

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

本科专业： 数据科学与大数据技术

指导导师： 陈向群

二○二一 年 六 月

北京大学本科毕业论文导师评阅表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学生姓名 | 宋怡馨 | 学生学号 | 1800011747 | 论文成绩 |  |
| 学院（系） | 信息科学技术学院 | | | 学生所在专业 | 数据科学与大数据技术 |
| 导师姓名 | 陈向群 | 导师单位/  所在研究所 |  | 导师职称 |  |
| 论文题目  （中、英文） | | 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 | | | |
| 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

小四号，宋体/Time new roman，1.5倍行距

**关键词：**3-5个

# Abstract

5号，Time new roman，1.5倍行距

**Key Words：**3-5个

**全文目录**

[摘要 9](#_Toc476314826)

[Abstract 10](#_Toc476314827)

[第一章 绪论（二号、黑体） 12](#_Toc476314828)

[1. 第一级（三号、宋体/Time new roman、加粗） 12](#_Toc476314829)

[1.1 第二级（四号，宋体/Time new roman，加粗） 12](#_Toc476314830)

[1.1.1 第三级（小四，宋体/Time new roman，加粗） 12](#_Toc476314831)

[参考文献 14](#_Toc476314832)

[本科期间的主要工作和成果 15](#_Toc476314833)

[致谢 15](#_Toc476314834)

**图表格式**



图1 （五号，宋体/Time new roman）

表1 （五号，宋体/Time new roman）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表头 |  | |  |  | |
|  |  |  |  |  |
| 内容，（五号，宋体/Time new roman） |  |  |  |  |  |
|  |  |  |  |  |  |

(表格使用如上所示的三线表)

# 第一章 绪论

### 研究背景与意义

### 研究现状

### 研究内容

### 论文结构

# 第二章 相关工作

### 图数据库系统

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，其中图计算领域包括xxxx，典型代表为xxxxx。图存储领域则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。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；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；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，适用于上文提到的OLAP场景，代表有HBase、Hive和BigTable。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。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。

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。

### 多版本并发控制

### 数据库事务

### 分布式ID生成算法

### Nebula Graph系统架构

# 第三章 多版本并发控制实现

### 整体架构

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）和提交（CommitTx）操作的解析，在Storage层增加了Transaction Manager用于管理事务，Index Map用于键的多版本管理，以及AOF Manager用于将操作持久化和断电恢复；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）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，在现实中并不常用，等同于未进行任何事务隔离时的访问，无需进行修改；串行化（Serialization）则要求所有事务的实现都是串行的，是最高的隔离级别，在本文实现中只需修改Transaction Manager，在当前事务未完成前，阻塞新事务的注册即可，不需要多版本存储实现，因此也是比较简单的。

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。

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。

Transaction Manager是事务实现的关键，承担了事务注册、提交、回滚等功能，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AOF Manager用来写入对Index Map的操作到文件以及恢复时读取，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AOF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，但现有数据库中使用已不太多。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，本文选择了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x+1，这样可以避免其他持有Bx版本的写者进行下一步。

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】

### 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)。

### 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，定时且并行地扫描Index Map和VE Map中的每条版本链，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Map加读写锁，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Golang中的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记录，并批量写入AOF Manager，最后释放写锁。

### 操作流程

### 开启事务（BeginTX）

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Index Map中各个键的访问事务集。

### 读操作（READ）

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。

### 增加操作（CREATE）

增加操作为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。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。基于这一分析，本文在此处选择了类似RCU的机制对版本链进行了替换处理。RCU全称为Read-Copy-Update，可以在不影响读的情况下写入，并在所有读者都离开临界区后进行替换。

Create操作在锁定原有版本链最新版本的基础上，会先生成一个仅自己可见的私有版本，存储在对应事务的UndoWriteSets中，UndoWriteSets维护了key与新的版本链之间的关系，该事务对同一key的后续操作也将记录在这个新的版本链上而非Index Map中的版本链上。其次，当事务提交时，会用新版本链替代Index Map中的旧版本链，此时旧版本链中所有版本都已失效，需要删除底层KV store中的对象。

需要注意的是，本文选择的这种实现方式可以支持RC和RR隔离级别的读取，但对于RU隔离级别，其他事务需要能够读取未提交的更新，而私有版本不同于其他修改写入，是单独的一条版本链，所以不能直接读取得到。考虑到RU隔离级别等同于不开启任何事务的效果，此处认为这一实现方式是能支持多版本事务的实现方式，下文将不再讨论对RU隔离级别的支持性。

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。

### 删除操作（DELETE）

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，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，提交后将无法访问这一key。所以在事务的UndoWriteSets中将对应key的版本链置为空，并在事务提交后用新的版本链（空链表）替换原有的版本链，依次删除失效的旧版本链中的版本，实现删除操作。

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。

### 更新操作（UPDATE）

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。

### 提交事务（CommitTX）

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，对当前可见的最新版本以及新写入的版本都设置了当前事务的写锁，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】

### 回滚事务（AbortTX）

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。

【此处是上面举例的图示】

2、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。

3、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。

# 第四章 持久化设计

### AOF系统设计

AOF系统的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，其中包含了一个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）。由于后文叙述的无锁队列实现，即使只使用一个buffer，在写入磁盘的同时也支持继续写AOF entry，高并发场景下并不会引起阻塞。

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，其实际作用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，而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（M）、增加（A）、删除（D），则A和D只需要记录操作的key与version即可，而M还需记录修改的值。也即是说，M包含operation、key、version、delta四个域，而A和D只包含operation、key、version三个域。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的四种编码】

1. Modify：在本文实现中，Modify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，每条AOF记录只指定一个version。
2. Add：Add操作会在Index Map上插入新的版本，因此出现在UPDATE、CREATE这两种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AOF记录指定一个key的所有版本，编码时将version设置为0；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遍历版本链对过期版本的删除，此时每条AOF记录只指定一个version。

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，而仅有一个主管线程负责将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。
2. 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。
3. 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。
4. 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。

### 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。

### 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。

AOF重写采用定时执行的方式，每100秒进行一次重写，在重写过程中当前AOF文件不可访问，期间的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，使用一个临时Index Map读入AOF，下文则介绍写回阶段的主要流程。

AOF读入完成后，内存中已有临时Index Map的精简形态（不包含冗余key），此时遍历Map，对每一个key的版本链上的版本，依次生成Add操作的AOF entry并写入新的AOF文件。写入完成后释放Index Map并将新文件重命名为原始AOF文件名以达到覆盖效果。

# 参考文献

正文，五号，中文宋体，英文Time new roman，1倍行距

1. 期刊 作者. 论文名. 刊名, 出版年份, 卷号(期号): 起始-截止页
2. 专著 作者. 书名. 版本(第一版不写). 出版城市: 出版社, 出版年份: 起始-截止页
3. 论文集 论文作者. 论文题目//编者. 论文集名: 其他题名信息. 出版城市(或者会议城市): 出版者, 出版年: 引文起始-截止页码
4. 学位论文 作者. 学位论文题名. 城市: 论文保存单位, 年份
5. 网络文献 作者. 题名[文献类型标志/文献载体标志]. 出版地: 出版者, 出版年(更新日期)[引用日期]. 获取和访问路径

\*注意: 作者姓前名后, 超过3名作者列前3名, 后加“, 等”; 英文姓名, 姓前名后, 姓首字母大写, 名缩写; 文献的项目要完整, 各项的顺序和标点要和格式要求一致; 未公开发表的论文、报告不列入正式文献, 如有必要可在正文当页下加注。英文文献格式同上。参考文献在正文中按出现顺序用[1], [2]......在右上角标注, 放在“参考文献”中时, 用[1], [2], ...顺序标注。

# 致谢

正文，小四号，中文宋体，英文Time new roman，1.5倍行距