摘 要

数据库管理系统是各种现代信息管理系统最重要，最常用的状态存储方案。传统关系型数据库提供原子性、一致性、持久性和隔离性。由于互联网技术的发展，为了方便大量的半结构化数据的存储，近些年来涌现出了一批NoSql数据库。这些数据库提供了较快的读写速度，但是牺牲了一定传统数据库的特性。随着数据量的增大，分布式数据库应运而生…… 本文旨在探索常规数据库存储方案的实现原理，优化方案。

**关键词：**关系型数据库，SQL，NoSql，分布式数据库

内容目录

[摘 要 I](#__RefHeading___Toc2054_2331702048)

[第一章 绪 论 IV](#__RefHeading___Toc2056_2331702048)

[1.1 研究工作的背景与意义 IV](#__RefHeading___Toc2058_2331702048)

[1.2 数据库技术的国内外研究历史与现状 IV](#__RefHeading___Toc2060_2331702048)

[1.3 本文的主要贡献与创新 V](#__RefHeading___Toc2062_2331702048)

[1.4 本论文的结构安排 V](#__RefHeading___Toc2064_2331702048)

[第二章 数据库储存基本原理 6](#__RefHeading___Toc2066_2331702048)

[2.1 B+树实现原理 6](#__RefHeading___Toc2068_2331702048)

[2.1.1 B+树简介 6](#__RefHeading___Toc2070_2331702048)

[2.1.2 B+树的定义 6](#__RefHeading___Toc2072_2331702048)

[2.1.3 B+树的性能 7](#__RefHeading___Toc2074_2331702048)

[2.1.4 B+树的例子 7](#__RefHeading___Toc2076_2331702048)

[2.1.4 B+树的基本操作 7](#__RefHeading___Toc2078_2331702048)

[2.1.4.1 查询 8](#__RefHeading___Toc2080_2331702048)

[2.1.4.2 插入 8](#__RefHeading___Toc2082_2331702048)

[2.1.4.3删除 10](#__RefHeading___Toc2084_2331702048)

[2.2 Hash存储原理 10](#__RefHeading___Toc2086_2331702048)

[2.2.1 Hash存储原理 10](#__RefHeading___Toc2088_2331702048)

[2.2.2 Hash存储的性能 11](#__RefHeading___Toc2090_2331702048)

[2.2.3 Hash存储的基本操作 11](#__RefHeading___Toc2092_2331702048)

[2.1.4.1 查询查询 11](#__RefHeading___Toc2094_2331702048)

[2.1.4.1 其他 11](#__RefHeading___Toc2096_2331702048)

[2.2.4 Hash存储示意图 11](#__RefHeading___Toc2098_2331702048)

[2.3 本章小结 12](#__RefHeading___Toc2100_2331702048)

[第三章 数据库存储的设计与实现 12](#__RefHeading___Toc2102_2331702048)

[3.1 B+树的设计与实现 12](#__RefHeading___Toc2104_2331702048)

[3.1.1 设计目标 12](#__RefHeading___Toc2106_2331702048)

[3.1.2 概要设计 12](#__RefHeading___Toc2108_2331702048)

[3.1.2.1缓存方案 13](#__RefHeading___Toc2110_2331702048)

[3.1.2.2重做日志 13](#__RefHeading___Toc2112_2331702048)

[3.1.2.3内存操作 13](#__RefHeading___Toc2114_2331702048)

[3.1.2.4持久化 14](#__RefHeading___Toc2116_2331702048)

[3.1.2.5从磁盘中读入 17](#__RefHeading___Toc2118_2331702048)

[3.1.2.6 故障恢复 17](#__RefHeading___Toc2120_2331702048)

[3.1.2.7碎片整理 18](#__RefHeading___Toc2122_2331702048)

[3.1.2.8启动流程 18](#__RefHeading___Toc2124_2331702048)

[3.1.3 数据存储协议 18](#__RefHeading___Toc2126_2331702048)

[3.1.3.1 元数据页结构 19](#__RefHeading___Toc2128_2331702048)

[3.1.3.2非叶子节点数据页结构 20](#__RefHeading___Toc2130_2331702048)

[3.1.3.3叶子节点数据页结构 22](#__RefHeading___Toc2132_2331702048)

[3.1.3.4 重做日志文件 22](#__RefHeading___Toc2134_2331702048)

[3.2 Hash存储的设计与实现 23](#__RefHeading___Toc2136_2331702048)

[3.3 本章小结 23](#__RefHeading___Toc2138_2331702048)

[第四章 实现数据库 23](#__RefHeading___Toc2140_2331702048)

[4.1 目标 24](#__RefHeading___Toc2142_2331702048)

[4.2 存储方案 24](#__RefHeading___Toc2144_2331702048)

[4.2.1 元数据存储方案 24](#__RefHeading___Toc2146_2331702048)

[4.2.2索引存储方案 24](#__RefHeading___Toc2148_2331702048)

[4.2.3 数据存储方案 24](#__RefHeading___Toc2150_2331702048)

[4.2.4支持的数据类型 25](#__RefHeading___Toc2152_2331702048)

[4.2.5 基本操作 25](#__RefHeading___Toc2154_2331702048)

[4.2.5.1 创建一个简单DBMS 25](#__RefHeading___Toc2156_2331702048)

[4.2.5.2 加载简单DBMS 26](#__RefHeading___Toc2158_2331702048)

[4.2.5.3创建数据库 26](#__RefHeading___Toc2160_2331702048)

[4.2.5.4 删除数据库 26](#__RefHeading___Toc2162_2331702048)

[4.2.5.5创建表 26](#__RefHeading___Toc2164_2331702048)

[4.2.5.6删除表 26](#__RefHeading___Toc2166_2331702048)

[4.2.5.7 插入记录 26](#__RefHeading___Toc2168_2331702048)

[4.2.5.8查询记录 27](#__RefHeading___Toc2170_2331702048)

[4.2.5.9 删除记录 27](#__RefHeading___Toc2172_2331702048)

[4.2.5.10 更新记录 27](#__RefHeading___Toc2174_2331702048)

[4.3 并发控制 27](#__RefHeading___Toc2176_2331702048)

[4.4 本章小结 28](#__RefHeading___Toc2178_2331702048)

[第五章 成果展示 29](#__RefHeading___Toc2754_1108143602)

[5.1 项目简介 29](#__RefHeading___Toc2756_1108143602)

[5.2 具体实现 30](#__RefHeading___Toc2758_1108143602)

[5.3 效果演示 31](#__RefHeading___Toc2760_1108143602)

[5.3.1 命令简介 31](#__RefHeading___Toc2762_1108143602)

[5.2.2支持命令和sql 31](#__RefHeading___Toc2764_1108143602)

[5.2.3样例 31](#__RefHeading___Toc2766_1108143602)

[第四章 全文总结与展望 33](#__RefHeading___Toc2194_2331702048)

[4.1 全文总结 33](#__RefHeading___Toc2196_2331702048)

[4.2 后续工作展望 33](#__RefHeading___Toc2198_2331702048)

[致 谢 34](#__RefHeading___Toc2200_2331702048)

[参考文献 35](#__RefHeading___Toc2202_2331702048)

第一章 绪 论

1.1 研究工作的背景与意义

数据库按照单机系统可以分为单机数据库、分布式数据库两大类。按照是关系存储分为关系型数据库（Sql）、非关系型数据库（NoSql）。关系型数据库出现较早，应用最为广泛，市场上存在众多实现：MySql、Sql Server、Oracle等。随着互联网的发展，NoSql数据库开始流行：HBase、Redis、MongoDB等。近些年来有些结合NoSql和Sql共同有点的被称为NewSQL数据库开始出现如：TiDB、 CockRoachDB等。

数据库是信息系统数据存储的最重要方式，了解数据库的实现原理，各种实现原理的性能特点尤为重要。此为，随着数据量的猛增，对于同时能支持OLTP和OLAP的数据库必要。

1.2 数据库技术的国内外研究历史与现状

数据库技术的发展与应用 从20世纪60年代末期开始到现在，数据库技术已经发展了30多年。在这30多年的历程中，人们在数据库技术的理论研究和系统开发上都取得了辉煌的成就，而且已经开始对新一代数据库系统的深入研究。数据库系统已经成为现代计算机系统的重要组成部分。

20世纪60年代中期，数据库技术是用来解决文件处理系统问题的。当时的数据库处理技术还很脆弱，常常发生应用不能提交的情况。20世纪70年代关系模型的诞生为数据库专家提供了构造和处理数据库的标准方法，推动了关系数据库的发展和应用。1979年，Ashton-Tate公司引入了微机产品dBase Ⅱ，并称之为关系数据库管理系统，从此数据库技术移植到了个人计算机上。20世纪80年代中期到后期，终端用户开始使用局域网技术将独立的计算机连接成网络，终端之间共享数据库，形成了一种新型的多用户数据处理，称为客户机/服务器数据库结构。现在，数据库技术正在被用来同Internet技术相结合，以便在机构内联网、部门局域网甚至WWW上发布数据库数据。

1.3 本文的主要贡献与创新

本论文以 数据库底层底层的两大实现原理树结构和Hash结构为重点研究内容，并研究重做日志，内存缓存技术，主要创新点与贡献如下：

* 设计实现了，B+Tree存储引擎，用于存储索引，支持O(logn)的插入查询，支持顺序查找、精确查找。
* 设计实现了，Hash存储引擎，用于存储数据，支持O(1)插入查找，支持精确查找。

1.4 本论文的结构安排

本文的章节结构安排如下：

* 本文采用从下而上的结构安排内容
* 第二章 介绍数据库底层存储的基本原理
* 第三章 介绍数据库底层存储的设计与实现
* 第四章 站在宏观的基础上如何利用第三章实现的基础实现数据库
* 第五章 展示成果

第二章 数据库储存基本原理

数据库底层存储方案落到树结构上不外乎两种：树形结构、Hash结果。由于磁盘的特性树形结构一般主流实现有Btree、LSM两种。本章主要介绍B+Tree和Hash存储的原理

2.1 B+树实现原理

2.1.1 B+树简介

B+树一种数据结构，一般用作数据库索引的存储结构，是一颗多叉树。 结合了有序数组（范围查找）和ALV（插入删除效率高）好处，特性如下

* 支持精确查找
* 支持范围查找
* 局部性较好

2.1.2 B+树的定义

假设树的度为m，B+树的定义如下

* m>=3（若m==2，树会退化为链表；m==1，树无法形成）
* 同一个节点内按照key从小到大排序
* 叶子节点包含 数目相同的key和value，和一个指向右孩子的指针next
  + key[i]与value[i]一一对应
  + 所有的叶子节点在同一层
* 非叶子节点包含 数目相同key和pointer（指向孩子的指针）
  + key[i]子树为pointer[i]，满足pointer[i]这棵树的key的取值范围为[key[i], key[i+1])
* 根节点
  + 若是叶子节点 key的数目取值范围为：[0, m]
  + 若是非叶子节点 key的取值范围为：[(m+1)/2, m] （(m+1)/2等价于m/2向上取整）
* 非根节点 key的数目取值范围为：[(m+1)/2, m]

满足以上性质的多叉树为B+树

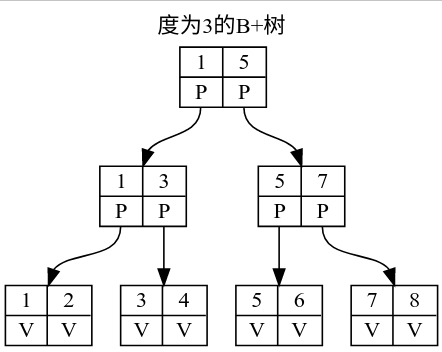
相关术语

* 树节点的度：等于节点孩子的数目
* 树的度：树中所有节点的度的最大值

2.1.3 B+树的性能

* 查询效率O(logn)
* 插入效率O(logn)
* 删除效率O(logn)
* 空间利用率大于1/2

2.1.4 B+树的例子



2.1.4 B+树的基本操作

2.1.4.1 查询

给一个颗B+树和一个key，查询对应的value

1. 开始
2. p = 根节点
3. 对p.key数组进行二分查找，找到第一个小于等于key的元素的下标i
4. 若p是叶子节点直接返回p.value[i]
5. 若p不是叶子节点 p=p.children[i]，跳到第2步
6. 结束

2.1.4.2 插入

给一个颗B+树和一个<key,value>，将其插入B+树，这是个递归算法

* 定义函数insert，接收树的节点和键值对
* 对根节点调用insert，伪代码如下

Node insert(now, key, value){

if(now is 叶子节点){

将<key,value>插入到now相应的位置，使now.key数组有序

} else {

对now.key数组进行二分查找，找到第一个小于等于key的元素的下标i

new = insert(now.children[i],key,value)

if(new!=NULL)

将<new.key[0],new>插入到now相应的位置，使now.key数组有序

}

if(now元素数>树的度)

将now后一半的元素放到一个新的Node中，并返回

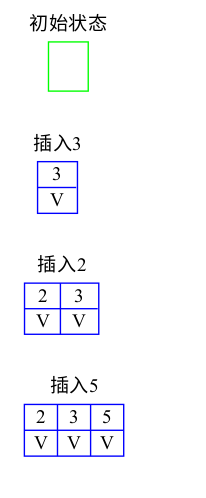
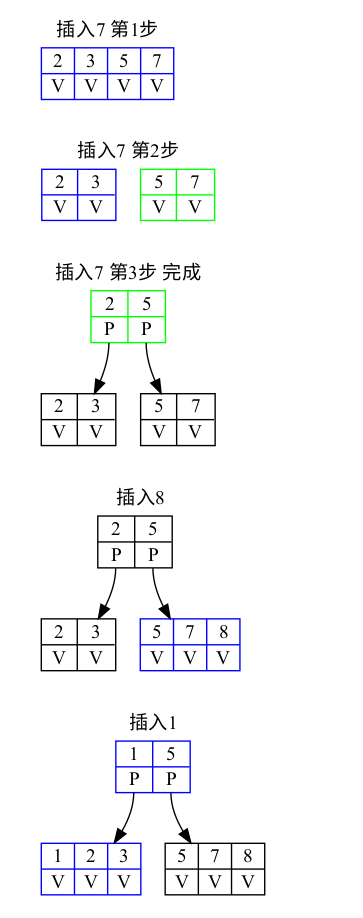
返回NULL

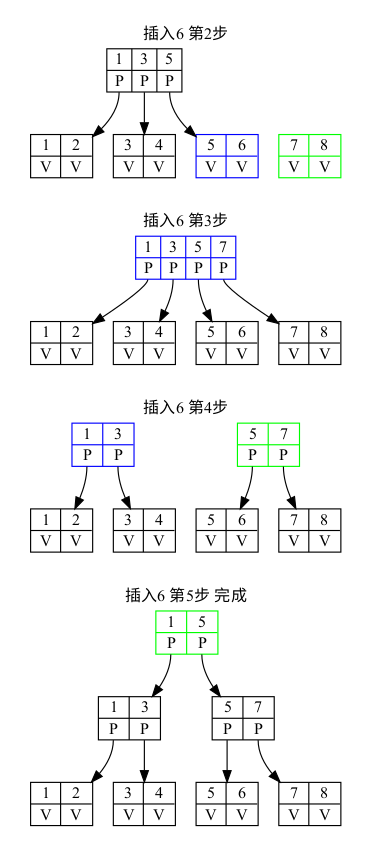
}

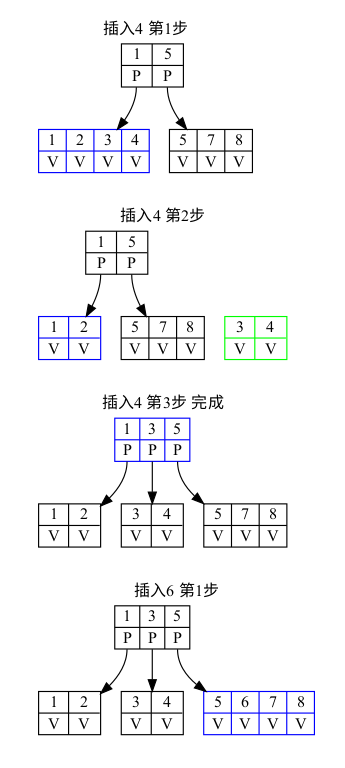
例子：度为3的B+树插入key为[3,2,5,7,8,1,4,6]的过程

图示说明：

* 红色节点表示这一步已经被删除
* 绿色节点表示相对上一步新创建
* 蓝色节点表示相对上一步发生修改
* 黑色节点表示相对上一步没有发生变化







2.1.4.3删除

给一个颗B+树和一个`key`，将其从B+树中删除

这是个递归算法

1. 定义函数`remove`，接收树的节点和键值对

2. 对根节点调用`remove`，伪代码如下

int remove(now, key){

if(now is 叶子节点){

将key从now中删除

return 1

} else {

对now.key数组进行二分查找，找到第一个小于等于key的元素的下标i

cnt = remove(now.children[i],key)

If(now.children[i].size过少不满足树的定义){

if(now.children[i]与其左兄弟的节点数均衡下满足树的定义){

均衡下元素，返回cnt

} else {

now.children[i]与其左兄弟合并成一个节点;并删除now一个引用

返回cnt

}

}

}

}

2.2 Hash存储原理

2.2.1 Hash存储原理

Hash存储一种数据结构，一般作为KV存储的结构，特性如下

* 支持精确查找

2.2.2 Hash存储的性能

* 查询效率O(1)
* 插入效率O(1)
* 删除效率O(1)

2.2.3 Hash存储的基本操作

2.1.4.1 查询**查询**

给一个颗B+树和一个key，查询对应的value

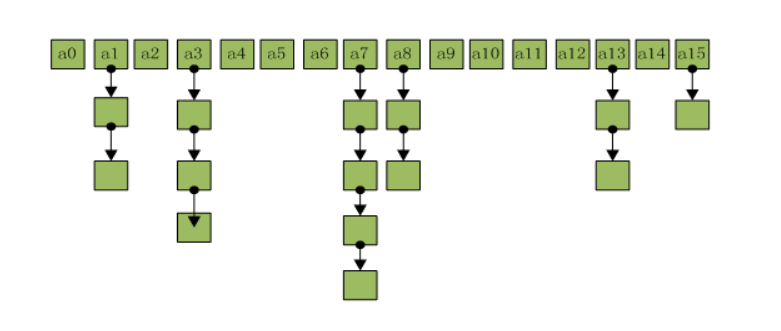
1. 通过预设的Hash函数计算Hash值

2. 与Hash桶数组长度求模

3. 插入到指定的位置

2.1.4.1 其他

略

2.2.4 Hash存储示意图

2.3 本章小结

本章从数据库底层存存储发原理出发，阐述了数据库两种数据存储方案的基本原理

第三章 数据库存储的设计与实现

3.1 B+树的设计与实现

3.1.1 设计目标

* 数据持久化到磁盘中
* 支持可配置的固定长度的key
* 查询效率要高，使用缓存
  + 支持精确查找，时间复杂度O(logn)
  + 支持范围查找，时间复杂度O(logn+m)
* 插入效率要高
  + 支持插入删除
  + 支持任意时刻断电恢复
  + 故障允许短时间或少量数据的丢失
* 文件大小理论上无限值（最大2^64）

3.1.2 概要设计

使用B+树来构建索引，操作磁盘的粒度为页（默认16k），也就是B+树的一个结点

由于写回磁盘的代价过高，所以采用异步的方式将**修改**或**新增的节点**写回磁盘

为了简化设计，每个索引文件对应一个进程处理；本文描述的都是针对单个索引文件的操作。

3.1.2.1缓存方案

在进行数据（<K,V>）操作（插入/修改/删除）时，必然涉及读写数据。这里使用三个两类LRU缓存（下文称LRU）进行磁盘数据的缓存：

* 第一类缓存（unchangedCache）：存放在执行一系列操作过程中没有发生更改的节点（也就是与磁盘中的数据一致的数据）。
  + 这样的LRU需要1个。
  + 这些数据都是从磁盘中读取而来的
* 第二类缓存（changedCache）：存放发生修改的节点（也就是与磁盘不一致或者磁盘不存在的数据）
  + 这样的LRU需要2个，每次使用1个，当一个满了之后，将会异步做持久化操作，另一个启用工作
  + 这些数据都是unchangedCache发生修改的数据在淘汰时添加而来的
  + 工作中的LRU叫做changeCacheWork
  + 持久化中的LRU叫做changeCacheFreeze

3.1.2.2重做日志

同时由于数据延时异步刷磁盘，所以要使用重做日志（Redo log）防止数据丢失

* 每个第二类缓存对应一个重做日志文件，
* 当一个第二类缓存在使用中，对象的写操作将追加到重做日志中
* 当一个第二类缓存持久化完成后就可以清空对应的重做日志

3.1.2.3内存操作

针对操作大概流程如下

* 收到一个操作
* 一、先读changeCache
* changeCacheFreeze查找操作节点，存在则拷贝一份
  + 若发生更改，放入changeCacheWork
  + 若没有发生更改忽略
* changeCacheFreeze不存在
* changeCacheWork中查找，存在，返回该节点引用，直接修改即可
* changeCacheWork不存在
* 向unchangeCache中查找，存在，返回该节点引用，直接修改即可
* unchangeCache不存在
* 二、再读unchangeCache
* 从磁盘中读入，插入unchangeCache并返回
  + 插入过程中若发生淘汰，淘汰的节点若发生修改则插入changeCacheWork
  + 当changeCacheWork满，且changeCacheFreeze持久化完成，则交换两者身份后，异步对changeCacheFreeze进行持久化

注意所有发生修改的节点都要分配新的页Id

3.1.2.4持久化

持久化过程的思路是，在持久化过程中，要维护两棵树在磁盘中。这样若在持久化过程中发生故障；在启动后可以通过原树+重做日志恢复数据。

既然需要维护两棵树，则每个逻辑上的B+tree节点在磁盘中可能存在两个页，每个页都包含数据，同一时刻这两个页有一个是有效数据，另外的为无效数据。持久化版本号大的为有效数据。这两个页分为一下两种：

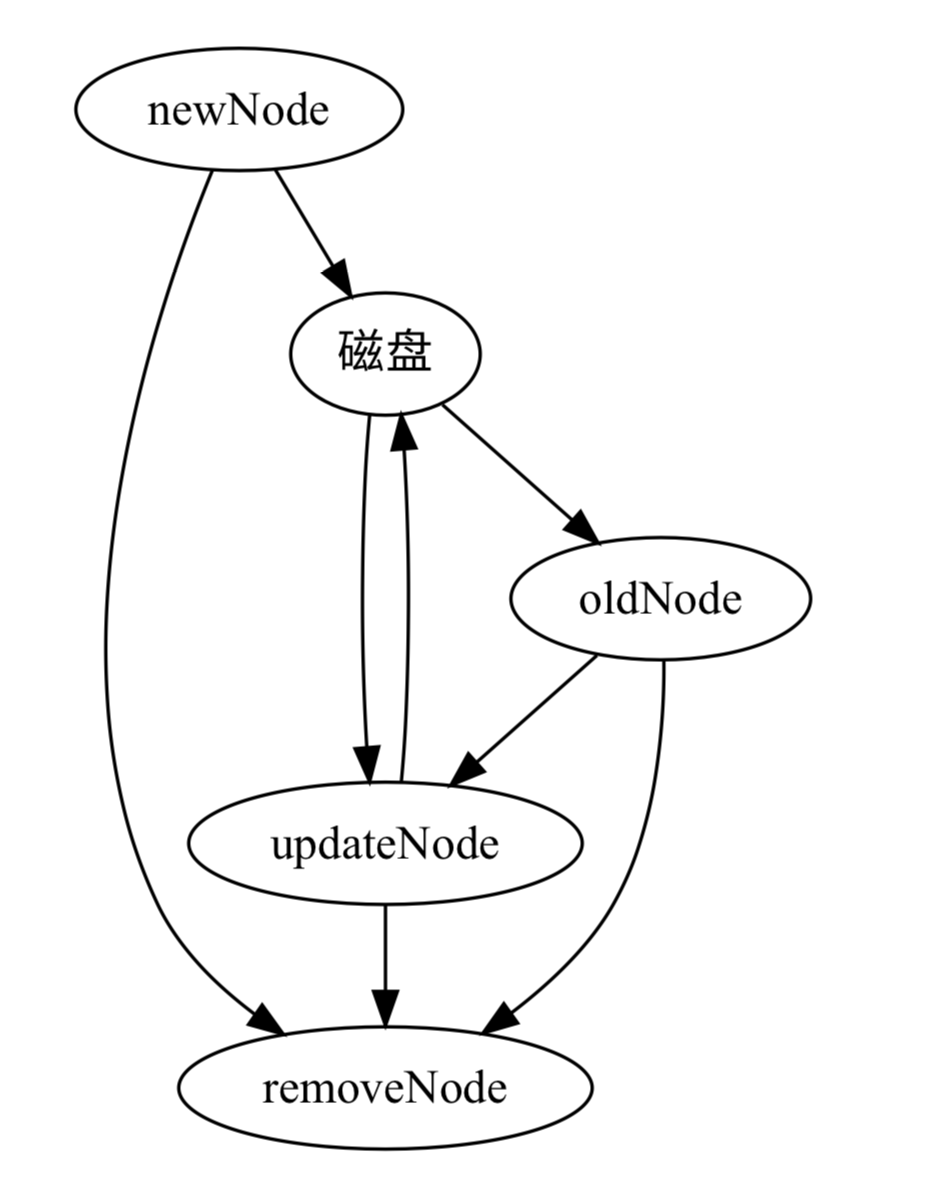
* 被链接页 该节点的父亲直接指向该页
* 影子页 该节点的必须通过链接页的after字段找到页

unchangeCache仅存放一种节点：oldNode

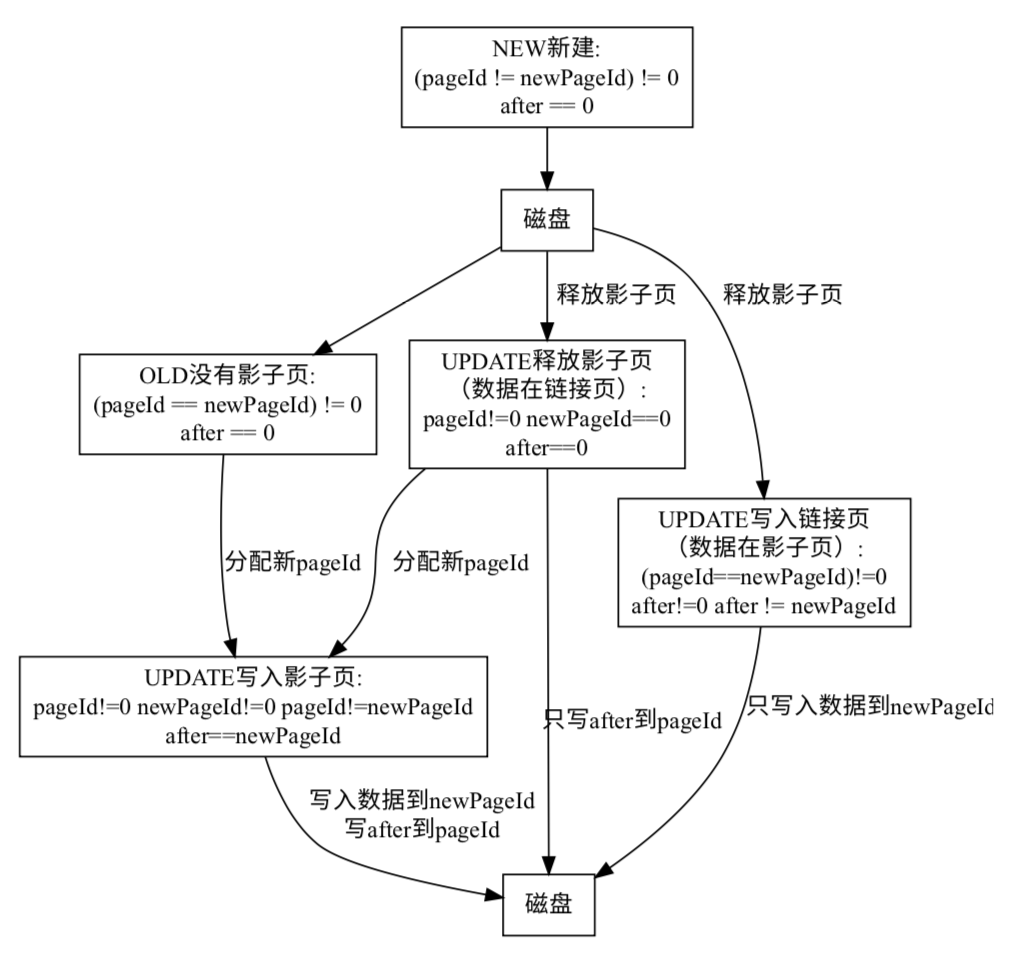
changeCacheFreeze中的节点分为3类：

* 第一类节点（newNode）：新创建的节点，内存存在，磁盘不存在
* 第二类节点（updateNode）：更新的节点，内存存在新的，磁盘存在旧的
* 第三类节点（removeNode）：被删除的节点，内存存被标记删除，磁盘存在，不会对磁盘有任何操作

状态转换图如下：



详细转换图如下（为了美观，REMOVE状态未绘制出，参见上图）：



changeCacheFreeze两种状态：

* 第一种状态：正在进行持久化
  + 此时不能进行两个changeCache的切换
  + 且必须先读changeCacheFreeze再读changeCacheWork
* 第二种状态：可以进行持久化
  + 此时可以进行两个changeCache的切换
  + 不需要读changeCacheFreeze，直接读changeCacheWork

每一次持久化都会创建一个递增版本号，所有新增加的页都会写入这个版本号（nodeVersion），用于故障恢复。

大概执行过程为

* 修改磁盘中文件某字段标记正在进行持久化
* 需要对changeCacheFreeze进行几次遍历
* 第一次遍历：
  + 将所有newNode写入磁盘
* 第二次遍历：
  + 将所有updateNode
  + 将数据写入无效数据页，
  + 修改after指针
* 将磁盘中文件的部分元数据备份到备份区域
* 修改磁盘中文件某字段标记为正在进行元数据修改
* 修改元数据（切换树，因为有两个树）
* 恢复所有标记字段

3.1.2.5从磁盘中读入

需要注意的是，根据版本号（nodeVersion），决定读取当前页还是effect页，则直接读effect指向的页

* 如果 effect==0 返回当前页
* 否则，如果 effect.nodeVersion>=meta.nextNodeVersion 返回effect页

3.1.2.6 故障恢复

故障分为几种情况

* 没有线程进行持久化操作：
  + 直接执行重做日志
* 线程持久化操作中，但没有进行修改元数据：
  + 遍历，将所有遍历到的废弃标记清空
  + 直接执行重做日志
* 线程持久化操作中，正在元数据修改：
  + 从元数据备份区进行元数据恢复
  + 遍历，将所有遍历到的废弃标记清空
  + 直接执行重做日志

3.1.2.7碎片整理

由于持久化操作会产生大量副本，且删除仅仅标记为废弃，所以磁盘文件会无限膨胀，所以需要进行文件碎片整理

碎片整理有两个方案：

* 方案一：遍历树，将所有废弃节点创建一个可用页链表
* 方案二：在方案一的基础上进行数据迁移

为了效率，选择方案一

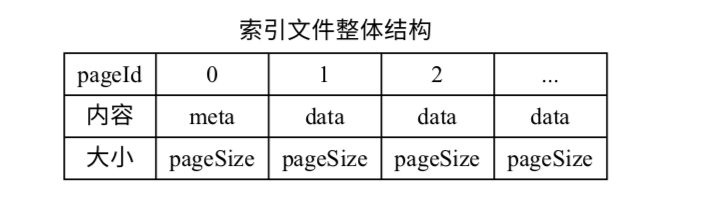
3.1.2.8启动流程

* 检测状态，进行故障恢复
* 执行重做日志
* 进行碎片整理

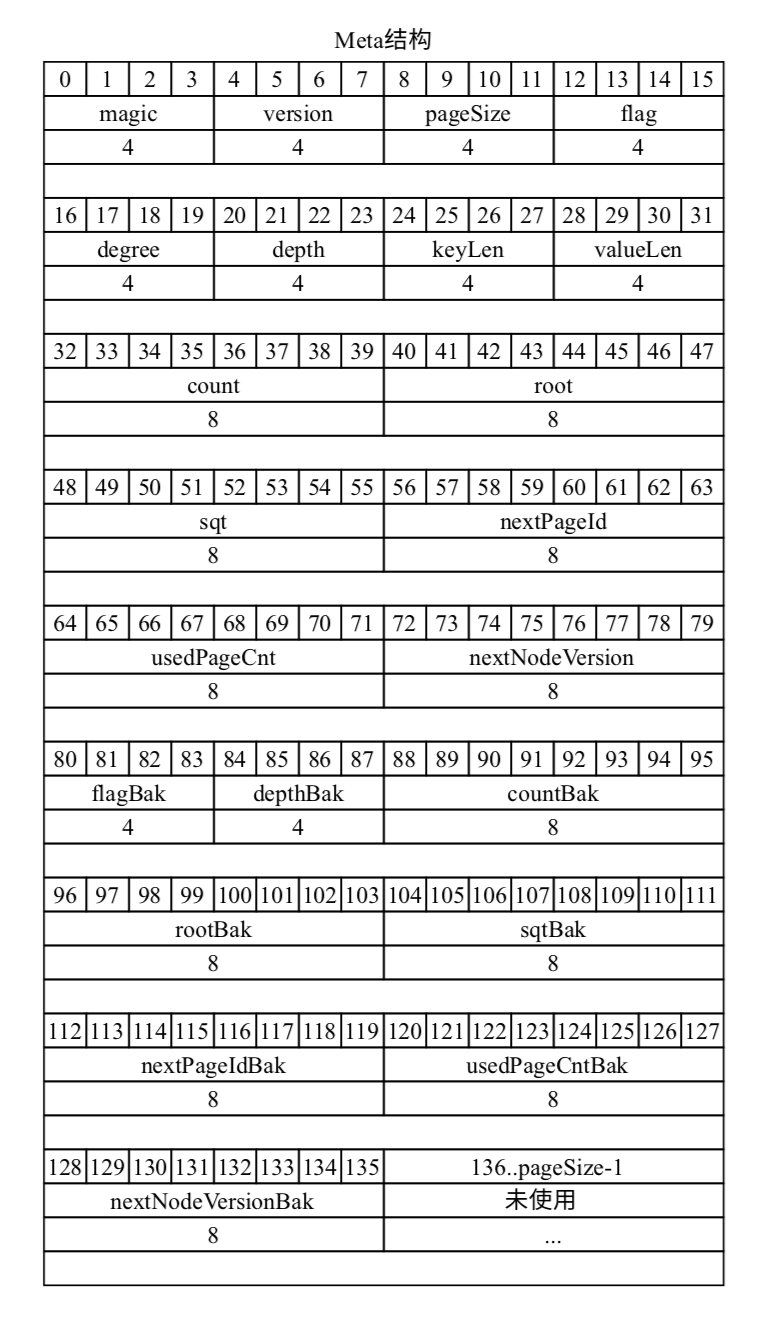
3.1.3 数据存储协议

使用B+树数据结构

* 索引文件划分为多个连续的页（Page）
* 每个页大小可配置（pageSize，默认16k），取值范围为[64,2^64]
* 每个页按顺序进行编号，从0开始
* 第0页，存放元数据（meta）
* 其他页存放数据（data）
* 所有数据以网络字节序存储（大端）



3.1.3.1 元数据页结构

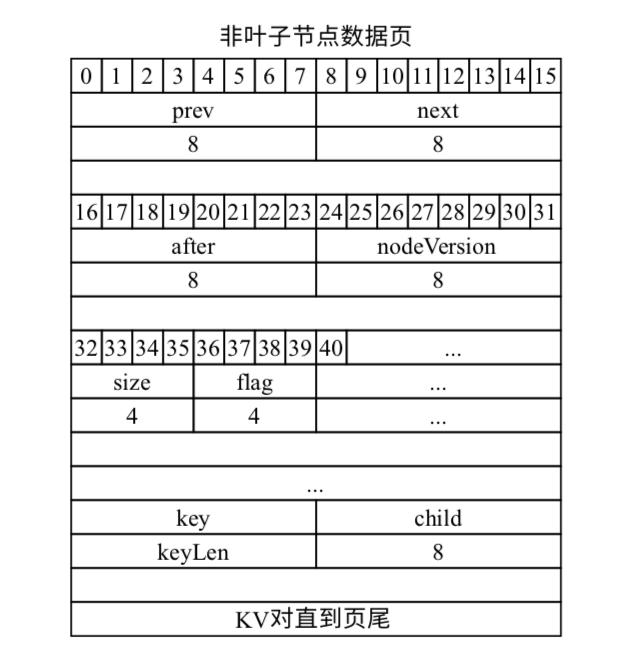




* magic 4字节 魔数 0x960729db 表示该文件是索引存储文件
* version 4字节 文件版本号 目前为 1
* pageSize 4字节 页大小 默认为 16k
* flag 4字节 标志
  + flag[0] isUnique 表示该索引文件是否唯一
  + flag[1] isPersistence 是否正在进行持久化
  + flag[2] isSwitchTree 是否正在进行切换树
  + flag[3] isCreating 是否正在进行创建文件
  + flag[31..4]未定义
* degree 4字节 B+树的度，根据pageSize计算和data页结构计算
* depth 4字节 树的深度，用于判断树叶子节点
* keyLen 4字节 键字节数 长度，简单起见 小于 (页长度-链接数据页控制字段)/3
* valueLen 4字节 值字节数，简单起见 小于 (页长度-叶子数据页控制字段-keyLen)
* count 8字节 该索引数据计数
* root 8字节 根节点所在的页
* sqt 8字节 第一个叶子节点所在的页
* nextPageId 8字节 下一个可用页的号
* usedPageCnt 8字节 [1, nextPageId)已将使用的页的数目，结合nextPageId在达到一定情况下进行自动磁盘整理
* nextNodeVersion 8字节 下一次磁盘持久化使用的版本号
* xxxBak 重要数据的备份，用于断电恢复

注意：keyLen和valueLen的限制是为保证一个页至少放下一对KV

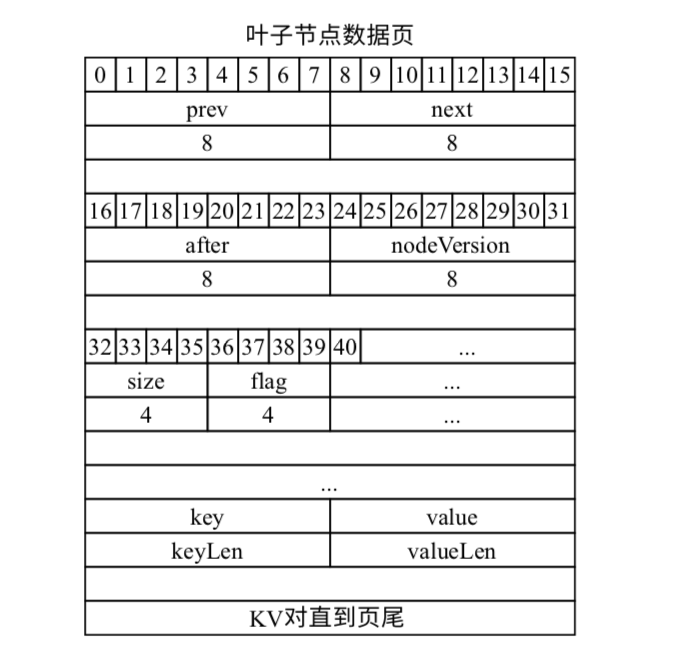
#### 3.1.3.2非叶子节点数据页结构



* prev 8字节 表示左兄弟所在的页
* next 8字节 表示右兄弟所在的页
* after 8字节
  + 0 没有影子节点
  + !0 影子节点所在位置
* nodeVersion 8字节 该节点在持久化的版本号
* size 4字节 该页已使用的key的数目，不能大于degree
* flag 4字节 标志
  + flag[0] 该页是否被废弃
  + flag[31..1]未定义
* key keyLen字节 键
* child 8字节 键指向孩子所在的页

3.1.3.3叶子节点数据页结构

简单起见规定一个页能放下所有的KV对



与非叶子节点数据页区别在于：value valueLen字节 存放的是数据

3.1.3.4 重做日志文件

一个索引文件对应0个或1个重做日志。

重做日志文件名为索引文件名\_+持久化版本号+.redolog，其中持久化版本号是一个64位无符号整数的16进制形式。例如：test\_0x0000000000000002.redolog

重做日志的作用是当发生断电时进行数据恢复。

重做日志文件是一个顺序结构文件，由多个操作元组组成。每个操作元组结构如下：



* type 1字节 表示操作类型
  + 0 无效操作
  + 1 表示插入，此时后面包含key和value
  + 2 表示移除，此时后面仅包含一个key字段
  + 3 表示精确移除，此时后面包含key和value
* key keyLen字节 必选 代操作的key
* value valueLen字节 可选 代表待操作的值

3.2 Hash存储的设计与实现

篇幅有限，参见附件

3.3 本章小结

本章从实现出发，阐述了B+树的相关数据结构，实现细节，存储协议。

1. 实现数据库

大型的关系型数据库一般使用C/S 架构，即客户端服务器模式。一些小型的嵌入式数据库也有使用客户端服务端一体的架构。本文目的是探索数据库的实现原理特定，所以我使用客户端服务端一体的架构。

4.1 目标

* 单进程整合实现
* 支持表的创建删除
* 字段支持多种类型
* 提供线程安全的记录增删改查
* 所有操作不支持修改,仅支持创建删除

4.2 存储方案

4.2.1 元数据存储方案

* 存储在HashEngine中
* 文件名为metadata.hashengine
* 定义如下key
  + database\_list 存放数据库列表
  + ${databaseName}\_database 存放该数据库表列表
  + ${databaseName}\_${tableName}\_table存放字段定义列表

4.2.2索引存储方案

* 存储在HashEngine
* 文件名为${databaseName}\_${table}\_${field}.indexengine
* key为 field 名, 长度等于field定义的长度, 若为字符串类型, 不足的补零
* value为 主键值, 长度等于field定义的长度, 若为字符串类型, 不足的补零

4.2.3 数据存储方案

* 存储在HashEngine中
* 文件名为${databaseName}\_${table}.hashengine
* key 为 主键, 长度等于field定义的长度
* value 一条记录
  + 由value组成, 与field一一对应
  + 数字类型直接存储即可
  + 字符串类型存储方案为 length:4字节+字节数组

4.2.4支持的数据类型

* 无符号整数
  + type = 1
  + length = ?
* 有符号整数
  + type = 2
  + length = ?
* 可变字符串
  + type = 3
  + maxLength = ?

4.2.5 基本操作

4.2.5.1 创建一个简单DBMS

* 创建基本结构
  + databaseMap 数据库名->HashMap<表名, List<字段定义>>
  + dataMap 文件名->Hash引擎的一个映射
  + indexMap 文件名->索引引擎的一个映射
* 在dirpath中
  + 创建一个hashengine文件名为metadata.hashengine
  + 并将其放入dataMap中
  + 该结构用于持久化databaseMap

4.2.5.2 加载简单DBMS

* 创建基本结构
* 读取metadata.hashengine
* 重新构建databaseMap
* 根据databaseMap中的情况加载Hash引擎和索引引擎

4.2.5.3创建数据库

* 检测databaseMap中是否存同名的数据库
* 若不存在, 则databaseMap中插入一个记录<数据库名, HashMap>
* 将databaseMap持久化到metadata.hashengine中

4.2.5.4 删除数据库

* 递归删除数据库下面的索引引擎和Hash引擎
* 将databaseMap持久化到metadata.hashengine中

4.2.5.5创建表

* 检测databaseMap中是否存在数据库
* 检测数据库中是否存在同名的表
* 若不存在, 则在数据库中插入一条记录
* 将databaseMap持久化到metadata.hashengine中

4.2.5.6删除表

* 递归删除表相关的文件和结构
* 将databaseMap持久化到metadata.hashengine中

4.2.5.7 插入记录

* 获取到字段定义
* 将数据转化为网络字节序
* 将相关索引字段插入到索引引擎
* 将数据插入到Hash引擎
* 完成

4.2.5.8查询记录

* 解析条件
* 如果需要查询索引
* 查询索引引擎获取到主键列表
* 查询Hash引擎获取到记录
* 如果是覆盖索引，直接返回
* 否者就再使用过滤条件进行过滤，然后返回

4.2.5.9 删除记录

* 调用查询记录函数获取到记录列表
* 删除索引
* 删除数据

4.2.5.10 更新记录

* 删除记录
* 插入记录

4.3 并发控制

为了保证并发安全，我们使用表锁来实现并发控制。每一章表对应一个互斥锁，多表的所有操作都需要获取该锁，否则阻塞等待

4.4 本章小结

本章讨论如何利用前两章实现的技术设施来实现一个简单的数据库。

第五章 成果展示

5.1 项目简介

开发语言：C语言

API：Linux Posix

编译运行环境：Linux

编译器：GCC

构建工具：make

版本控制工具：git

项目地址：<https://github.com/rectcircle/course-design-DBMS>

目录结构

.

├── doc

├── include

├── LICENSE

├── Makefile

├── make.sh

├── out

│ ├── bin

│ ├── obj

│ │ ├── debug

│ │ └── release

│ └── test

├── README.md

└── src

├── main

└── test

doc 文档

* include 头文件
* LICENSE 源码许可证
* Makefile Makefile文件
* [make.sh](http://make.sh/)~~编译脚本，已废弃，请忽略~~
* out 编译输出目录
  + bin 可执行文件目录
  + obj 目标对象目录
    - debug 包含调试信息的目标对象
    - release 不包含调试信息的目标对象
  + test 测试的可执行文件
* [README.md](http://readme.md/)
* src 源文件目录，该目录下的文件不包含main函数
  + main 包含main函数的c程序，命名以main-开头
  + test 包含main函数的测试源文件，命名以test-开头

模块划分 ：

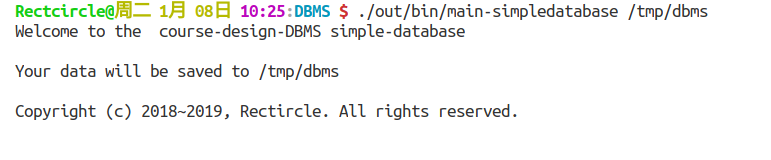
* 工具模块
  + hashmap
  + lrucache LRU内存缓存
  + array 带长度的字节数组
  + list 单链表
  + test 单元测试套件
* redolog.h 重做日志实现
* indexengine.h 索引引擎实现（磁盘B+树）
* hashengine.h hash引擎实现
* sampledatabase.h 简单的数据库实现

5.2 具体实现

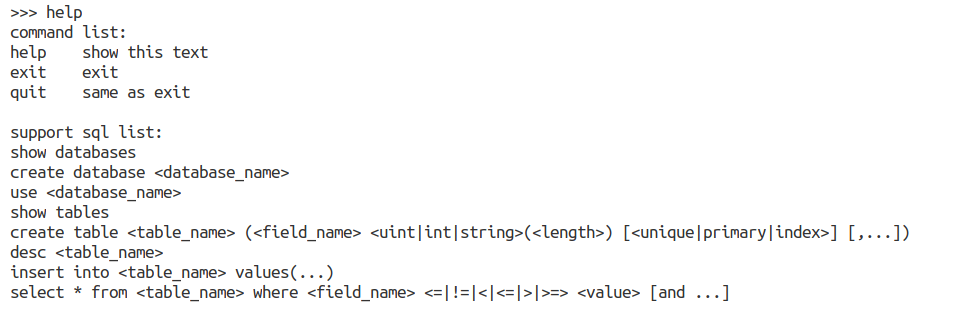
参见代码仓库

5.3 效果演示

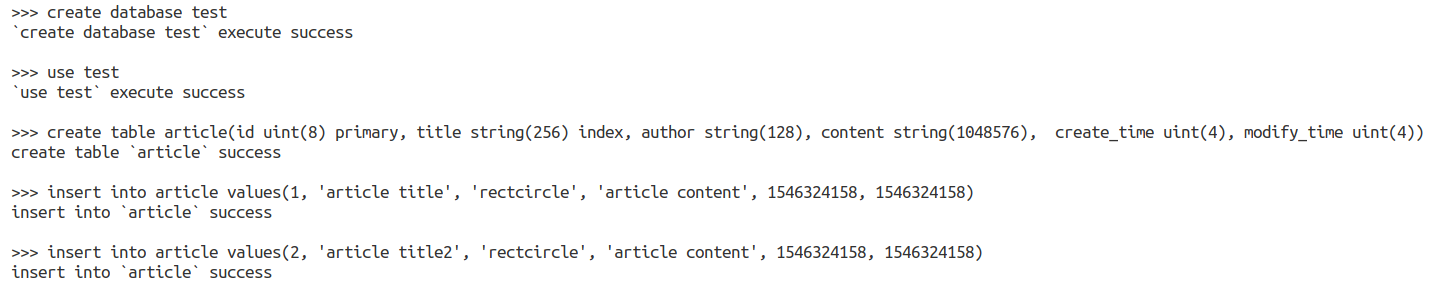
5.3.1 命令简介

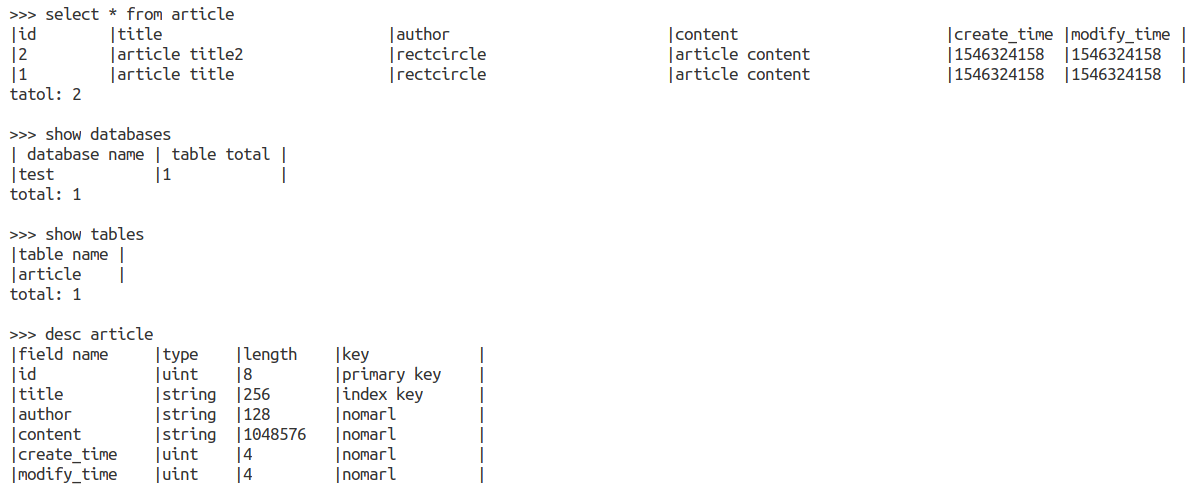


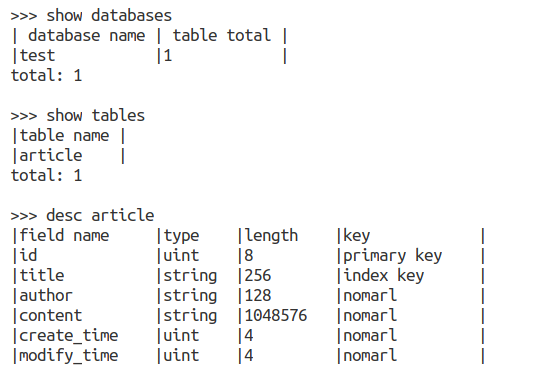
5.2.2支持命令和sql

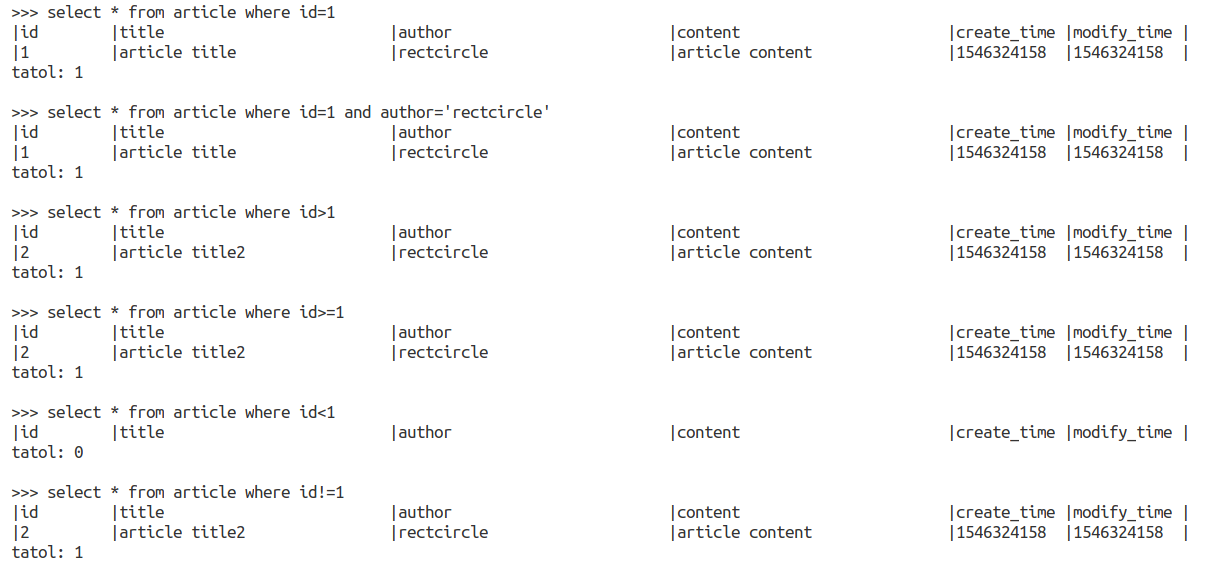


5.2.3样例







第四章 全文总结与展望

4.1 全文总结

本文以数据库的底层实现为研究背景，主要探索了B+树和Hash存储的实现。

4.2 后续工作展望

未来可以利用已将实现的B+树和Hash存储，进一步探索事务管理，分布式数据库与各种网络模型。将分布式理论与数据库理论相结合。

致 谢

本论文的工作是在我们的小组成员的齐心协力之下，完成的感谢大家的付出。

参考文献

1. 旧金山大学数据结构可视化 https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
2. B+Tree百度百科 https://baike.baidu.com/item/B%2B%E6%A0%9
3. 杨传辉,大规模分布式存储系统：原理解析与架构实战, 机械工业出版社,2013