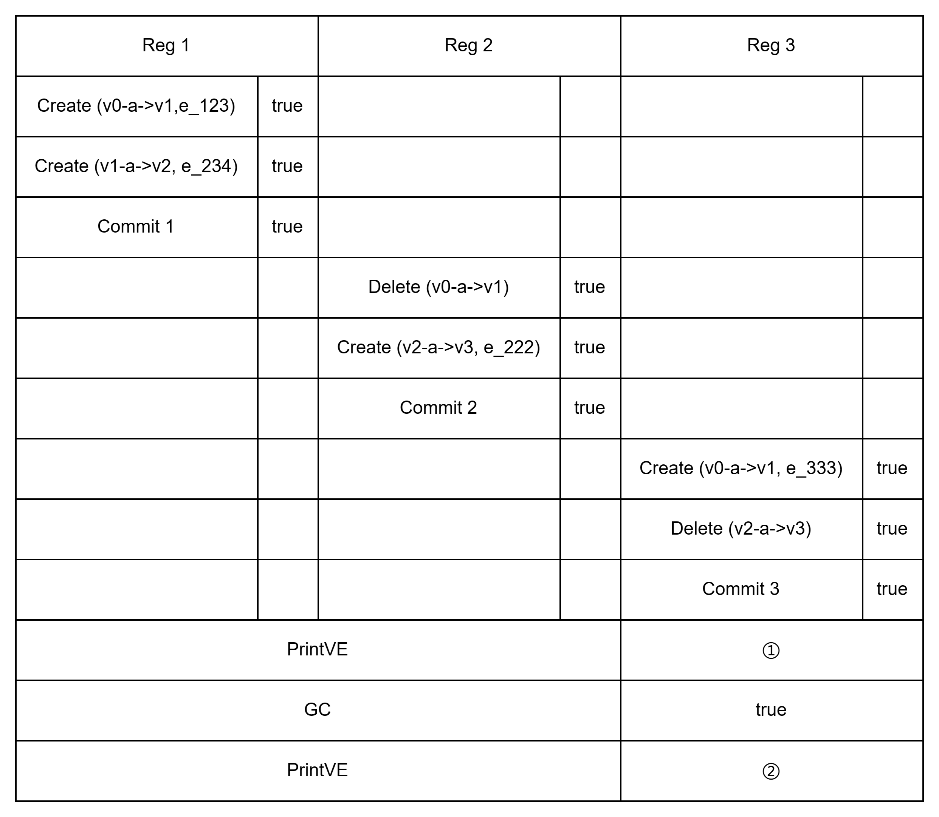


图5.19 第二部分-GC-VEAOF

## 第三部分

第三部分为本文设计系统的更具体场景实验，包括了读偏斜、写偏斜以及丢失更新三种并发异常现象的实验验证，以及针对nGQL特有图查询和图算法语句实现的获取相邻顶点的接口的实验验证。

这一部分涉及的API为GetNeighbors以及基础CRUD API。

### 5.4.1 并发异常现象

并发异常现象包括脏写、脏读、不可重复读、幻读以及读偏斜、写偏斜和丢失更新，其中前四种已经在第一部分的不同隔离级别下的CRUD进行了实验，所以此处会在不同隔离级别下进行后三项的实验。

实验一为读偏斜现象的验证，场景为简单转账场景，从账户x转账30到账户y，此处省去了计算验证过程，业务一致性要求x与y的总和始终为100。事务2负责验证这一约束，事务3进行转账操作。在RC隔离级别下，由于事务2读取的x值较老，而读取到的y的值为事务3已经提交的新版本，因此会得出x=60、y=70这一不满足业务一致性要求的结果。但在RR隔离级别下，事务2只能读取创建时间不晚于自身的版本，只会读到老的y值，此时x与y的值满足事务一致性，不会发生读偏斜现象。

下图左为RC隔离级别下的结果，右图为RR隔离级别下的结果

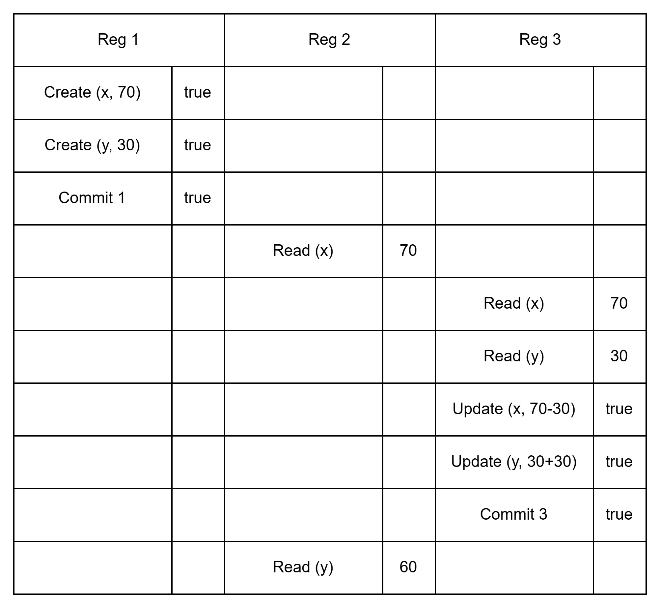
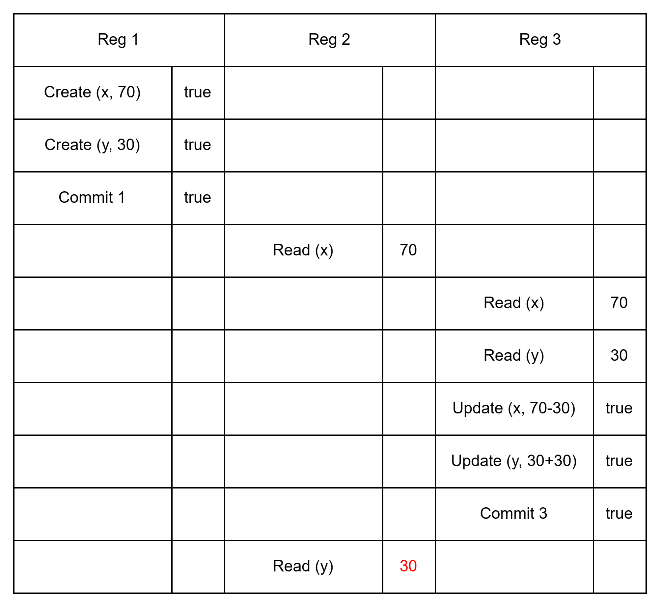
 

图5.20 第三部分-读偏斜-RC/RR

实验二为写偏斜现象的验证，场景为交换两个变量x和y的值，其中一个事务读取x并写入y，另一个事务读取y并写入x，如果x最终不等于y，则发生了写偏斜。由于本文使用MVTO实现禁止了迟到的写者写入，因此相对较晚的事务将无法完成写入操作，只能有一个事务成功写入，即使在RC隔离级别下也不会发生写偏斜。

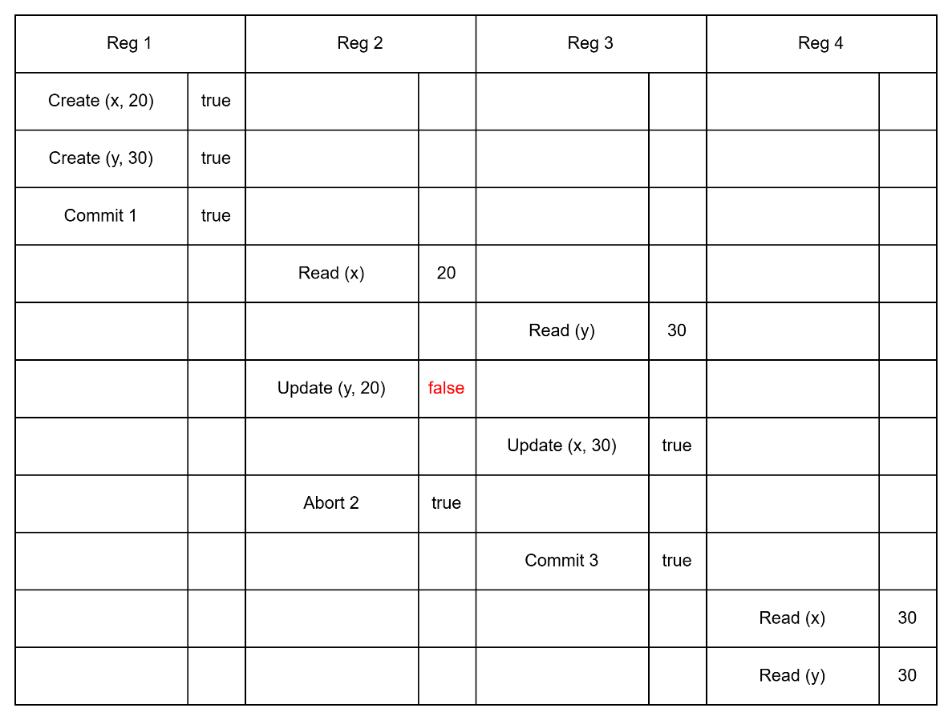


图5.21 第三部分-写偏斜-RC

实验三为丢失更新现象的验证，场景为两个事务同时读取同一个值并分别更新，如果允许它们提交则后提交的会覆盖先提交的，造成丢失更新。由于本文禁止了迟到写者写入，因此在下面两图的两种更新顺序下，事务2都无法写入成功，只有事务3成功写入。

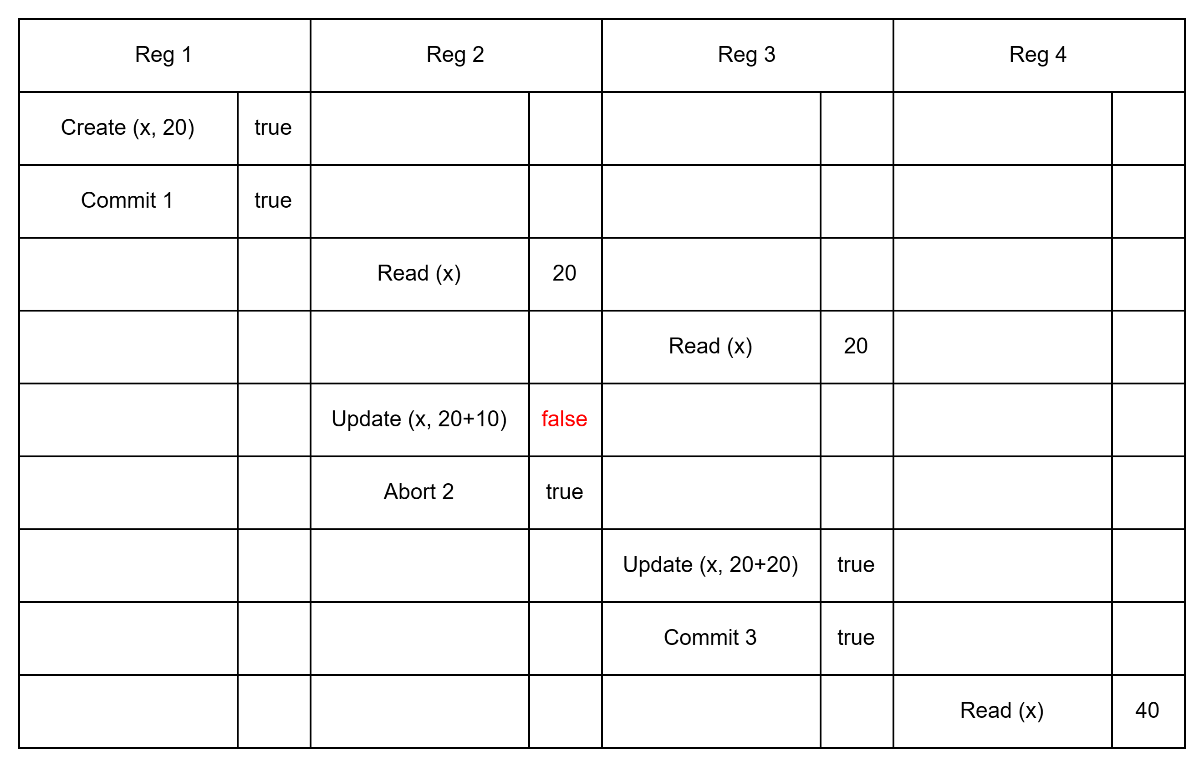


图5.22 第三部分-丢失更新-旧事务先更新

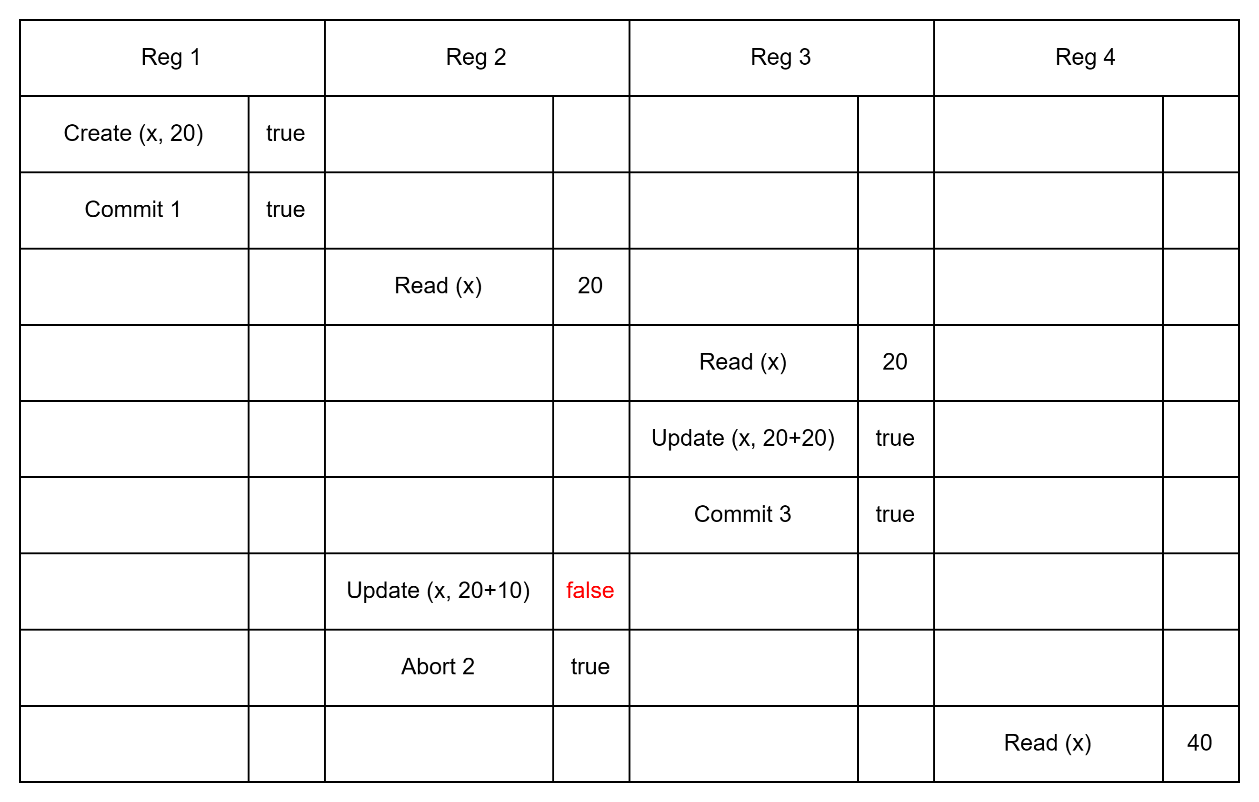


图5.23 第三部分-丢失更新-新事务先更新

### 5.4.2 nGQL

nGQL特有的图查询操作有四种，分别是图遍历操作符GO、获取属性操作符FETCH以及基于索引的查找操作符LOOKUP和基于索引的图模式匹配操作符MATCH。其中基于索引的后两种操作是针对纯属性条件出发的查询场景，会按照属性的值依次对顶点或边进行排序，在Nebula Graph的官方文档说明中不建议在生产环境下使用索引，因为这需要依赖底层RocksDB的前缀扫描功能，比较慢，因此本文未对基于索引的操作进行修改。FETCH操作接收一个顶点或边的类型以及标识，返回指定的属性，因此在存储层调用本文实现的Read即可。

GO操作给定顶点出发，按照给定的步数和遍历方式，经过指定类型的边遍历到对端顶点，返回顶点的相关信息，对应的存储层关键操作为GetNeighbors。GetNeighbors同时还是nGQL内置的各种图算法的实现基础，例如求最短路、求所有路径、求子图等算法。本部分对GetNeighbors进行了实现，并进行了简单的实验验证。

以最基础的一步遍历为说明，GetNeighbors接收一个顶点和遍历的边类型，返回这个顶点的所有满足条件的相邻顶点，并且支持对边的属性进行筛选。这个接口在获取顶点后拼接出key，访问内存中的出边顶点集和入边顶点集两个集合，获得顶点的出入边集合，再解码边的key得到顶点的邻居。在本文实现中，GetNeighbors可以在只通过内存索引而不对磁盘KVStore进行前缀扫描的情况下得到结果，如果业务场景不需要基于边的属性进行筛选，无疑可以节省时间和资源。

实验四创建了一批出入顶点以及类型不同的边，并调用GetNeighbors接口获取结果。GetNeighbors (vertex, edgeType, pos) 表示按照边类型查找指定顶点的出边（pos=1）、入边（pos=-1）、出边和入边（pos=0）。

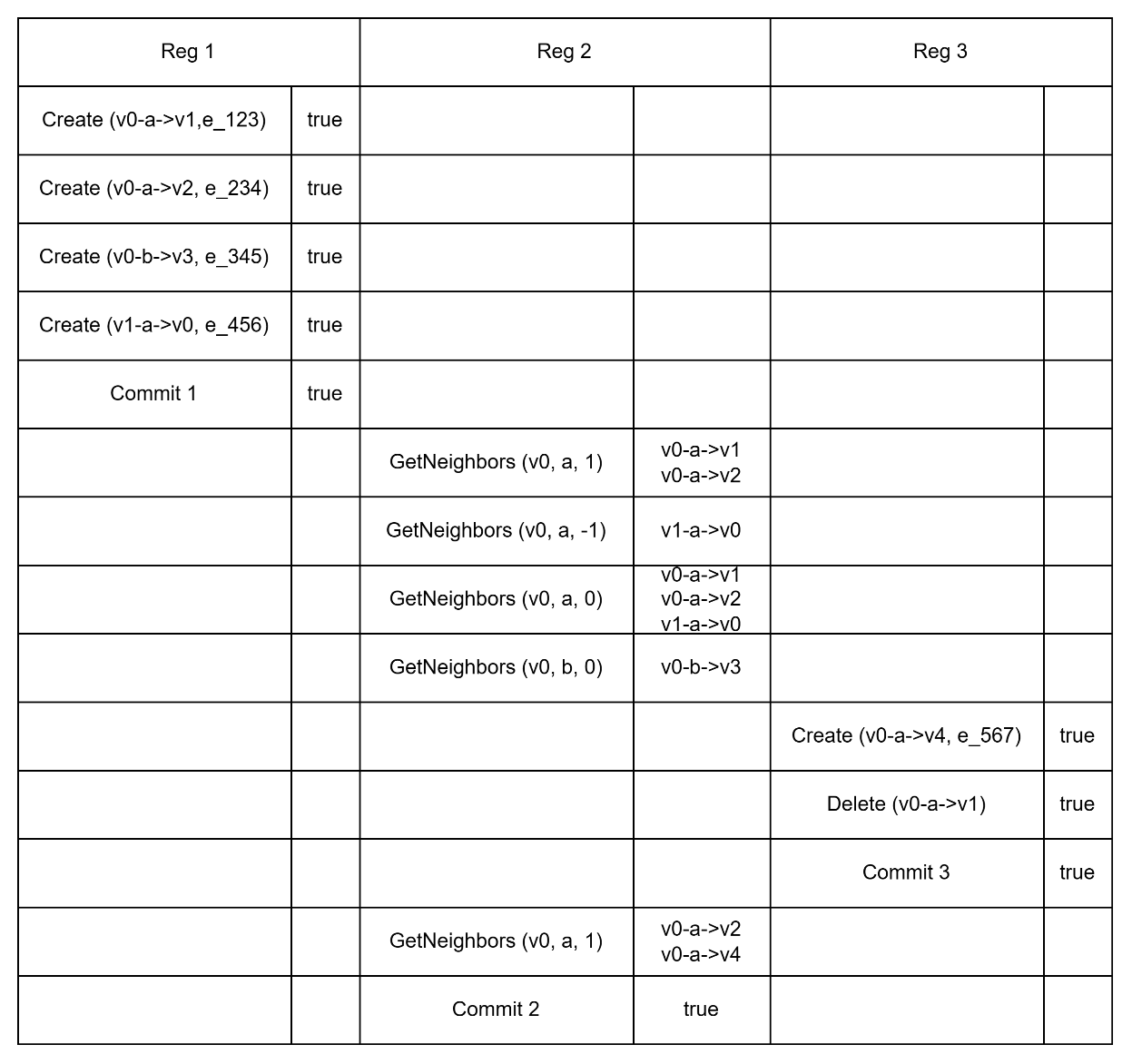


图5.24 第三部分-查找邻居

# 结论

本章将对本文的研究工作进行总结，并说明本文研究的一些不足点和改进点，提出未来的研究和改进方向。

## 总结

图数据在如今起到了越来越重要的作用，而图数据库系统特别是属性图数据库系统也自然成为数据库领域的研究热点之一。为了使图数据库系统能够适用于更多场景，为分布式图数据库系统设计并实现严格事务系统也是一项有实际意义的需求。现有的商业化图数据库中支持原生图、分布式可扩展的属性图数据库并不多，这其中的优秀产品Nebula Graph并不支持事务，需要依靠用户手动重试，对比关系型数据库领域，是一个不可忽视的缺失。

本文主要工作及贡献主要体现在以下几个方面：

1、提出了一种针对属性图数据库的基于多版本并发控制协议的数据库事务设计，并在此基础上实现了读已提交和可重复读隔离级别以及与多版本兼容的简单的可串行化隔离级别。

2、对多版本并发控制协议的选型以及并发冲突现象进行了分析，改进了原始论文中的实现，降低理论要求的严格性以提高实践中的性能。

3、设计了一个包含事务管理系统、属性多版本内存索引、点边关系多版本内存索引以及持久化及恢复系统等四个部分的单机事务系统，并编写样例进行正确性实验验证。使用这一套系统不会增加访问磁盘次数，并可以降低图查询和图算法中常用的获取顶点邻居操作访问底层磁盘存储的次数，因而理论上可以在这类图数据结构特有场景下降低查询延时和系统IO总量。

## 不足与展望

本文选择的数据库是一个分布式属性图数据库，因此其上的完整事务实现需要考虑分布式事务，而本文只基于存储服务实现了单机事务以及分布式架构下的分布式ID和时钟方案，距离实现满足分布式原子性、分布式一致性等特性的分布式事务的目标还有不少需要完善的地方。

同时，本文实现中存在一些可以优化并缩小存储占用、提高执行效率之处，例如持久化系统对事务操作的编码和存储，目前本文只是将操作字段转为字符串序列再进行文件写入和读出操作，但如果考虑利用编码冗余性，进行按位编码，并使用序号代替操作中部分具有枚举性质地字段，可以压缩持久化记录文件的大小。由于切边存储方式会导致每条边被存储两次，因此如果在持久化系统中对边进行编号后存储，可以减少存两次边带来的冗余。本文代码实现大多使用CPP STL库的标准容器，但现有的各种CPP开源库，例如功能更完善的Boost、性能更高的Folly，以及特定情形下的无锁数据结构的开源实现，是比STL效率更高的选择。

MVCC协议只是一种设计的约定，在细节方面不同的产品有不同实现方式，例如当事务冲突发生时，可以选择回滚当前事务，等待并重试，或在特定条件下抢占锁进行执行。本文以及提出MVTO概念的论文中的标准实现，对于并发冲突的解决方案都是直接回滚，因为这种实现不会出现死锁现象，但这可能导致更高回滚率，特别是出现长事务时，一旦回滚，之前已成功执行的部分也将回退。一些开源或商业数据库采用了不同的实现方式，例如CockroachDB在读写冲突场景下发现事务的时间戳太小以至于不能继续执行写操作时，会推迟自身时间戳直到恰好大于这个临界时间戳再重试事务，而在写写冲突场景下会自旋等待后尝试重新执行，这样的实现方案可以减少使用事务对用户业务的入侵，并降低回滚概率，这是该数据库只提供可串行化隔离级别也能达到足够吞吐量的一个重要原因[33]。

如果本文的设计要进一步考虑提高事务执行成功率，则需要修改出现冲突时的策略，将回滚换为等待重试，因此还需要在系统中引入死锁检测或预防系统，并可以采用类似网络传输协议CSMA/CA使用的指数回退算法进行重试以降低冲突概率[34]。为了实现分布式事务，除了本文实现的单机事务系统以及分布式ID和分布式HLC时钟方案，还需要实现共识层协议用来同步各节点的事务状态以保证分布式原子性，以及一个能给出全局有序关系的时间判断装置——HLC时钟方案只能为有因果关系的事件进行排序，对于发生在不同节点上且没有相互交流的并发事务无法判断先后顺序——这可以参考Weaver的实现[35]。它是一个基于HLC实现全局有序关系的分布式数据库，对于HLC无法排序的事务按照先到先提交的方式给出全序，并且只有当两个事务发生了冲突需要排序时才进行排序，对于无冲突关系的两个HLC并发事务不做处理。

# 参考文献

[1] Negro, A. (2021). *Graph-Powered Machine Learning.* Manning Publications Co.

[2] Conn, S. S. (2005). OLTP and OLAP data integration: a review of feasible implementation methods and architectures for real time data analysis. *Proceedings. IEEE SoutheastCon,* 2005, 515-520.

[3] Robinson, I. & Webber, J. & Eifrem E. (2015). *Graph Databases.* O’Reilly Media, Inc.

[4] McKenney, P. (2013). RCU Usage In the Linux Kernel: One Decade Later. *Technical Report*, 2013.

[5] Wu, Y. (2017). An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *VLDB*, 10(7), 781-792.

[6] Gupta, A. (2017). NoSQL databases: Critical Analysis and Comparison. *IC3TSN*, 2017, 293-299.

[7] Mukherjee, S. (2019). The Battle between NoSQL Databases and RDBMS. *SSRN,* 2019.

[8] Li, Z. (2018). NoSQL Databases. *The Geographic Information Science & Technology Body of Knowledge*, 2018(2).

[9] Gremlin Document. [2023-05-06]. <https://tinkerpop.apache.org/docs/current/tutorials/getting-started>

[10] GraphQL. [2023-05-06]. <https://graphql.org/>

[11] W3C RDF. [2023-05-06]. <https://www.w3.org/RDF/>

[12] ANSI SQL-92.[2023-4-23]. <https://datacadamia.com/_media/data/type/relation/sql/sql1992.txt>

[13] Gilbert, S. (2012). Perspectives on the CAP Theorem. *Computer*, 45(2), 30-36.

[14] Nebula Graph. [2023-05-06]. <https://www.nebula-graph.com.cn/>

[15] Dong, S. (2017). Optimizing Space Amplification in RocksDB. *CIDR*, 2017, 82-90.

[16] Redis. [2023-05-06]. [https://redis.com/](https://redis.com/whitepapers/)

[17] MongoDB. [2023-05-06]. <https://www.mongodb.com/>

[18] Rakhmatulin, A. (2015). BigTable vs. HBase. *Internet Scale Distributed Systems,* 2015.

[19] Neo4j. [2023-05-06]. <https://neo4j.com/>

[20] Huge Graph. [2023-05-06]. <https://hugegraph.apache.org/docs/>

[21] JanusGraph. [2023-05-06]. <https://janusgraph.org/>

[22] Özsu, M. T. & Valduriez, P. (2011). *Principles of Distributed Database Systems: Third Edition.* Springer.

[23] Gray, J. N. (1976). *Granularity of Locks and Degrees of Consistency in a Shared Data Base.* North Holland Publishing Company.

[24] Berenson, H. (1995). A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD*, 95, 1-10.

[25] Kung, H. T. (1981). On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), 213-226.

[26] Ports, D. R. K. (2012). Serializable Snapshot Isolation in PostgreSQL. *VLDB*, 5(12), 1850-1861.

[27] Helland P. (2007). Life Beyond Distributed Transactions: an Apostate’s Opinion. *CIDR*, 2007, 132-141.

[28] Apache Zookeeper. [2023-05-06]. <https://zookeeper.apache.org/>

[29] Corbett, J. C. (2013). Spanner: Google’s Globally Distributed Database. *ACM Trans. Comput. Syst.* 31(3), Article 8, 22.

[30] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communication of the ACM*, 1978(21), 558-565.

[31] Kulkarni, S. S. (2014). Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases. *OPODIS*, 2014, 17-32.

[32] Nalawala, H. S. (2022) A Comprehensive Study of “etcd” - An Open-Source Distributed Key-Value Store with Relevant Distributed Databases. *Emerging Technologies for Computing, Communication and Smart Cities*, 875.

[33] Taft, R. (2020). CockroachDB: The Resilient Geo-Distributed SQL Database. *SIGMOD*, 20, 1493-1509.

[34] Kwak, B. J. (2005) Performance Analysis of Exponential Backoff. *IEEE/ACM Transactions on Networking*, 13(2), 343-355.

[35] Dubey, A. (2016). Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *VLDB*, 9(11), 852-863.

# 附录 A 部分系统设计代码

事务系统

# 致谢

自2018年九月进入北京大学，历经五年，我也即将完成本科阶段的学习，在本科毕业论文完稿之际，我的心中充满了喜悦以及感激之情。在此，我要向本科阶段以及本篇论文写作过程中帮助过我的人们表达我的感谢，并以这篇致谢为我本科阶段最后一项正式工作划上圆满的句号。

在这五年间，我的成长离不开老师们、同学们、家人们以及朋友们的无私帮助，因此，向所有帮助过我的人表达我衷心的感谢，谢谢你们陪伴我成长。

首先感谢我的毕业论文导师陈向群教授。我在高中时期来到北大参加信科夏令营时曾听过陈老师为高中生讲的有关操作系统的课程，后来在大二的计算机系统导论课程上与陈老师相识，并在其后修读了陈老师开设的操作系统课程，其间进行了多次学业上的交流。后来，我在陈老师的实验室中旁听过组会，也在后续的计算机系统导论课程中担任助教，面临本科毕业论文选题时，是陈老师高屋建瓴的指点和循循善诱的引导让我明白了自己想做什么，并初步确定了大四以及研究生阶段计划深入研究的领域，陈老师对系统方面科研领域的洞察力以及对学生的培养和引导令我钦佩不已。另外，陈老师在生活中也对我关爱有加，在我大二因为打球摔伤脚踝时，陈老师不仅关心我的情况，告诉我休养和康复知识，还带了膏药给我，这份关心我至今记得。当我在学习生活中遇见了困惑或处于一段艰难时期时，与陈老师的交流也能带给我走出困境解决困惑的启发以及继续前进的动力。可以说与陈老师的结识在很大程度上影响了我在本科期间对学习与生活、科研与工作的态度，并帮助我找到了自己喜欢的计算机方向。

感谢我的男朋友陈子威在我的毕业论文系统设计、代码编写以及论文写作过程中对我的支持和帮助，作为我在实习期间认识的同事，有着更丰富的业界经验以及科研经验。他在我选择题目方向时提出了不少建议并鼓励我在喜欢的领域上深入探索，在我进行系统设计时提醒我要提纲挈领，并帮我找到了许多优秀的开源组件实现，不仅使我意识到做系统设计应该先将其实现出来，而非在设计阶段就纠结于细节上的优化，帮助我走出了系统设计时思维卡壳的阶段，还帮助我走出“闭门造车”的思维局限区。同时，他作为硕士毕业生，也在我进行论文写作的过程中提供了学术规范以及格式指导等方面的帮助，使我能更好地完成本科毕业论文。在日常生活中，他还是我的“小黄鸭”，当我在写代码以及看论文遇到问题时，我都会与他进行讨论交流，在叙述自己的思路和实现的过程中逐渐厘清正确的方法。在我学习工作压力大而情绪不佳，特别是2022年不能出校的那段时间里，也是他在倾听我、鼓励我和帮助我。可以说，从认识后一直到我完成本科毕业论文这段时间里，我能保持良好稳定的心态和持续的产出，他有很大的功劳。

感谢我的排球队队友，信科排球队是一支充满了快乐的球队，我们在球场上一起拼搏，在球场下也能约饭约自习以及进行学术讨论交流。完全人格，首在体育，在排球场上运动锻炼使我有了健康的身体来支持我的学习生活，汗如雨下之外更是理解了坚持和热情的重要性，锻炼了我坚毅的品格。排球队的学长以及学姐们也是我交流取经的主要对象，他们在我的本科学习以及本科毕业论文写作过程中也对我起到了帮助，特别是信科2017级的周厚健学长，不论是课程资料还是论文相关资料，他都不吝帮助。

感谢我在大四下在学校心理咨询时遇见的心理咨询师姐姐，持续两个月的心理咨询帮助我从大四下的低谷期逐渐走出漩涡，让我在工作以及写论文最忙碌的时候能够有地方毫无顾忌地表达我的想法。她是一位良好的倾听者，能够设身处地的和我交流，同时给出了许多看待问题的新角度以及突破困境的建议感谢她。在这个大家都很忙碌而疏于交流的时代，感谢她能够倾听我和帮助我。

感谢我的父母对我一路成长的支持，使我不用为读书期间的生活费用发愁，也不必因为赚钱养家的负担而提前放弃自己的追求。一路走来，父母给予了我在大多数情况下的选择自由，从我在入学第一年时决定转专业到信息科学技术学院，再到我在实习、保研、出国等节点上的选择，父母都没有对我有太多干涉。当我感觉学习生活困难时，父母是我的坚强后盾，给予我放手一搏和选择的底气，他们的理解以及虽远在千里之外但时常的关心也是安抚我情绪的良药。

感谢歌手周深，周深的歌声永远能带给我坚持生活的勇气和力量。虽然由于期末考试以及上课的原因，我在北京的五年时间内仅仅只在今年《铃芽之旅》首映礼上见过周深，但他的歌却伴随了我整个本科生涯。在我写计算机系统导论、计算机网络、编译原理等课程的Lab感到绝望和崩溃时，是他的一首首振奋人心的歌曲，让我热泪盈眶，让我坚持下去直到走出绝望。每当我觉得疲惫时，周深的拼搏故事以及努力工作的敬业态度也是激励我坚持和拼搏的一大来源。他对待世界的温柔却坚定的态度，在漫天质疑的话语中证明自己的成长的态度，都是我在生活中努力学习的，带给我坚持生活甚至热爱生活的力量。

感谢我的朋友，数院2018级毕业生吴清玉同学，我们相识于入学军训，两个非常能唠的人一见如故，此后在排球场上一起打球，在学习间隙约饭。我们都是混迹于男生堆里的女生，虽然不是同一个学院，但相似的学习生活背景以及生活习惯带来了相近的感受，我们之间的交流的广度和深度都超越我和他人的交流，可以从学习聊到生活，从运动聊到情感，从自我选择聊到社会现状。感谢吴清玉同学将笔记和习题借给我，辅导我学习数分，帮助我度过本科阶段我认为最难的课之一的数学分析，我们一同修读一些课程时的互帮互助也让我感受到了朋友的温暖。同为理科院系女生，吴清玉同学是我在学习、学工、运动和科研等方面上的榜样。

感谢所有在群聊里听过我吐槽以及聊日常生活的朋友们，虽然我们大多从未见过面，只是“点赞之交”，但多年的网友关系已经让我们足够熟悉，虽然现实生活中我很少与朋友一起行动，但在网络世界里，我们分享了喜怒哀乐。谢谢你们和我一起吐槽或怒骂那些令人烦心的事，在我陷入自我怀疑的低谷时给我安慰和鼓励，我也从你们这里获得了许多好吃好玩的美食景点推荐。特别感谢我曾经的室友赵沁乐和马玉娇，让我这个社恐的人能够在寝室大胆说话，我曾经在寝室说话量几乎占生活中的八成以上，你们毕业后我很少再在学校中线下和人交流。

感谢我在字节跳动的实习经历，这段持续八个月的实习提高了我的工程能力，规范了我的编程习惯，并且让我在短时间内接触到了大量的先进业界实践。虽然这段实习经历并不轻松，但我在其中获得了大量的成长，成为了一个更负责的、更有责任感的能独当一面的人。感谢实习期间的同组以及合作端的许多同事，和你们的交流不仅满足了我在线下与人说话的需求，也教会了我全新的思考问题方式。我的研究方向以及这篇本科毕业论文的选题就来自于对实习经历的体会和思考，如果我不曾体验过业界的真实情况，将很难确认自己的喜好，遑论选择一个的确感兴趣的方向进行研究。

感谢北京大学和信息科学技术学院在我本科就读期间提供的良好环境和多种优秀资源，我获得了扎实的学科基础以及充分的学术训练，养成了自主学习的好习惯。我在本科期间还修读了国家发展研究院开设的经济学双学位，这一段学习经历带给了我学科交叉的背景和看待问题分析问题的新方法，使我发现经济学和计算机科学以及现实世界的共通之处和相似的哲学，让我更热爱生活。在北京大学的学习过程中，我逐渐形成了兼容并包的处世态度，拥有了健全的体魄和人格，而丰富多彩的课余活动既增长了我的见识又让我学会了劳逸结合以及合理规划时间，有幸能在北京大学度过我的本科时光，收获完整的本科生活。

# 北京大学学位论文原创性声明和使用授权说明

**原创性声明**

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：

日期： 年 月 日

**学位论文使用授权说明**

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

* 按照学校要求提交学位论文的印刷本和电子版本；
* 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
* 学校可以采用影印、缩印、数字化或其它复制手段保存论文；

论文作者签名： 导师签名：

日期： 年 月 日