摘 要

随着半导体技术的发展，内存的价格越来越低，内存的容量越来越大。这使得将数据库放置在内存当中变成了可能。再加上应用程序对于数据读写的实时性要求不断提高，内存数据库系统变得越来越重要。在这样一个时刻一款内存数据库管理系统svimrdb应运而生。因为该内存关系数据库管理系统基于StoneValley库，所以名为svimrdb，意为StoneValley in-memory relational database。

本文对 svimrdb 内存数据库管理系统的原理设计做出了详细的阐述。并对 svimrdb 的代码实现做了细致的解释。当今，计算机科学与技术发展趋于成熟。计算机编程语言尤其是 C 语言发展稳定。基于 C 语言的 cstdlib 库也变得功能强大。操作系统的发展使得内存管理变得容易。这些技术都给 svimrdb 的出现打下了坚实的基础。svimrdb 内存数据库管理系统也在不断更新演进中趋于强大和稳健。文章的上半部分介绍了内存数据库管理系统的概念及基本原理。在下半部分，文章抽丝剥茧地深入剖析了 svimrdb 的源代码和实现过程，致力于为内存数据库管理系统的发展添砖加瓦。

关键词：内存数据库，数据库管理系统，程序库，svimrdb

Abstract

With the development of semiconductor technology, the price of memory is getting lower and lower, and the capacity of memory is getting larger and larger. This makes it possible to place the database in memory. Coupled with the increasing real-time requirements for data reads and writes, in-memory database systems are becoming more and more important. At such a moment, an in-memory database management system, SVIMRDB, came into being. Because of this in-memory relational database management system is based on StoneValley library, it is named as SVIMRDB. The meaning of SVIMRDB is StoneValley in-memory relational database.

This paper elaborates on the principle design of SVIMRDB in-memory database management system. The code implementation of SVIMRDB is explained in detail. Nowadays, the development of computer science and technology is becoming more mature. Computer programming languages, especially C, have developed steadily. The C-based cstdlib library has also become powerful. The development of operating systems has made memory management easy. These technologies have laid a solid foundation for the emergence of SVIMRDB. The SVIMRDB in-memory database management system is also becoming more powerful and robust in the process of continuous update and evolution. The first half of the article introduces the concept and basic principles of an in-memory database management system. In the second half, the article provides an in-depth analysis of the source code and implementation process of SVIMRDB, and is committed to contributing to the development of in-memory database management system.

Keywords: In-memory Databases, Database Management System, Programming Library, SVIMRDB

目 录

1 绪论

1.1 内存数据库的概念

内存数据库的概念出现在二十世纪六十年代末。将数据库中的全部数据或者大部分数据存放在内存中就是内存数据库。最开始，内存数据库主要作用于嵌入式系统以及磁盘数据库系统的辅助系统或者加速系统。在当时，内存容量小、单位价格高。因此还未出现成型的专门的内存数据库系统。

在内存数据库中磁盘作为永久存储数据的后备存储设备，内存则作为常规存储设备。根据内存的工作特性，内存中的数据断电后就会消失。处理器可以直接访问内存数据库中的数据。与磁盘数据库不同的是，内存数据库中的数据长期驻留在内存中。内存数据库没有磁盘数据库中输入/输出的巨大开销。在内存数据库中使用了针对内存特性进行优化的各种存储结构、索引结构还有操作算法。这样做进一步提升了内存数据库的性能。

内存数据库具有一些优越性，尤其是在需要快速访问和处理大量数据的场景中。以下是内存数据库的一些优势：

快速访问速度： 内存数据库将数据存储在 RAM 中，可以实现非常快速的读写访问速度。相比于传统磁盘存储的数据库系统，它们无需通过慢速的磁盘 I/O 操作，因此能够实现更低的读写延迟。

高吞吐量： 由于内存数据库可以更快地读取和写入数据，它们通常具有更高的吞吐量。这对于需要处理大量事务或实时数据的应用程序非常重要，如金融交易、实时分析等。

实时处理： 内存数据库适用于需要实时数据处理的场景，因为它们可以迅速响应事件和查询，而无需等待磁盘读写。

复杂查询性能优化：内存数据库能够更好地处理复杂查询，因为数据存储在内存中，而不是分散在多个磁盘块上。这使得在内存中执行连接、聚合和其他复杂查询更为高效。

缓存效果： 内存数据库本身就具有缓存效果，因为数据存储在内存中。这有助于降低对底层数据存储系统的负载，减少对磁盘的频繁访问。

低延迟： 由于数据直接存储在内存中，可以减少与磁盘 I/O 相关的延迟。这对于对延迟敏感的应用程序（如实时分析和响应性应用）非常重要。

容易扩展： 内存数据库通常更容易水平扩展，因为它们的性能主要受限于可用的内存，而不是磁盘 I/O。这使得在需要时能够简单地添加更多的内存节点，从而提高整体系统性能。

临时数据存储： 内存数据库适用于存储临时数据，因为数据在关机或重启后通常会

丢失。这对于缓存、会话存储等应用场景非常有用。

1.2 内存数据库的发展状况

IBM公司于1969年研制了国际上最早的层次数据库管理系统。该层次数据库管理系统提供了两种数据管理方法，一种是采用内存存储的Fast Path和另一种采用磁盘存储的系统。Fast Path支持内存存储数据，该系统是内存数据库的雏形。它将需要频繁访问，要求响应速度高的数据直接存放在物理内存中进行访问和管理，体现了内存数据库的主要设计思想。

D．J．De Witt等人于1984年发表了“内存数据库系统的实现技术”一文，提出了Main Memory Database的概念。专家预言计算机内存昂贵的价格一定会渐渐下降，大容量的数据有可能全部存储在内存中，因此展开了对内存数据库关键技术的研究。

IBM于1985年推出了OBE内存数据库系统，该内存数据库系统运行于IBM 370上。OBE在关系存储和索引上大量使用指针。连接操作使用了嵌套循环算法。

R．B．Hagman于1986年提出了使用检查点技术实现的内存数据库恢复机制。威斯康星大学提出了按区双向锁定模式解决内存数据库中的并发控制问题，并设计出MM-DBMS内存数据库。贝尔实验室推出了DALI内存数据库模型，其特点是使用内存映射体系，采用分区技术把数据库的数据文件映射到共享内存，处理器可以直接通过指针访问存储在内存数据库中的信息，而且数据库的并发控制和日志机制可以根据需要打开或者关闭。

ACM SIGMOD会议于1987年有论文提出了用堆文件作为内存数据库的存储结构。Southern Methodist大学设计出MARS内存数据库模型。这个模型采用了双处理器分别用于数据库和恢复处理。事务提交点之前的任务由数据库处理器负责。恢复处理器负责事务提交。恢复处理器将日志和更新的数据写到磁盘数据库中。周期检查点由恢复处理器负责。MARS采用了双处理器、易失性内存和非易失性内存存储设备将事务处理划分为两个独立的阶段，独立加速各自阶段的处理性能。

普林斯顿大学于1988年设计出TPK内存数据库。TPK提供了一种多处理器架构下多线程的处理模式。它包括输入、执行、输出、检查点4类线程。通常配置为单查询线程和单检查点线程。单查询线程设计不需要并发控制，而输入和输出线程数量可以为多个。它们使用队列结构与其他线程连接。TPK的多线程内存数据库技术实现了一种多阶段查询处理技术。

普林斯顿大学在1990年又设计出System M内存数据库。System M由一系列操作服务线程构成。其中包括消息服务线程、事务服务线程、日志服务线程和检查点服务线程等。System M可以支持并发查询服务线程，但是仍然要控制活动事务服务线程的数量。

美国OSE公司于1994年推出了第一个商业化的、开始世纪应用的内存数据库产品Polyhedra。TimesTen公司于1996年成立并且推出了第一个商业版内存数据库TimesTen 2.0。

Oracle公司于2005年收购TimesTen公司。德国SoftwareAG公司于1998年推出了内存数据库SQL-IMDB。2001年，美国McObject公司推出了内存数据库eXtreamDB。加拿大Empress公司推出了EmpressDB内存数据库。

荷兰CWI研究院于2003年研制了基于列存储模型的内存数据库MonetDB[11][12]。此后又研制了基于向量处理技术的MonetDB/X100系统。2008年MonetDB的商业化版本Vectorwie系统被推出。2010年Ingres公司和CWI研究院合作推出了VectorWise1.0版。2011年3月VectorWise1.5版本获得了TPC-H 100 GB数据量测试第一名，当前仍然是TPC-H性能最好的数据库。

IBM于2008年收购Solid公司的内存数据库SolidDB，该数据库成为IBM公司的一个产品。随后，IBM提出Blink BI商业智能内存查询处理引擎，并为Informix提供内存加速包IWA。

SAP公司于2011年推出SAP HANA[13]高性能分析应用系统。该系统是面向企业分析型应用的内存计算技术产品。

2008年Oracle公司推出软硬件集成设计的Oracle Exadata数据库服务器。Oracle Exadata是由Database Machine与Exadata Storage Server组成的一体机平台。2012年Oracle公司推出Oracle Exadata X3 Database In-Memory Machine，大幅增加了内存配置，实现了将全部数据加载到内存，将所有的数据库I/O全部转移到闪存，以提供高性能数据访问和查询处理。

1.3 研究目标与内容

本文基于svimrdb内存关系数据管理系统库主要阐述了其设计与实现。文章以软件工程的顺序介绍了svimrdb的设计，总共分四大部分。分别为需求分析、概要设计、详细设计和软件测试。在需求分析阶段，文章阐述了svimrdb的开发动机和用户需求。在概要设计阶段，文章阐述了svimrdb内存数据库管理系统的架构设计与大致的模块区分。在详细设计阶段，文章以模块为段落详细阐述了模块内部的设计与实现。最后在软件测试阶段，笔者对svimrdb进行了功能和性能上的测试，写出了测试结果。

此外，文章还研究了基于平衡二叉搜索树的集合算法和基于哈希表的索引算法。在数据库原理方向，文章研究了内存数据库的事务处理与并发控制，并研究了数据库管理系统的事务恢复机制和锁机制。实现了并发控制的锁模块。

2 软件需求分析

本节对软件需求进行分析，记录了使用者对软件产品提出的要求、约束和限制，体现了软件系统的功能，为svimrdb内存关系数据库管理系统的开发打好基础、做好准备。

2.1 功能需求

软件应该提供一组C风格的API。这些API实现一个关系数据库应有的操作。实现对关系表中数据的增加、删除、修改和查找的功能。其中，关系表中支持插入C语言中常见的数据类型。数据库软件的查询功能应该能支持用户自定义功能。关系表格中任意数据均可作为查询对象。查询条件可以任意组合。数据库管理系统应该具备一定的稳定性，发生故障后避免立即崩溃而是给出故障原因。同时该数据库管理系统具备数据备份和恢复机制保障数据信息的准确完整和安全。备份的数据可以存放在内存中，也可以存放在外存中。

对于扩展性方面，软件编写应当遵循一定的标准。软件应当能够适应技术的变化及市场需求的变化，满足新要求对数据库管理系统的功能需要。软件的操作性方面，软件的操作应当尽量简单。功能直观、简洁、友好、使用方便。应当避免复杂的函数调用。软件的代码具有可维护性。硬件方面，数据库管理系统的硬件需求应当降低，在少内存，低CPU主频的条件下能够运行。在大内存，多处理器环境下依然能表现良好。软件应该支持市面上主流的操作系统平台。系统应该具备二次开发能力。能够快速实现业务操作流程。软件应当支持并发控制，采用锁机制保护数据的完整性。

2.2 数据需求

svimrdb应该支持C语言所支持的数据类型。比如char、short、int、long、float、double、零结尾字符串和零结尾宽字符串。其中char为8bit整数、short为16bit整数、int为32bit整数、long为64bit整数、float为32bit浮点数、double为64bit浮点数（支持IEEE 754标准）。零结尾字符串为byte数组。零结尾宽字符串继承自cstdlib的wchar.h接口。数据库管理系统支持随意插入、删除和更改以上数据类型到关系表中。也支持将以上数据类型打印至控制台界面上。数据库管理系统将这些数据长期存储在内存中，也可以将数据临时存储在外存上。

2.3 约束和限制

svimrdb基于StoneValley库，cstdlib库和pthread库。其中pthread库支持POSIX标准。也就是说svimrdb首先可以运行在支持POSIX标准的操作系统平台上。遵循POSIX标准的程序具有一定的可移植性。svimrdb不支超出以上三个库的函数。其次svimrdb给用户提供纯C

的应用程序编程接口，不支持C++调用。svimrdb使用GNU/Linux作为开发平台，vim作为代码编辑工具，并使用clang作为代码编译器。开发时间与开发成本的限制上，svimrdb开发时间约为2个月。时间较短，成本较少。

3 概要设计

3.1 总体设计

svimrdb数据库管理系统由ISO标准C语言编写而成，遵循ISO/IEC 9899：1990标准。软件分为6个模块。它们分别为核心关系代数函数模块、支持并发控制的锁模块、支持关系表格的模块、支持事务处理的模块、支持外部存储的模块和杂项模块。模块化的设计降低了svimrdb的复杂性，并且一定程度上提高了svimrdb的可维护性。用户在增加svimrdb的功能时可以给该数据库管理系统添加功能模块。

svimrdb在查询时将表格转换为视图，核心关系代数模块提供了对视图的查询操作。而对于表格的增删改查则由支持关系表格的模块提供。在对关系表的操作时需要提供事务ID。事务是由支持事务处理的模块和支持并发控制的模块共同控制。支持外部存储的模块是一个独立模块，有独立的头文件接口。svimrdb的5个模块（除了杂项模块）之间大致上是相互隔离的，每个模块提供自己的功能，只有少部分函数和全局变量在模块间被共同调用。5个模块内部的函数通过调用杂项模块内部的函数与杂项模块交互。5大模块提供的函数由svimrdb.h头文件导出并作为接口。数据库管理系统架构如图3-1所示。

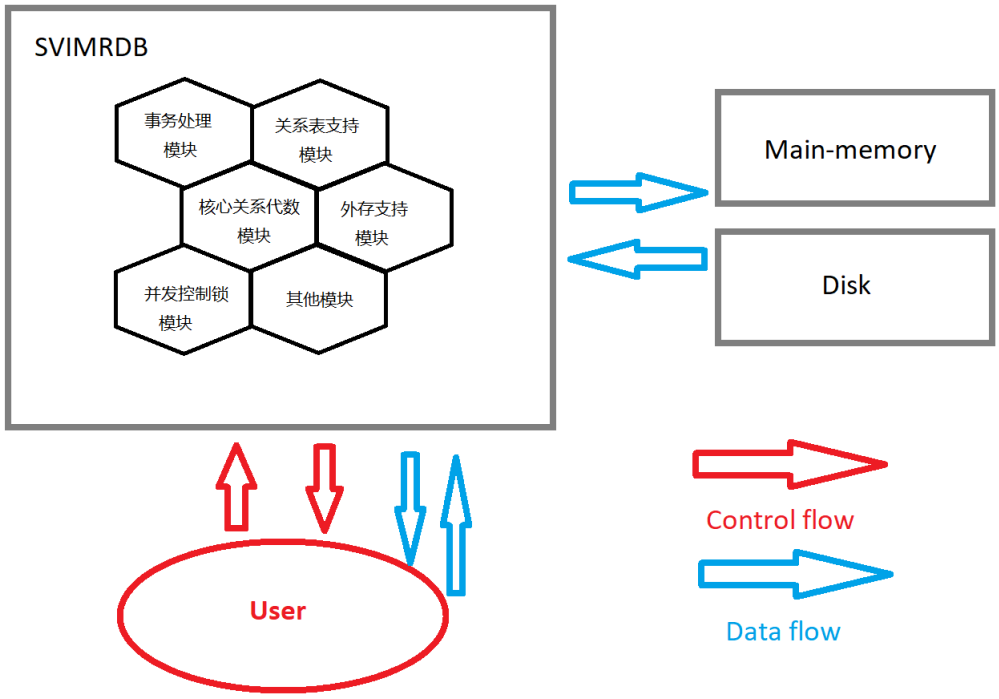

图 3-1 svimrdb 内存关系数据库管理系统架构

svimrdb设计时考虑到整个数据库管理系统应继承C语言简洁的特点，数据库管理系统本身也应当简洁，模块与模块之间不应有太过复杂的关系和函数的相互调用。其次各模块中的函数执行效率应当高，这样才能符合内存数据库高响应、高并发的特性。svimrdb的设计原则包括删除冗余复杂的功能，减少一切不必要的函数调用，为用户提供一款简洁的高效的内存关系数据库管理系统。

使用 svimrdb 数据库管理系统来管理数据有以下优点，包括最小的数据冗余、提高程序和数据的独立性、提供有效的数据访问、增强数据的共享、增强数据的一致性、增强数据的完整性、提高应用程序的开发效率、节约经济成本、平衡需求冲突、增强数据的可访问性和响应性、增强并发性、减少程序的维护量、增强数据的备份和恢复服务和增强数据的质量。数据库的设计应遵循 ANSI-SPARC 三层体系结构，请见图 3-2。

3.2 接口设计

svimrdb各个功能均为标准的C函数，拥有标准C函数的接口。数据库管理系统的用户访问接口以控制台形式为主。打印表格函数使用cstdlib库的printf和wprintf函数。svimrdb数据库管理系统大部分函数导出在svimrdb.h头文件中。因此svimrdb.h是数据库管理系统的主要接口。支持外部存储模块内的函数导出在sixmem.h头文件中。以做到核心功能和附加功能的隔离。

在关系表支持模块中函数siInsertIntoTable被设计成变参函数，它使用了cstdlib的stdarg.h接口。变参函数是得函数调用变得更加容易和方便。无论表格有多少列都可以使用该函数向表格中插入元素。只要将要插入的元素按照表格列的顺序依次排列至函数参数列表中即可实现元组的插入。在函数siCreateSelectView中使用了select回调函数作为参数。回调函数的使用使得svimrdb在做选择运算的时候可以任意组合条件进行查询，进一步增强了数据库管理系统的实用性。

cstdlib 库为 svimrdb 数据库管理系统提供了部分内存操作的 API，例如 memset 函数、memcpy 函数与 memmove 函数。在支持宽字符方面，svimrdb 也使用了 cstdlib 提供的 wchar 接口。在内存与数据结构管理方面，svimrdb 主要使用了 StoneValley 库提供的各种 API。同时，StoneValley 库也为 svimrdb 提供了许多算法，比如集合的交、并和差。

大多数关系数据库管理系统支持 SQL（结构化查询语言）。SQL 为用户提供了一套便捷的数据库使用接口。但是使用 SQL 也有代价，SQL 的分析和优化需要数据库管理系统处理大量信息。目前 svimrdb 还不支持 SQL。取而代之的是 svimrdb 有一套完善的关系代数函数。用户使用这些函数即可对表格关系实现增删改查。

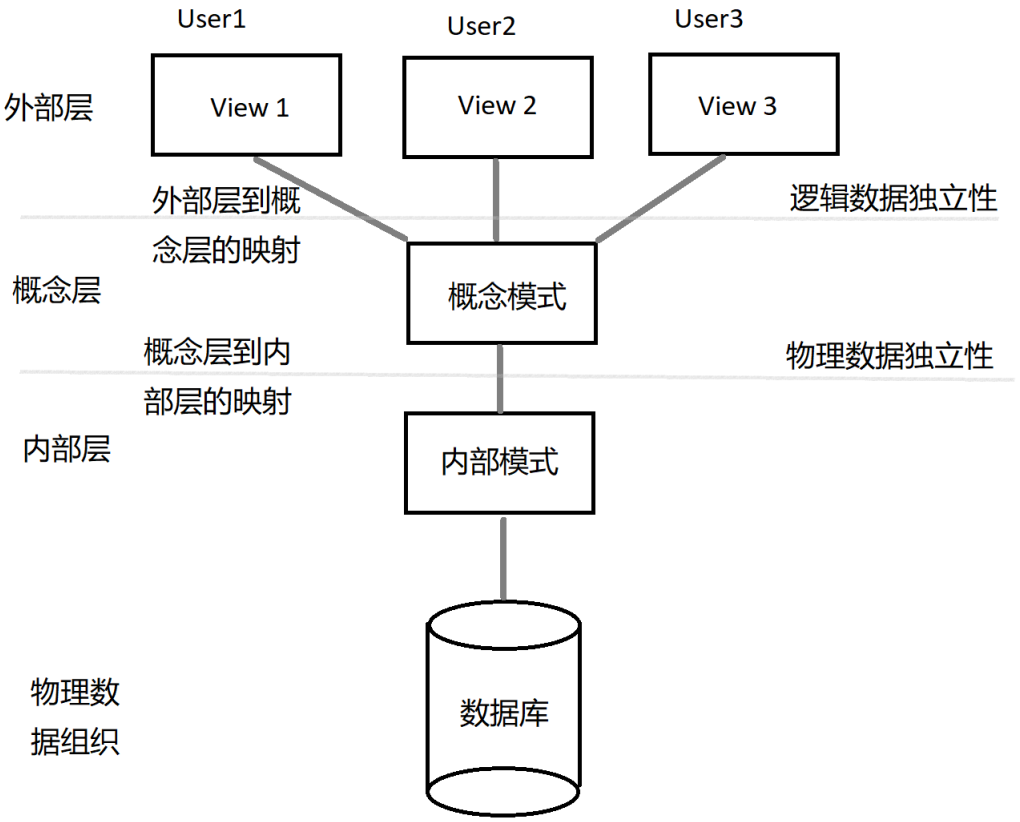

图 3-2 ANSI-SPARC 三层体系结构

3.3 数据结构设计

svimrdb所使用的数据结构和算法主要来自StoneValley库。svimrdb有关关系的数据结构主要有两种。一是视图，视图是表格的抽象。二是表格，表格是关系的具体实现。svimrdb存储关系表使用二维数组组成的矩阵结构。在二维数组矩阵中，每个单元都是指向数据的指针。svimrdb支持数据查询的函数使用平衡二分搜索树来管理和查找数据。平衡二分搜索树具有查询时间短，插入删除效率高等特点。svimrdb所用到的平衡二分搜索树主要有AVL树和AA树。svimrdb的列级约束索引使用散列表。散列表的查询效率非常高。svimrdb使用的散列表主要为链式散列表。对于用作数据库事务恢复的日志列表，svimrdb使用双向链表结构。这样做的好处是数据库管理系统可以从一个恢复点向前、向后查找恢复数据。

4 详细设计

4.1 svimrdb 接口的设计

svimrdb 需要一个接口来定义数据库用到的各种数据结构和导出函数。svimrdb 还需要一些常量来指定数据库可处理的数据类型、约束类型和数据更改的类型。以下将描述接口文件的设计。

svimrdb 内存关系数据库管理系统的接口设计有一个头文件。头文件的内部包含了一些数据库管理系统使用的重要结构体、宏、常量和函数声明。

首先需要引入 svimrdb 所需要的外部库的头文件。接下来引入 StoneValley 库的部分数据结构，分别是线性结构、树结构、集合结构、队列结构和哈希表结构。紧接着是对 cstdlib 库宽字符功能的引用。为了多线程并发执行，本关系数据库管理系统引入 pthread 库。接着是常量和数据结构的定义。

枚举体 CellType 区分数据库表格内部单元的数据类型。其中包含空、字符型、短整数型、整数型、长整数型，单精度浮点型，双精度浮点型，窄字符串型和宽字符串型。下面描述了关系表单元的结构。它是由枚举 CellType 和一个名叫 pdata 的 void *指针构成。

枚举体 en_ColRstct 描述了列约束的类型。这些类型有空类型（表示没有约束），非空约束、唯一值约束和主码约束。主码约束等于非空约束和唯一值约束，因为主码在表格关系中必须为唯一的值，而且不能是空值。

结构体 st_TBLHDR 描述了关系表格的头部。其中 ct 是整列的数据类型。strname 存放了列的名称。cr 表示列级约束。phsh 是指向列索引哈希表的指针。

select 回调函数定义了在用户做选择运算的时候需要输入的条件。select 函数扫描关系表中的每一行。函数返回 TRUE 选中该行，返回 FALSE 不选中当前元组。

st_TABLE 定义了表格结构体。在表格结构体 TABLE 中，tblname 是关系的名称，header 里存放关系中的每一列。tbldata 存放关系的数据。注意 tbldata 的类型是指向 MATRIX 结构体的指针。MATRIX 结构体来自 StoneValley 库，是用来存放矩阵的结构。在 svimrdb 中矩阵对应二维表。

en_LockType 表明了锁的类型。其中有共享锁、排他锁、意向共享锁、意向排他锁、意向共享排他锁和无锁。

st_Lock 是 svimrdb 中锁类型的结构。pobj 表示了上锁的对象。lt 表示了上锁的类型。

en_AltType 是数据日志中数据修改的类型。其中有表格单元被修改、增加元组、删除元组、增加列、删除列、增加关系表和删除关系几种中操作。

st_DatAlt 是存放数据日志的结构体。at 表示数据修改的类型。ptbl 指向被修改的表格。在 DATA 联合体中，dacell 结构体用来描述表格单元被修改的内容。ln 表示行，col 表示列。before 指向修改前的数据。datpl 结构体用来描述被修改的元组，sizln 描述被修改的行号，tupledat 存放行内的数据。dacol 描述被修改的列。sizcol 是列的序号。ddr 描述列头，coldat 描述列内的数据。

st_TRANS 是描述事务的结构体。qoprlst 是一个双向队列，该双向队列用于存放相对于事务的日志。setlock 是一个记录在事务上上锁的集合。

st_TBLREF 是表格参照的结构体。pold 指向原来的表格。pnew 指向修改后的新表格。

接下来是函数声明。这些函数的定义分别存放在 svimrdb 工程内其他的模块中。用户可以以 svimrdb.h 文件为接口调用这些函数。

BKSNUM 宏常量是哈希表的大小。

4.2 核心关系查询模块的设计与实现

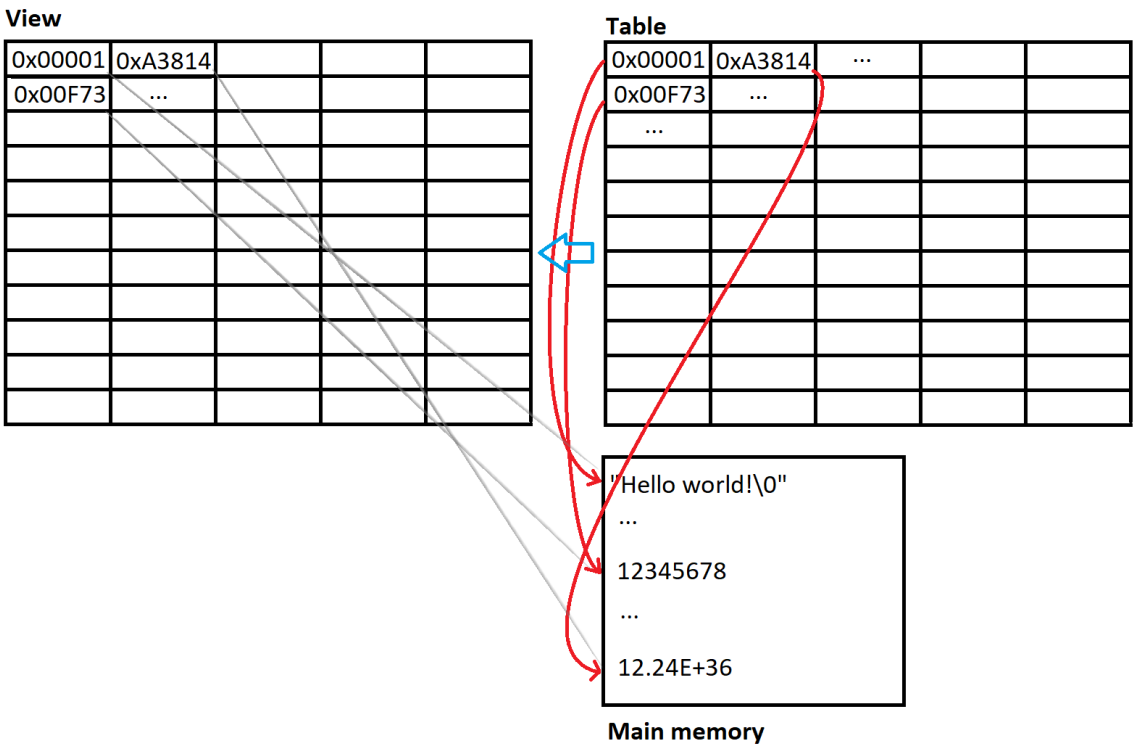

图 4-1 svimrdb 中视图与表格的区别

核心关系查询模块内部实现了基本的关系代数函数。这些关系代数函数用于查询并修改表格中的数据。这些关系代数函数有 siCreateUniqueView、siCreateUnionView、siCreateIntersectionView 、 siCreateDifferenceView 、 siCreateCartesianProductView 、

siCreateSelectView、siCreateProjectView、siCreateCell、siDeleteCell 和 siAlterCell。这里要提及一个概念：在 svimrdb 中视图指的是没有表格头部描述的关系二维表矩阵。视图与表格的区别在于表格拥有头部说明，该头部说明包含了表格中每一列的描述。而视图更像是一个表格的副本。它拥有表格内部所有的数据指针。视图与表格的区别如图 4-1 所示。

进一步来看核心关系查询模块的设计：如果把一张关系表看作元组组成的集合，siCreateUniqueView 用来去除重复元组，使关系表满足集合的定义：没有重复的元素。同时，siCreateUniqueView 可以被并集、交集和差集函数调用用来生成相应的集合。表格之间笛卡尔积的创建由 siCreateCartesianProductView 函数负责。灵活应用并交叉和笛卡尔积函数和选择、投影函数能够查询关系表的内容。选择函数中其中一个参数为 select 回调函数函数。在调用选择函数时需要编写 select 回调函数。投影函数用来选择关系表中相应的列。该函数是由一个数组参数确定要选择的列。剩余三个函数为表格单元的创建、删除和值的修改。

siCreateUniqueView 函数用来对视图创建一个以元组为单位，每个元组都是唯一的副本。在该函数内部，首先创建了一个集合 set。这个集合被用来排斥表格中重复的元素。其次一个 for 循环被使用，用来逐行将表格的元组添加进集合。接着，一个新的视图被创建。该视图的行数等于集合的元素个数。然后集合中的每一个元素被逐个遍历加入进新创建的视图。最后新创建的视图被返回。至此，原视图中重复的行被过滤掉了。如图 4-2 所示。

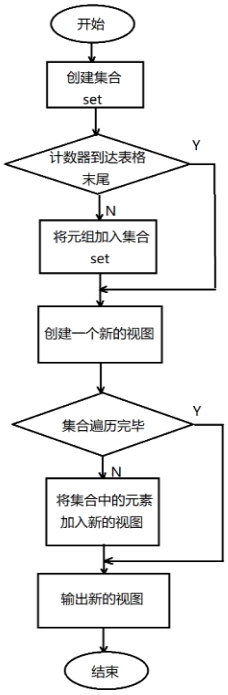

图 4-2 siCreateUniqueView 函数流程图

siCreateUnionView 函数用来创建两个视图的并集。它的原理是这样的：首先创建 a 视图的唯一确定集合并标记为 pma。其次创建 b 视图的唯一确定集合，标记为 pmb。然后判断 pma 和 pmb 中是否有空集的存在。如果 pma 和 pmb 都是空集，则返回空集。空集并空集等于空集。如果 pmb 是空集，则返回 pma。如果 pma 是空集则返回 pmb。当 pma 的行数不等于 pmb 的行数时，说明关系集合 pma 不等于关系集合 pmb，此时直接返回 pma 和 pmb 的并集。当 pma 的行数等于 pmb 的行数时，说明 pma 可能等于 pmb。此时，新建一集合，用来消除 pma 和 pmb 内部重复出现的项目。这时逐行扫描 pma，将 pma 中所有的行添加进集合。再逐行扫描 pmb，将 pmb 中所有元组添加进同一集合。最后遍历该集合中每一个元素，将元素列进返回的视图，该视图即为 a 和 b 的并集。如图 4-3 所示。

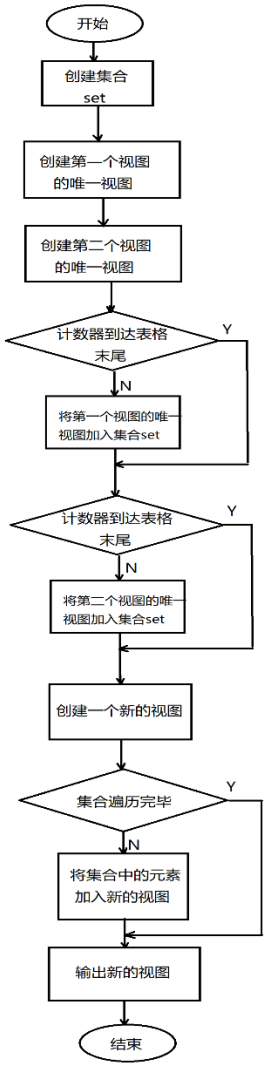

图 4-3 siCreateUnionView 函数流程图

siCreateIntersectionView 用来创建视图 a 和视图 b 的交集。在该函数中，首先判断 a 集合与 b 集合是否为空集。空集和空集的交集为空集。其次创建视图 a 和视图 b 以元组为单位的唯一确定集合并分别标记为 pma 和 pmb。创建两个集合变量分别为 seta 和 setb。将 pma 中的元组插入 seta。将 pmb 中的元素插入 setb。使用 StoneValley 库的内部函数 setCreateIntersectionT 创建 seta 与 setb 的交集 psetc。将 psetc 中的元组逐个插入返回的视图中。交集创建完毕。如图 4-4 所示。

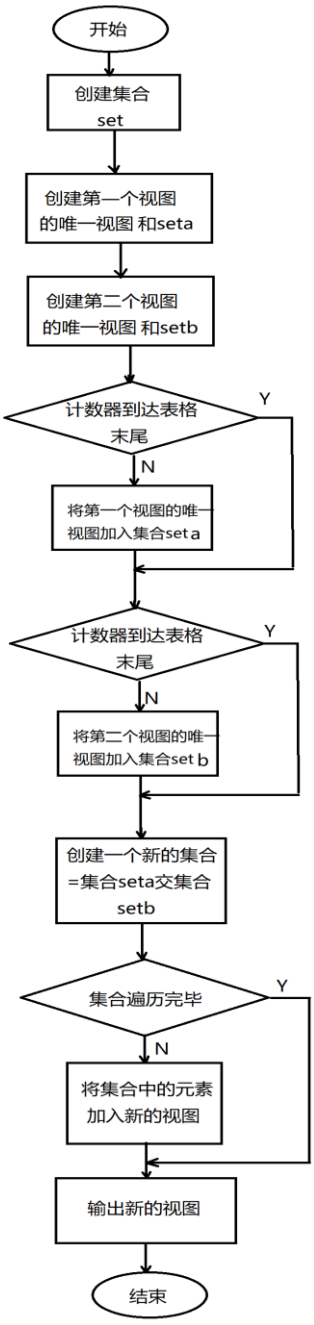

图 4-4 siCreateIntersectionView 函数流程图

siCreateDifferenceView 函数用来创建两个视图的差集。该函数是这样工作的：首先创建 a 视图和 b 视图的唯一确定集合分别标记为 pma 和 pmb。其次判断 a 视图是否为空，因为空集减去任何集合都等于空集。如果视图 a 不为空，而视图 b 为空的话，a 减去空集等于 a 本身。此时，只要返回集合 a 的副本即可。当 a 和 b 都不为空时，操作原理与交集大致相当。使用 StoneValley 内置函数 setCreateDifferenceT 来创建 a 与 b 的差集。最后将差集内的元组返回为视图即可。如图 4-5 所示。

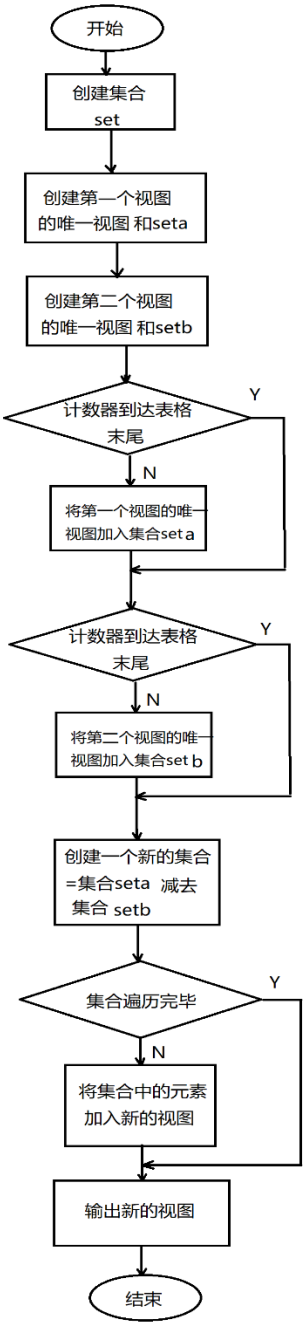

图 4-5 siCreateDifferenceView 函数流程图

siCreateCatesianProductView 用来创建视图 a 和视图 b 的广义笛卡尔积。在视图 a 和视图 b 的广义笛卡尔积 r 中，r 的行数等于 a 的行数乘以 b 的行数。r 的列数等于 a 的列数加上 b 的列数。笛卡尔积的创建使用了一个 for 循环。在 for 循环的内部是两个 memcpy 函数，因此笛卡尔积创建的效率能够保证。如图 4-6 所示。

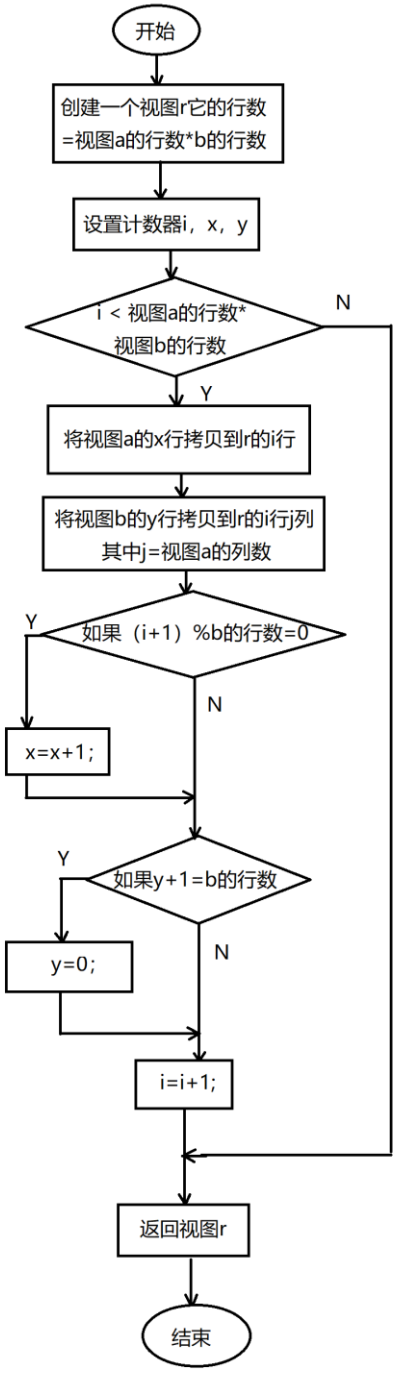

图 4-6 siCreateCatesianProductView 函数流程图

siCreateSelectView 函数从一个视图当中选出相应的元组。siCreateSelectView 函数是这样工作的。首先函数创建一个空的视图标记为 r。接着函数使用一个 for 循环遍历输入视图的元组。在遍历每个元组的同时调用 select 回调函数 cbfsel。如果 cbfsel 函数的返回值为 TRUE，就把遍历到的元组添加进 r 内。每次添加元组后，调整 r 的大小，使 r 适应新添加的元组。最后返回 r 的同时通过函数参数 pparr 指针返回哪一行被添加进了 r。如图 4-7 所示。

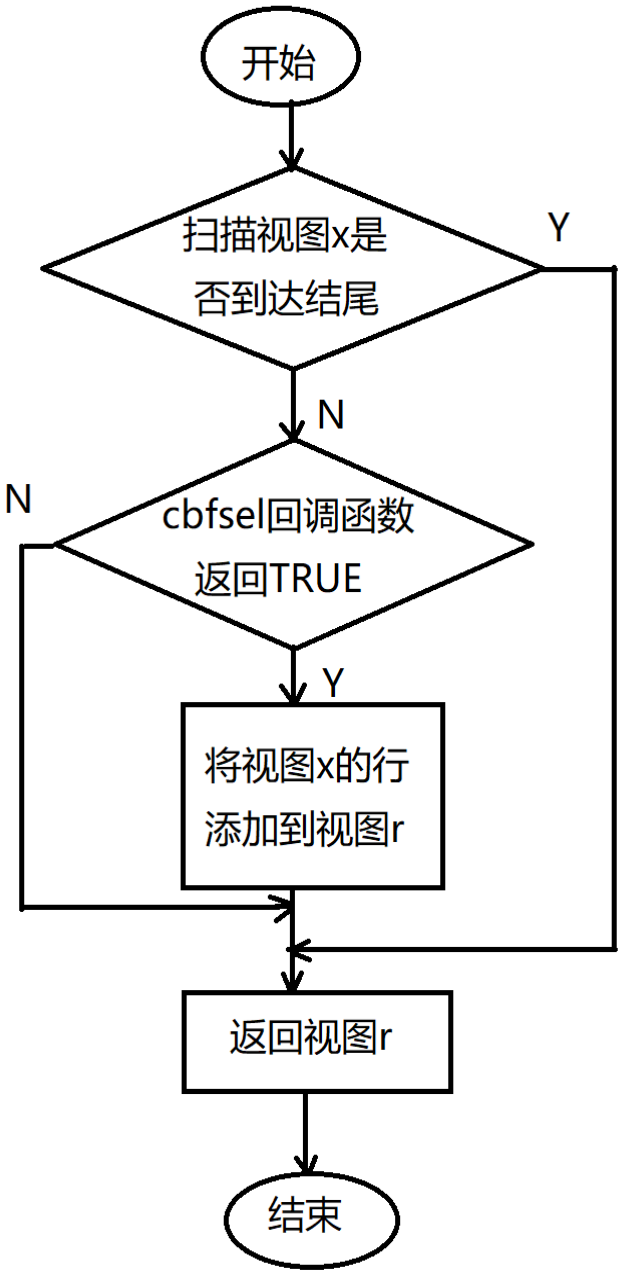

图 4-7 siCreateSelectView 函数流程图

siCreateProjectView 函数返回一个视图的投影。该函数就是在一个视图中选择相应的列。首先创建一个视图 r。然后根据函数参数 parrz 的内容，选择相应的列。将选择到的列复制进 r 中。最后返回 r。如图 4-8 所示。

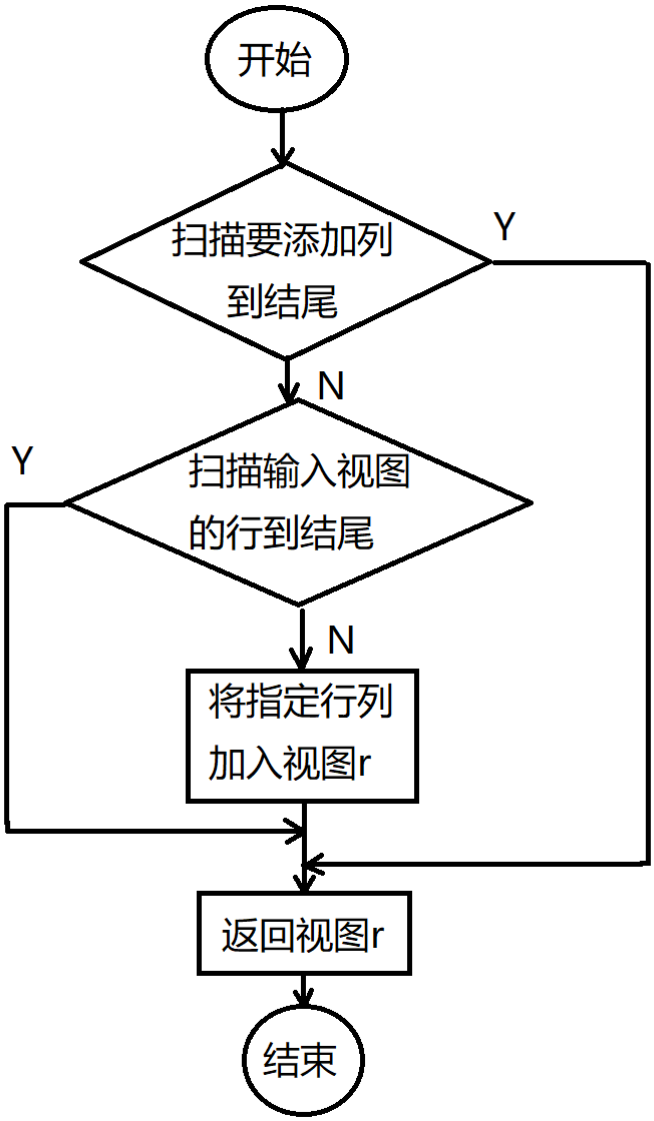

图 4-8 siCreateProjectView 函数流程图

siCreateCell 函数用于在表格单元中新建对应的 Cell 结构体。函数有两个参数，第一个参数 pitem 指向要添加的内容。第二个参数 ct 指明要添加数据的类型。siCreateCell 函数会返回一个指向 CELL 结构体的指针。如果该指针为空，则表明当前创建的单元为空单元。

siDeleteCell 函数释放 siCreateCell 函数创建的单元的内存，并把当前单元置为空。

siAlterCell 函数可以重新申请一个 CELL 单元的 pdata 指针指向的内存。

基本的关系运算只包含并、差、笛卡尔积、选择和投影。其他三种运算，例如交、

连接、和除运算都可以通过前述 5 中基本关系运算来表达。用户可以自由组合以上函数的调用来实现各种关系运算。

svimrdb 中使用到的集合结构来自于 StoneValley 库。StoneValley 库中的集合可以由平衡二分搜索树来实现。其中 AVL 树和 AA 树都可以用来实现集合。AA 树是一种红黑树的变种[19]。红黑树的插入删除效率较高。平衡二分搜索树的查询效率是 O(log N)。使用平衡二分搜索树可以保证 svimrdb 的执行效率和空间效率。

4.3 关系表支持模块的设计与实现

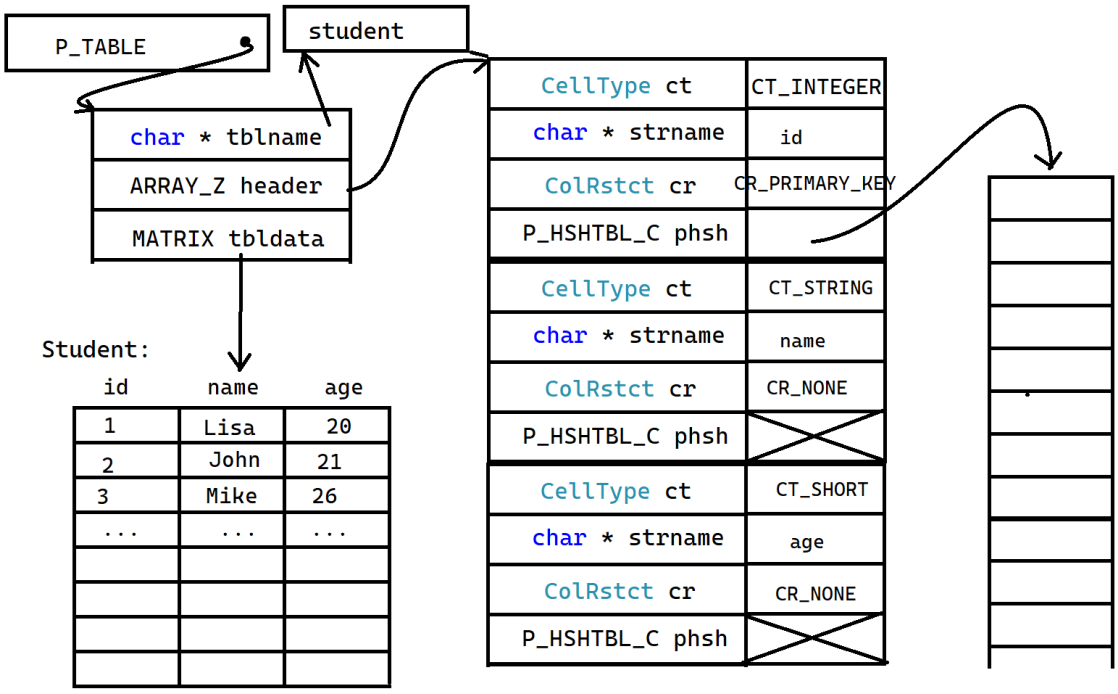

图 4-9 关系表的内存结构图

关系表支持模块中实现了对关系表增删改查的 API 和一些增强的对视图进行操作的 API。

关系表支持模块的设计如下：在本模块中有一些函数支持视图的操作。比如排序视图、实例化视图、删除实例化视图、打印视图函数。在对关系表的实际操作过程中，需要将关系表转换成相应的视图矩阵才能进行查询操作。siGetColumnByString 函数用来通过列名称字符串返回列号。该函数配合核心关系查询模块中的 siCreateProjectView 函数使用可以方便地选择表格中的列。其余的函数是对表格进行操作的函数。siCreateViewOfTable 函数提取表格中的数据并形成一个视图。提取表格视图完毕后即可对表格进行查询。对表格的创建、复制和删除功能紧接其后。插入元组，删除元组和更

新表格单元的函数实现了行和单元的增加、删除和修改功能。其次是对表格中列的操作：这些操作有增加列和移除列。

siSortView 函数用于对视图按列进行排序。在 siSortView 的三个参数中，pmtx 指向一个矩阵形成的视图。col 指向视图中的某一列。当 ascd 为 TRUE 的时候，siSortView 按从小到大排序，反之则按降序排序。因为排序函数中使用了全局变量，在多线程调用时全局变量会遭到影响。所以在 siSortView 函数中排序前要使用 pthread 库提供的互斥体保护全局变量。对视图的排序实质上由 cstdlib 的 qsort 函数实现。

siInstantiateView 用来对每个视图单元中的 P_CELL 指针实例化。该函数的作用是在视图上重新创建一个实例副本。该函数用于关系表的复制操作。

siDestroyView 函数用于将 siInstantiateView 函数创建出来的副本加以销毁。释放内存。

siPrintView 函数用于将视图打印到控制台上。

siGetColumnByString 函数使用参数接受一个关系表和一个字符串。该函数在关系表的表头描述结构体中查询和输入字符串相等的列并返回列的编号。编号从 0 开始。如果找不到相应的列则返回-1.该函数对大小写不敏感。

siCreateViewOfTable 函数用于以当前表格为副本创建一个新的视图。siCreateViewOfTable 函数直接使用了 StoneValley 库内的 strCopyMatrix 函数。

siCreateTable 函数用来创建关系表。它的第一个参数是指向一个事务的指针，表示在当前事务下创建关系表。如果该参数为 NULL 则表示无条件创建关系表。第二个参数是关系表的名称。第三个参数是关系表的表头描述信息，也就是关系表的列信息等等。在 siCreateTable 函数中首先在内存中申请一个关系表结构体 TABLE。接着函数初始化了关系表的表头描述结构。紧接着如果输入的表头描述结构体中含有列索引的话，该函数复制了列索引到新创建的关系表中。其次，函数初始化了关系表矩阵，为添加数据做好准备。最后，如果关系表创建成功则将关系表创建的动作写入相应事务的日志动作列表中。返回创建的关系表的指针。至此关系表创建完成。

siCopyTable 用来复制关系表。该函数实际上调用了 siCreateTable。相当于在原来关系表的基础上创建了一个一模一样的新表。在重新申请新表的二维矩阵大小后，使用两个嵌套的 for 循环将原表中的数据全部复制到新表当中去。

siDeleteTable 实现了删除关系表的功能。与 siCreateTable 类似，该函数需要一个指向事务的指针作为参数。第二个参数是指向指定关系表的指针。按照 siDeleteTable 的实现，第一步首先在相应的事务日志动作列表中记录关系被删除的日志。注意此时要在日志中保留被删前的关系表以供恢复。接着函数释放了指向关系表名称字符串的指针。第三步，逐个扫描关系表列描述结构体，删除列级索引。第四步，删除表头描述结构体数组。第五步，释放关系表二维矩阵中各个单元所占用的内存。

siInsertIntoTable 函数的作用是将元组插入关系表。该函数是一个变参函数。在函数内部，首先，通过调用 strResizeMatrix 函数改变表格数据存储矩阵的大小，使得矩阵的行数等于原行数加一的值。其次，通过一个 for 循环遍历关系表头描述数组将数据按照列的排布逐个添加进表格。在添加之前，首先定位到表格中的与单元。然后在该单元上申请一块新的内存。这时候判断表格的列级约束，查看要添加的项目是否为唯一值或者主键。通过查找哈希表索引确定元素是否已经出现在表格中。如果当前列拥有主码约束，而且已经有相同值存在在关系中则返回插入失败。在插入确定完成后记录表格修改后的值，即为新插入的一行到事务的日志动作列表 qoprlst 中。如果插入失败则调整表格行数为表格行数减去一后的值。

siDeleteFromTable 函数用于移除表格中的某一行。首先要记录移除的那一行到事务的日志动作列表中去。该过程分为几个小步骤。第一，记录表格修改属性为删除元组 AT_DEL_TUPLE。第二，申请一个元组长度大小的数组。第三，将要删除的行内的单元逐个添加到数组当中。第四，将数组存入日志。第五，将日志写入事务日志动作列表 qoprlst。然后逐列判断该列是否存在列级约束。如果列级约束存在就从索引哈希表中删除相应数据。然后调用 siDeleteCell 函数释放单元内存。接着将关系矩阵中剩下的行统统向上移动一行。最后调整关系矩阵大小，将关系矩阵的行数设置为原行数减一的值。

siUpdateTableCell 函数用于更新关系表单元中的数据。首先该函数通过函数参数定位到关系表的特定行列。其次定位到单元所在的列获取表格列描述信息。如果要更改的值为 NULL 而列级约束为非空，则返回 FALSE 表示更改失败。如果列级约束为唯一或者主键，则查询哈希表索引找到相应的元素查看更改后的值是否符合要求。如果不符合要求则返回 FALSE，更改失败。如果符合要求，则操作哈希表索引，删除原记录，添加新记录。如果上述更改成功则将 AT_ALTER_CELL 属性添加进日志。同时，将更改的单元行列信息，更改前值，更改后值一并写入日志。最后定位到表格中相应行列将更改后的值写入表格。

siAddTableColumn 函数为关系表添加新的列。第一步，将新列的名称转换为大写字母，以避免大小写字母冲突。其次，逐个搜索表头描述信息，如果有名称相等的列存在则返回 FALSE。然后将新列的表头描述信息添加进表格的表头描述信息中。这时候要复制表的名称和新建列级约束使用的哈希表索引。调整表格的大小，为表格增加一列。再将表格新增加的一列的全部元素置为空。将 AT_ADD_COLUMN 添加表列信息写入日志动作列表。添加完成。

siDropTableColumn 函数删除关系表内特定的一列。首先判断要删除的列号是否在范围内。如果在范围内的话则记录日志信息。这个过程分为以下几步：首先设置日志信息为 AT_DEL_COLUMN 删除表列。记录被删除列的关系表指针。记录被删除的列号。为删除的记录创建一个数组。复制被删除列的名称。将被删除列中所有的元素添加进数组

中。最后提交日志到日志动作列表。如果被删除列的哈希表索引不为空则删除整条哈希表。释放被删除列的名称字符串指针。删除被删除列的表头描述信息。使用一个 for 循环逐行扫描被删除列将余下的列合并到前一列。调用 strResizeMatrix 函数修改表格大小。如果列删除成功则返回 TRUE，否则返回 FALSE。

关系表是关系内存数据库中的重要数据结构。对关系表的操作较为繁琐。因为使用了列级约束哈希表索引,所以添加删除元素的速度能过够保证。一般来说哈希表的添加、删除和查找的时间复杂度约为 O(1)。关系表的内存结构图在图 4-9 中。

4.4 事务支持模块的设计与实现

内存关系数据库管理系统的事务管理日志恢复模块包含 4 个函数：siBeginTransaction、siCommitTransaction、siRollbackTransaction 和 siReleaseAllTransaction。psetTrans 是一个集合类型的公有变量。它用来存储事务结构体。该公有变量需要 pthread 库中互斥体的保护才能进行合法的多线程访问。

事务支持模块的设计如下：首先通过 siBeginTransaction 取得一个新事务的指针。其次，对表格的修改需要输入该指针作为参数，以确定到哪个事务修改了哪张表格。在表格的修改过程中，所有修改的信息都被以日志的方式记录在以事务为单元的修改动作列表中。该列表是一个双端队列。siCommitTransaction 用于确认事务的修改。siRollbackTransaction 用于回滚所有的修改直至 siBeginTransaction 运行之前。如果数据库程序运行结束，需要调用 siReleaseAllTransaction 释放所有的事务所占的内存。

siBeginTransaction 函数用来创建一个新的事务。在此函数内部，首先新建了一个新的 TRANS 结构体，并保存该结构体的指针。然后，初始化该结构体中的日志动作列表双向队列。再初始化锁集合。其次，为全局变量 psetTrans 指针分配内存，并将新建的 TRANS 结构体指针插入到集合内部。最后返回 TRANS 结构体指针。

siCommitTransaction 函数用来确认事务的提交。此函数遍历输入的事务指针内部的日志动作列表双向队列，每次从队列头部出队一个元素。取出该元素后，对元素的操作类型进行判断。如果当前操作为 AT_ALTER_CELL 改变关系表单元，释放日志动作队列中存储的先前单元数据。如果当前操作为 AT_DEL_TUPLE 删除元组，则通过循环逐个释放先前存放的元组元素数据。如果当前操作为 AT_DEL_COLUMN 删除列，通过循环逐个释放先前存放的列单元数据。释放列名字符串指针。释放数据存放数组。如果当前操作为 AT_DEL_TABLE 删除表格，就删除先前保存的表格副本。从 psetTrans 集合中移除 P_TRANS 指针。释放对应该事务的所有锁对象。最后，释放 P_TRANS 申请的内存。

siRollbackTransaction 函数回滚一个事务内部日志动作列表内记录的所有动作。此函数遍历日志动作列表，每次出队一个元素。判断出队元素的操作类型。如果操作是

AT_ALTER_CELL，调用 siUpdateTabelCell 函数，将表格单元先前的值覆盖表格单元修改后的值。以此达到回滚目的。如果操作是 AT_ADD_TUPLE，调用 siDeleteFromTable 删除相应的元组。如果操作是 AT_DEL_TUPLE，首先重新调整表格大小，将表格行数增加一行。其次，将删除掉的元组内元素逐个添加到表格内部。如果当前添加的列有列级约束，则判断约束类型。在唯一值约束和主键约束下为哈希表索引添加相应的数据。再把记录元组单元数据的数组依照元素逐个释放删除。如果操作是 AT_ADD_COLUMN，使用 siDropTableColumn 删除已经添加的列。如果操作是 AT_DEL_COLUMN，首先在表格头描述信息中添加列名称等信息。其次判断被删除列的约束类型。如果是唯一值和主键约束，就为列创建一个散列表索引。再修改表格将删除的列添加回表格。这时要判断每一列元素的约束。将相应的数据添加到哈希表索引中。然后删除删除列时保存的数据，释放表列名称字符串指针。如果操作是 AT_ADD_TABLE，调用 siDeleteTable 函数删除被添加的表格。如果操作是 AT_DEL_TABLE，恢复被删除的表格。注意，此时因为恢复后的表格地址与恢复前的表格地址不一样，此时要通过函数的 pparr 参数返回恢复之前的地址和恢复之后的地址。最后，做一些收尾工作。将事务指针从事务指针集合中删除，再将事务对应的锁集合删除。释放事务指针。

siReleaseAllTransaction 函数从事务集合中释放所有的事务。使用 setDeleteT 函数释放集合所占用的内存，再将 psetTrans 指针置为 NULL。注意，该函数只能在程序结束运行之前被调用。释放事务所占用的内存应该使用 siCommitTransaction 和 siRollbackTransaction 函数。

4.5 并发控制锁模块的设计与实现

并发控制锁模块的设计是通过 siTrylock 加锁一个对象并且阻塞等待，直到 siUnlock 函数释放掉该锁。原理如图 4-10

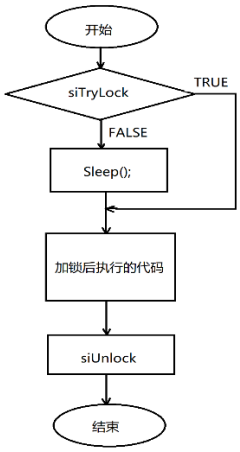

图 4-10 加锁与解锁流程图

管理并发控制的锁模块拥有两个导出函数，分别是 siTrylock 和 siUnlock。以下是 silock.c 模块的原理。

siTrylock 函数在特定的事务中为 pobj 对象加锁。可加锁的类型在 LockType 枚举体中描述。该函数首先通过_sicbftvsTrans 回调函数遍历 psetTrans 事务集合。在_sicbftvsTrans 函数内部首先取得被遍历事务结构体的指针。其次，在被遍历结构体内再通过_sicbftvsFindLock 回调函数遍历锁集合。在_sicbftvsFindLock 回调函数内部通过锁的相容矩阵 bCompatibleMatrix 判断锁是否相容。如果不相容返回 CBF_TERMINATE。如果相容，返回 CBF_CONTINUE。回到 siTrylock 函数中，如果锁相容，则在相应事务上加锁。将锁写入事务对应的锁集合中。

siUnlock 函数简单地将锁从事务锁集合中删除。

4.6 杂项模块的设计与实现

杂项模块收集了不适宜放在其他模块中的函数。

杂项模块主要包含哈希回调函数。用来散列特定的值。

siStrLCase 函数用来将字符串转换为小写形式。

siHashChar、siHashShort、siHashInt、siHashLong，siHashFloat，siHashDouble，siHashString 和 siHashWString 函数分别用来散列各个数据的值。

siPlatformSize 返回当前处理器平台数据寄存器长度大小（以字节 Byte 为单位）。此函数在 32 位系统上返回 4，在 64 位系统上返回 8.

hshSearchCPlusA 和 hshSearchCPlusW 函数分别在链式哈希表中搜索窄字符串和宽字符串。这两个函数的应用使得 StoneValley 库中的链式哈希表能够支持字符串。

4.7 支持外部存储模块的实现

sixmem（SVIMRDB eXternal Memory）是 svimrdb 的外部存储器支持组件。拥有了该组件，svimrdb 就可以将数据表存放至磁盘存储器等外存上。

支持外部存储的模块设计为简单地将表格从内存中转移到磁盘上保存。在使用写入和读取函数的时候，首先打开一个文件指针。其次将表格作为参数把表格的内容按照一定格式全部写入到文件的指定位置上。读取函数从文件的特定位置读取表格内容，并转换到内存。因为写入和读取函数都可以从文件特定位置写入读取内容，所以一个文件中可以存放多个表格。

该模块的接口文件内包含外部存储使用到的结构体和函数声明。其中 XCELL 是存储表格单元所使用的结构体。ct 表示数据类型。fpos 表示外部存储器的位置。svimrdb 将表格存储在 db 文件中，db 文件结构如图 4-11 所示。

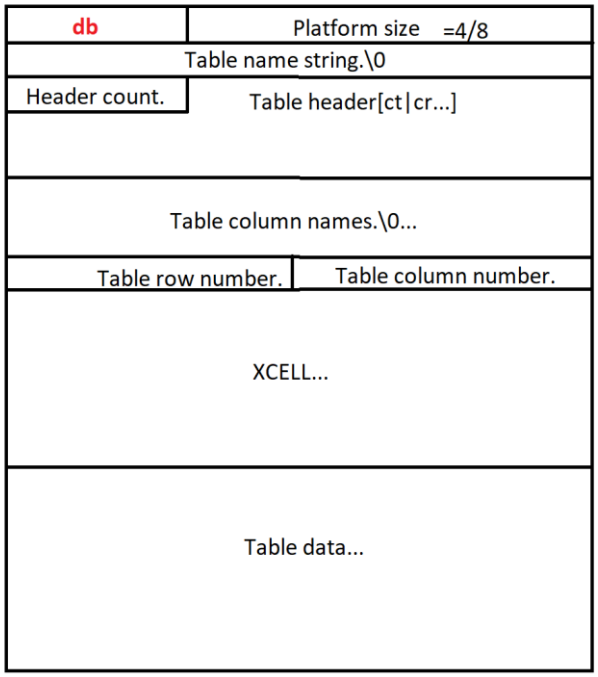

图 4-11 db 文件结构图

外存支持模块中包含两个函数。分别是 siSaveTable 和 siLoadTable。siSaveTable 的第一个参数是 fp 文件指针，它指向一个合法打开的文件。第二个参数是 lpos，它指向打开文件中的某一个位置。第三个参数是 ptable，它指向内存中一张表格的指针。首先 siSaveTable 定位到 lpos 所指的文件位置，然后写入“db”文件魔数，用来标识文件类型。接着函数写入平台整数长度。之后函数写入表格名称字符串。函数又写入了表头描述结构体数量。再者，函数写入表头描述结构体中的 ct 数据类型和 cr 列级约束。然后函数写入表格的名称字符串。表格名称字符串可以有多个，数量与表头描述结构体数量相同。然后，函数写入行数与列数。接着，扫描全表将 XCELL 写入文件。最后将表格数据按照类型全部放在文件末尾。至此，表格已经被存储到外存上了。

在函数 siLoadTable 中，函数接受两个参数：fp 是指向一个合法打开的文件的指针。Lops 指向从何处加载表格。函数返回一个指向表格的指针。在本函数中，第一步将文件指针移动到 lpos 指到的位置。第二步，读取“db”魔数。第三步，读取平台整数长度。第四步，判断魔术是否合法并且判断平台整数长度是否匹配。如果匹配，执行第五步：在内存中创建一个表格。第六步，读取表格名称字符串。第七步读取表头描述结构体数量。第八步根据这个数量读取表格描述结构体。第九步根据之前的数量读取表列名称字符串。第十步，读取表格行数与列数。。第十一步，在内存中根据表格行列申请相应大小的内存。第十二步根据 XCELL 结构体读取表格内容，并填充到表格单元里。注意这里要判断某一列上是否含有列级约束。如果有列级约束，应该建立哈希表索引，并将数据添加到索

引中。最后返回表格指针。

5 软件测试

本节中以示例文件的编译与执行为例，测试了数据库管理系统的部分功能。此次测试是对 svimrdb 进行的黑盒测试。关于对数据库管理系统内部函数进行的白盒测试，白盒测试在在开发过程中进行。白盒测试主要测试函数执行结果的正确性。测试过程为先找到函数参数的定义域和边界值，将特定的值输入函数并执行以覆盖逻辑控制条件。查看输出和值域进行对比，查找有无超出范围的可能。每个开发的迭代过程均安排了相应的白盒测试。

在第一个示例文件中（exp_2024-01-30_1.c）建立了三张表格，并通过查询将表格打印出来。在 main 函数中，申请了两个线程，分别为 pt1 和 pt2。使用 pthread_create 函数创建线程。线程服务函数分别为 foo 和 bar。在函数 foo 中，开始一个事务。在该事务底下创建三张表格。注意写表格内容的时候根据三级封锁协议，要给表格增加共享锁和写锁直到事务结束才能释放。事务使用 siCommitTransaction 函数提交成功，并释放加在三张表格上所有的锁。在 bar 函数中，开启另一个事务。在该事务中依次查询三张表格。将查询结果打印到控制台上。在查询读取表格之前，应该给表格加读锁。main 函数中首先使用 pthread_join 函数启动 foo 函数。然后启动 bar 函数。foo 函数启动后，三张表分别加读锁与写锁。bar 函数启动后根据三级封锁协议与锁相容关系，一个事务给一个对象加了写锁以后，其他事务不能再给该对象加读锁。因此程序在 SLEEP 7 处陷入等待。等 foo 函数完成后，释放所有的锁，bar 函数进入执行。最后 main 函数释放了所有的锁对象，执行完毕，结果正如预期。如图 5-1 所示。

```
SLEEP 7
select * from student:
sno     sname   ssex    sage    sdept
2023001 Lisa    F       20      CS
2023002 John    M       19      CS
2023003 Alice   F       18      MA
2023004 Jack    M       19      IS

select * from course:
cno     cname   cpno    ccredit
1       Database        5       4
2       Math    0       2
3       Informatics     1       4
4       Operating System        6       3
5       Data Structure  7       4
6       Data Processing 0       2
7       PASCAL  6       4

select * from sc:
sno     cno     grade
2023001 1       92.000000
2023001 2       85.000000
2023001 3       88.000000
2023002 2       90.000000
2023002 3       80.000000

select sdept from student:
CS
IS
MA
Thread 1 returns: 0
Thread 2 returns: 0
```

图 5-1 示例程序执行结果

第二个示例文件（exp_2023-07-08_1.c）对数据库管理系统的表格支持功能进行了测试。文件中，通过函数 siCreateTable 创建了一个名为 student 的表格。其次使用 siInertIntoTable 函数给 student 表格插入 5 个元组。在插入元组时，多插入了一个主键重复的元组，以测试 svimrdb 是否能查询到主键的重复并拒绝插入。然后，该测试使用了 siDeleteFromTable 函数删除了行号为 0 的元组。接着该测试文件依次测试了添加列、删除列和更新表格单元的功能。测试分别调用了 siAddTableColumn、siDropTableColumn 和 siUpdateTableCell 函数以测试其执行结果是否正确。紧接着，测试文件通过调用 siCreateViewOfTable 为 student 表创建了一个视图。再使用 siPrintView 函数将视图打印到控制台上显示结果。接下来测试文件使用 siDeleteView 删除了视图，使用 siDeleteTable 删除了表格。最后测试文件使用 siRollbackTransaction 函数将事务回滚到事务建立之初。测试完毕，所有函数均正常执行，结果正常。

在对内存读写安全的方面进行测试时笔者使用了 valgrind 工具。valgrind 是 GNU/Linux 下测试内存泄漏的工具。笔者对以上两次测试生成的可执行文件使用了 valgrind 测试。测试结果显示 svimrdb 的函数在执行期间均为读错、写错内存，在执行完毕后堆内存均已安全释放。

数据吞吐量测试的进行如下（如图 5-2 所示）：首先建立一个简单的表格，该表格仅包含一个拥有主码列级约束的列。该列的数据类型为 int 类型。然后使用循环对该列插入 10000 个数据并测试插入 10000 个数据所需要的时间。在编译完成后，使用 GNU/Linux 上的 time 命令对编译生成的可执行文件进行测试。测试结果表明，10000 个数据在 0.15 秒内全部添加完成。

```
root@DESKTOP        :/mnt/c/Users/user1/Desktop/svimrdb/src# cat 1000.c
#include "svimrdb.h"

int main()
{
        size_t i;
        P_TRANS ptrans;
        P_TABLE ptbl;
        P_ARRAY_Z parrhdr;

        parrhdr = strCreateArrayZ(1, sizeof(TBLHDR));
        ((TBLHDR *)strLocateItemArrayZ(parrhdr, sizeof(TBLHDR), 0))->ct = CT_INTEGER;
        ((TBLHDR *)strLocateItemArrayZ(parrhdr, sizeof(TBLHDR), 0))->phsh = NULL;
        ((TBLHDR *)strLocateItemArrayZ(parrhdr, sizeof(TBLHDR), 0))->cr = CR_PRIMARY_KEY;
        ((TBLHDR *)strLocateItemArrayZ(parrhdr, sizeof(TBLHDR), 0))->strname = "number";


        ptrans = siBeginTransaction();

        ptbl = siCreateTable(ptrans, "Student", parrhdr);

        while (TRUE != siTrylock(ptrans, ptbl, LT_S)) // Share lock.
                ;

        while (TRUE != siTrylock(ptrans, ptbl, LT_X)) // Write lock.
                ;

        for (i = 0; i < 10000; ++i)
                siInsertIntoTable(ptrans, ptbl, i);

        siCommitTransaction(ptrans);

        siReleaseAllTransaction();

        return 0;
}
root@DESKTOP        :/mnt/c/Users/user1/Desktop/svimrdb/src# time ./a.out

real    0m0.127s
user    0m0.102s
sys     0m0.010s
```

图 5-2 数据吞吐测试程序执行结果

6 结论与展望

本文先是介绍了内存数据库的概念及发展状况。其次对svimrdb内存关系数据库做了大致的介绍，分析了svimrdb内存数据库的特性。接下来，文章用了大部分篇幅详细阐述了svimrdb内存关系数据库管理系统的具体实现过程以及运行结果。

在华东理工大学继续教育学院两年多的学习当中，我深切感受到了老师们的谆谆教诲与良苦用心还有同学们热情的学习心态。我也本着不负众望的态度认真打好基本功，努力学习每一门课程。svimrdb内存关系数据库管理系统便是我这两年学习成果的集成与精华。在设计svimrdb的过程中，我运用到了多门课程的内容，其中包括：C语言程序设计、操作系统、数据结构与算法分析、数据库系统概念与原理和软件工程。svrmidb的研究成果是与老师耐心的教导分不开的。在论文写作过程中，老师多次向我提出修改意见，使得论文尽善尽美。

希望svimrdb内存关系数据管理系统能顺利加入数据库管理系统的大家园中，为计算机数据库系统的发展尽自己的绵薄之力。日后我还会对svimrdb进行升级改造和不断维护，增强svimrdb的功能，使之支持SQL，成为一款真正有用且高效的内存关系数据库管理系统。

参考文献

[1] 数据库系统概论（第 5 版）王珊 萨师煊 2014；

[2] Database System Concepts Sixth Edition Abraham Silberschatz Henry F. Korth S. Sudarshan 2012；

[3] Database Systems A Practical Approach to Design, Implementation, and Management Sixth Edition 2016;

[4] Database Systems Concepts, Design and Applications 2009;

[5] Database System Implementation Second Edition Hector Garcia-Molina Jeffrey D. Ullman Jennifer Widom 2010;

[6] 内存数据库关键技术研究 王珊 肖艳芹 刘大为 覃雄派 2007；

[7] 实时内存数据库关键技术的研究与实现 薛竹飙 2006；

[8] Research of main memory database WANG Shan，XIAO Yanqin，LIU Dawei，QIN Xiongpai，王珊，肖艳芹，刘大为 2007;

[9] 内存数据库技术在客票系统中的应用研究 SHAN Xing-hua，单杏花，LIU Xiang-kun，刘相坤，ZHU Jian-sheng，朱建生 2009;

[10] 嵌入式实时内存数据库关键技术研究 黄晨 2008；

[11] MANEGOLDS, BONCZPA, KERSTERN M L. Optimizing Main-Memory Join On Modern Hardware. IEEE trans. On Knowledge and Data Eng. 2002

[12] BONCZ P A. Monet A Next-Generation DBMS Kernel For Query Intensive Applications. PhD Thesis, University van Amsterdam, 2002

[13] SIKKA V, FARBER F. LEHNER W, et al. Efficient Transaction Processing in SAP HANA Database: the End of A Column Store myth. SIGMOD Conference, 2012

[14] HEIMEL M, SAECKER M, PIRK H. et al. Hardware-Oblivious Parallelism for In-Memory Column-Sores. PVLDB 2013

[15] STONEBERKER M, WEISBERG A. The VoltDB Main Memory DBMS IEEE Data Eng. Bull, 2013

[16] 内存数据库关键技术研究 郭一帆 陈亚峰 河南建筑职业技术学院 《数字技术与应用》2013 年 第 5 期

[17] 一种高效内存数据库设计 陆宏 中国电子科技集团公司第二十八研究所 南京《指挥信息系统与技术》2012 年 第 1 期

[18] BALKESEN C. TEUBNER J. ALONSO G, et al. Main-memory Hash Joins on Multi-core CPUs Tuning to the underlying hardware. ICDE 2013

[19] Data Structures and Algorithm Analysis in C. (Second Edition) Mark Allen Weiss 2010.

[20] Mastering Algorithms with C. Kyle Loudon 2012.

[21] Data Structures Using C. R. Krishnamoorthy and G. Indirani Kumaravel 2009

[22] 内存数据库在学校信息系统中的应用研究 信息与电脑(理论版) 2021

[23] 一种数据库内存管理系统及方法 曹思源，王新波 广州辰创科技发展有限公司 2023

[24] 关系型数据库的管理方法及装置 高山岩，郭进伟，肖金亮，韩富晟 北京奥星贝斯科技有限公司 2023

[25] 内存数据库在门诊号源在线预约系统中的性能优化研究 陈锦莹《信息与电脑(理论版)》 2022 年 19 期

[26] 一种基于分布式全内存的关系型数据库技术的研究与实现 赵博妍，骆艳中，刘洋，李欣园，王波 《长江信息通信》 2023 年第 8 期