# 摘 要

实时历史数据库是工业企业自动化体系中的一个关键技术，在电力企业等流程工业中有着广泛应用。本文是在与中国电力科学研究院下属某公司合作开发的GDREAL 实时历史数据库系统的基础上，进行了理论和实践创新之后完成的。GDREAL 是一大型实时历史数据库系统，本文所涉及的磁盘历史数据库和实时内存数据库是 GDREAL 的重要组成部分。

论文以磁盘历史数据库的开发为背景，分析了磁盘历史数据库设计中的问题，对制约历史数据库发展的瓶颈——磁盘 I/O、文件的索引和数据的组织结构等进行了深入的研究，提出了一种新的磁盘存储结构是——Z 树结构，减少了磁盘 I/O 操作次数，极大地提高了磁盘存储性能；采用文件组的方式，设计了一个高效的文件集管理系统；对写入页面的大小和查询性能做了分析。

论文以实时内存数据库的开发为背景，分析了实时数据的存储流程，完成了实时内存数据库和历史数据缓冲区的设计和实现：为了避免内存碎片和减少动态内存分配和释放的开销，设计了一个内存池，将动态内存分配改为预先分配；结合历史数据的特点，设计实现了基于双队列的内存页面 LRU 淘汰算法，将之用于历史数据缓存页面的管理；采用断点续传机制，很好地实现了大量数据的分批返回。

论文详细介绍了实时历史数据库的事务和并发控制，结合传统的事务并发控制协议，区分实时事务和非实时事务，设计了基于优先级继承的有序共享的两段锁协议。

**关键词：**磁盘历史数据库，文件管理，实时数据库，内存管理，并发控制

# Abstract

Real-time historical database is a key technology in the industrial enterprise automation system which has extensive application in the process industry such as power enterprises. On the basis of GDREAL real-time historical database system co-developed with a company affiliated with the China Electric Power Research Institute, this thesis is completed after the theoretical and practical innovations. GDREAL is a large real-time historical database system. The disk historical database and real-time memory database are major components of GDREAL.

This thesis analyzes in-depth the problems in the design of disk historical database, and the disk I/O, file's index and organizational structure of data which are the bottleneck to the development of historical database. It proposes a new disk storing structure-Z tree, which reduces the number of disk I/O operation and greatly improves the performance of disk storage and querying. By the adopting of file sets, an efficient file management system is designed. The size of pages and querying performance are also analyzed.

This thesis analyzes the storage procedure of the real-time data and the design and implementation of the real-time memory database and historical data buffers on the background of development of real-time memory database. A memory pool is designed to avoid the memory chips. To make the change from the dynamic memory allocation to the pre-allocation, it decreases the overhead of dynamic memory allocation and release. The management of history data buffers uses the LRU algorithm based on the bi-queue to process the page elimination. For the destination of returning a large volume of data in batches, a new technology named broken point retransmission is proposed.

This thesis displays the transactions and concurrency control of real-time database. On the basis of the traditional transaction concurrency control protocols and distinguishing the difference of real-time transactions and non-real-time transactions, it brings forward the two phrases protocol with ordered sharing lock based on the priority succession.

**Key words**: disk historical database, disk I/O, file management, real-time database, memory management, concurrency control

# 图表清单

# 独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：曾强 日期：2006年 12月 4日

# 关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：曾强 导师签名：郭王佶

日期：2006 年 12月 5 日

# 第一章 引言

## 1.1 实时历史数据库的研究背景和意义

在我国国民经济中占有重要经济地位的流程工业如石化、炼油、化工、冶金、制药、建材、轻工、造纸、采矿、环保、电力等工业行业，这些行业普遍存在能耗大、产品质量差、生产过程工艺落后、自动化水平低、管理水平低、信息集成度低、综合竞争力弱等现状。在此背景下，如何不断提高企业的竞争力，减少能耗、降低成本、提高质量成为摆在我国流程工业企业面前的一个迫切需要解决的课题[1]。这对当前我国正在建设资源节约型、环境友好型社会，不断提高资源环境保障能力，实现国民经济又快又好发展具有重要意义。

实时数据作为企业的重要数据资源为企业决策提供支持，并对提高企业管理水平和生产效率、保证企业生产安全稳定运行、增强企业的竞争能力以及提高企业的综合效益具有重要意义。实时历史数据库是工业企业自动化体系中的一个关键技术，研究和开发具有自主知识产权的实时历史数据库产品具有重大的理论研究和现实意义。

从 20 世纪 80 年代开始，美国等发达国家陆续出现了有关实时数据处理的论文和专著，对实时数据库的理论进行了很好地探索，使实时数据库成为一个新兴的研究领域，得到了学术界的普遍重视，出现了大量论文和原型系统。研究人员利用数据库技术来解决实时系统中的数据管理问题，并利用实时系统技术提高数据库处理数据的实时性。也出现了若干实用化的商业实时数据库系统，并在实践中得到了很好的应用。但是由于实时数据库作为一新兴的研究领域，提出的要求高出传统数据库，到目前为止，实时数据库的理论和应用研究还未发展成熟，也没有形成统一的实时数据库标准。

另一方面，实时数据库理论的研究主要集中在如下几个方面：数据和数据库的结构与组织；事务的截至时间的软硬性；事务的优先级分派、调度和并发控制的协议与算法[2]；系统资源调度、恢复、通信的协议与算法[3]；查询处理算法；数据和事务特性的语义及这种语义与一致性、正确性的关系等等。但对制约历史数据库发展的瓶颈的文件的索引和数据的组织结构、内存缓存策略及磁盘 I/O 等没有深入的研究。因此，有必要对历史数据库进行专门的研究。

## 1.2 实时历史数据库的研究历史及现状

### 1.2.1 关系数据库简介

数据库系统的萌芽出现于60年代。当时计算机开始广泛地应用于数据管理，对数据的共享提出了越来越高的要求。传统的文件系统已经不能满足人们的需要。能够统一管理和共享数据的数据库管理系统(DBMS)应运而生。数据模型是数据库系统的核心和基础，各种DBMS软件都是基于某种数据模型的。所以通常也按照数据模型的特点将传统数据库系统分成网状数据库、层次数据库和关系数据库三类[4]。

最早出现的是网状DBMS，是美国通用电气公司Bachman等人在1961年开发成功的IDS(Integrated DataStore)。1961年通用电气公司(General Electric Co. )的Charles Bachman成功地开发出世界上第一个网状DBMS也是第一个数据库管理系统——集成数据存储(Integrated DataStore IDS)，奠定了网状数据库的基础，并在当时得到了广泛的发行和应用。

网状数据库模型对于层次和非层次结构的事务都能比较自然的模拟，在关系数据库出现之前网状DBMS要比层次DBMS用得普遍。在数据库发展史上，网状数据库占有重要地位。

层次型DBMS是紧随网络型数据库而出现的。最著名最典型的层次数据库系统是IBM公司在1968年开发的IMS(Information Management System)，一种适合其主机的层次数据库。这是IBM公司研制的最早的大型数据库系统程序产品。

网状数据库和层次数据库已经很好地解决了数据的集中和共享问题，但是在数据独立性和抽象级别上仍有很大欠缺。用户在对这两种数据库进行存取时，仍然需要明确数据的存储结构，指出存取路径。而后来出现的关系数据库较好地解决了这些问题。

1970年，IBM的研究员E.F.Codd博士在刊物《Communication of the ACM》上发表了一篇名为“A Relational Model of Data for Large Shared Data Banks”的论文，提出了关系模型的概念，奠定了关系模型的理论基础。尽管之前在1968年Childs已经提出了面向集合的模型，然而这篇论文被普遍认为是数据库系统历史上具有划时代意义的里程碑。后来Codd又陆续发表多篇文章，论述了范式理论和衡量关系系统的12条标准，用数学理论奠定了关系数据库的基础。关系模型有严格的数学基础，抽象级别比较高，而且简单清晰，便于理解和使用。

1970年关系模型建立之后，IBM公司在San Jose实验室增加了更多的研究人员研究这个项目，这个项目就是著名的System R。其目标是论证一个全功能关系DBMS的可行性。该项目结束于1979年，完成了第一个实现SQL的DBMS。

关系型数据库系统以关系代数为坚实的理论基础，经过几十年的发展和实际应用，技术越来越成熟和完善。其代表产品有Oracle、IBM公司的DB2、微软公司的MS SQL Server以及Informix、ADABASD等等。

### 1.2.2 实时数据库简介

以关系型数据库为代表的传统数据库系统旨在处理永久、稳定的数据，强调维护数据的完整性和一致性，其目标是追求高的数据量和低的系统代价，而不会考虑数据及事务的时间限制。

但新的数据库应用领域不断出现，包括数据通信、电力调度、股票交易、雷达跟踪及交通控制等。这些新的领域与传统的数据库应用领域有着明显不同的区别：

(1) 数据库系统要维护大量的共享数据。实时数据库管理系统和普通的关系型数据库管理系统一个重要的区别是数据库的存储容量。由于流程工业企业生产的复杂性导致了大量的实时信息需要存储，普通的关系型数据库管理系统是无法实现的[5]。

(2) 系统的对数据的处理有很强的时间限制，要求在规定的时间内完成数据操作，其主要目标是使尽量多的事务“在规定的时间要求内”完成，而不是公平地分配系统资源，因此它具有明显的时间约束特点[6]。对数据的时间约束是在数据库的普通一致性要求外，又规定了时态一致性的要求。在实时系统中，具有时间约束的数据主要来自于外部的动态数据，如由传感器采集的现场数据和由这些数据导出的数据。

(3) 实时数据库的调度系统与传统数据库中的事务调度是有差别的。它要求的是能使尽量多的事务在其期限内完成，因此大多数的实时事务调度策略都是围绕事务的优先级进行的。考虑到实时数据库中实时性是第一位的要求，其存储管理策略应使实时数据库在系统运行过程中，占用空间少，并常驻内存，从而保证数据库读取速度快，存取灵活，易于各功能模块之间的数据共享[7]。

传统的实时系统虽然支持任务的定时限制，但只针对结构和关系比较简单的数据，不涉及维护数据的完整性和一致性。因此，实时数据库系统是传统的实时

系统和数据库系统相结合的产物，但不是二者简单的相加。实时数据库使用与传统的关系数据库完全不同的算法来保证实时性，使用比实时系统复杂得多的数据维护机制来管理实时数据。在实时系统中，任务具有时间限制，通常以完成截止期的形式出现，并且以能够在这些事务的截止期之前完成的方式调度。在传统的实时系统中不考虑保持数据库的一致性，而在传统的数据库系统中没有事务的时间限制问题。实时数据库事务和传统的数据库事务一样，也必须保持数据库的一致性，此外它还必须满足事务的定时限制。也就是说为了成功的提交一个事务，将不得不同时满足事务的定时限制和事务的逻辑一致性要求。实时数据库事务处理的目标通常是最大化满足截止期的事务数，而传统数据库事务处理的目标是最小化事务的平均响应时间或最大化事务的平均吞吐量[8]。

1988 年发表的 ACM SIGMOD Record 实时数据库系统专刊提示了 RTDBS (Real-Time Database system，实时数据库系统)研究领域的诞生，标志着实时和传统数据库的融合产生的新兴研究领域的确立。

在数据库理论中，实时数据库系统就是其事务和数据都可以具有定时特性或显式的定时限制的数据库系统[9]。系统的正确性不仅依赖于逻辑结果，而且还依赖于逻辑结果产生的时间[10]。实时数据库的主要特征是在其数据和事务上施加了时间约束。数据的时间约束是在数据的一致性要求之外，增加了时态的一致性要求；事务的时间约束，即为事务规定了一个执行期限[11]。

目前国外已有许多实时数据库产品，如美国的 OSI 公司的 PlantInformation System，简称 PI；美国 AspenTech 公司的 Infoplus21 系统；美国 Honeywell 公司的 Uniformance(PHD)系统；英国 Wonderware 公司的 IndustrialSQL Server 产品等。这些软件产品在技术、设计和解决方案上多数依赖其公司主营业务的发展历史。例如 Industrial 是实时关系型数据库，Uniformance 严格依赖一个大型关系数据库系统，同时这些系统的产品成本和工程成本高，在一些技术和工程方面并不适合中国的流程工业。

90 年代初期，国内也相继出现了一些实时数据库产品，如中国国家电力公司自动化研究院的 NSIS 系统；中国大庆金桥信息技术工程有限公司的 ConRTDB 系统；中国北京三维天地计算技术开发有限公司开发的 SuperInfo 系统；中国北京石林电脑公司的 SLRS 系统：中国北京合利时系统工程股份有限公司的 Rea1MIS 系统等。但由于自身条件和技术的限制，与国外的实时数据库产品还有很大的差距。

### 1.2.3 历史数据库简介

历史数据库往往做为整个实时数据库系统的一部分嵌入其中。与对实时数据库进行研究的热门程度进行比较，就会发现对历史数据库的研究要冷清很多。对制约历史数据库发展的瓶颈的文件的索引和数据的组织结构、内存缓存策略及磁盘 I/O 等没有深入的研究，会影响整个实时数据库系统的效率和性能。

历史数据有助于工艺流程的改进、设备性能的维护和故障原因的诊断。它一般同趋势分析、报表生成和打印等服务紧密结合，在实时数据库系统中发挥着越来越重要的作用。

历史数据库保存实时数据的历史记录。流程工业对历史数据库的需求表现在两个方面：一个是先进控制和实时优化等应用的需要，它们需要的历史数据的特点是近期和实时性，也就是说，它们需要常常或者按照某固定的周期尽快地得到近期的历史数据；另一个是永久存储，提供实时性要求不高的历史数据查询。

历史数据库功能的实现一般采用两种方案，第一种是利用商用关系数据库，比如 MS SQL Server，Oracle，My SQL 等，进行二次开发；第二种是使用过程控制专用的历史数据库。采用两种方案实现的历史数据库均有成功的产品，比如 Simens 的 WinCC 就是由微软公司采用 SQL Server 开发；但是，大部分自动化组态软件采用专用的历史数据库，比如 InTouch 使用了 PI 实时数据库(包含历史数据库部分)，INTELLUTION 使用了实时历史数据库 iffistorian，组态王也自行开发了专用的历史数据库。由于历史数据是基于时间的一些连续的模拟量或数字量(比如温度、压力、流量、阀门开关等)，完全不同于普通关系数据库处理的那些离散的、非连续的、不基于时间的二维关系表数据(比如订单信息、财务信息、人事管理信息等)，所以，从技术应用层面来看，专用历史数据库将会更有优势，使用专用历史数据库在存储效率上占有明显的优势。

国际上系统研究实时数据库的时间不过 20 多年，而同实时数据库紧密相关的历史数据库的研究时间更短。这意味着历史数据库的研究任重而道远。进入 21 世纪后，国外一些知名的自动化公司加大了对流程工业数据库的研发力度，像 INTELLUTION，WONDERWARE 等均推出了自己的历史数据库。这些国外的数据库产品(如 OS 工公司的 PI 系统等)占领了国内大部分流程工业数据库市场，正是由于存在着技术上的垄断性，因而价格昂贵，维护费用高。尽管国内在一些公司和研究机构的积极努力下，也相继推出了一些流程工业数据库系统，不过研究的重点主要集中在实时数据库功能的扩展、稳定性的提高以及增强开放性等，对于历史数据高效、快速的存取以及历史数据库系统构架的研究还远远不够。

## 1.3 本文的工作

本论文是在与中国电力科学研究院下属某公司合作开发的GDREAL系统的基础上，并进行了理论和实践创新后完成的。GDREAL是一大型实时历史数据库系统，最多支持100000个标签点。作者有幸参与开发了这一大型项目的完整开发过程，包括从需求分析、概要设计、详细设计、编码和最终的测试过程。

本文从我国流程工业特别是电力企业的现状入手，充分考虑了实际需求，并在参考了国内外相关领域的理论研究和实际产品的基础上完成的。

主要设计和实现了在整个系统中占核心地位的磁盘历史数据库和实时内存数据库部分，创新点体现在以下几个方面：

(1) 提出了一种新的磁盘存储结构是——Z树结构，减少了磁盘I/O操作次数，极大地提高了磁盘存储性能；

(2) 采用文件组的方式，设计了一个高效的文件集管理系统；

(3) 为了避免内存碎片和减少动态内存分配和释放的开销，设计了一个内存池，改动态内存分配为预先分配；

(4) 结合历史数据的特点，设计实现了基于双队列的内存页面LRU淘汰算法，将之用于历史数据缓存页面的管理；

(5) 采用断点续传机制，很好地实现了大数据量的分批返回；

(6) 结合传统的事务并发控制协议，区分实时事务和非实时事务，提出并设计了基于优先级继承的有序共享的两段锁协议。

论文共分六章，各章的主要内容如下：

第一章，介绍了实时历史数据库的研究背景和意义，对其历史和研究现状进行了详细的描述并对相关内容进行了评述，对课题的研究目标和作者所做的工作进行了说明。

第二章，介绍了作者参与开发的GDREAL实时历史数据库系统，对其基本功能和所要达到的技术指标进行了说明，详细描述了GDREAL的体系结构，包括网络结构、模块划分和数据流程。

第三章，以磁盘历史数据库的开发为背景，分析了磁盘历史数据库设计中的问题，提出了以Z树作为存储结构的思想，结合Windows操作系统的文件操作，最终完成了磁盘历史数据库的设计和实现，并对性能做了分析。

第四章，以实时内存数据库的开发为背景，分析了实时数据的存储流程，完成了实时内存数据库和历史数据缓冲区的设计和实现。

第五章，详细介绍了实时历史数据库的事务和并发控制，结合 Windows 操作系统提供的同步机制，设计和实现了基于优先级继承的有序共享的两段锁协议。

第六章，结论部分，对课题研究工作进行了总结，并对进一步进行实时历史数据库的研究和系统开发提出了设想和建议。

# 第二章 GDREAL 实时历史数据库系统介绍

电力企业利用实时历史数据库系统保存大量的实时历史数据。但在已投入使用的系统中，大都采用国外的实时历史数据库产品，如 PI，EDNA 等。但这样一套实时历史数据库产品动辄上百万元，在实际使用中也与电厂实际有较大差距。而对于国内的实时历史数据库产品由于技术等方面的原因，在实际应用中也不尽如人意。为此，结合电力企业的实际情况，设计和开发了 GDREAL 实时历史数据库系统。

GDREAL 是一大型实时历史数据库系统，从立项到一期工程的完成用了近两年时间，现已进入了现场环境下的试运行，并在明年投入正式运行。二期工程即将展开。

## 2.1 GDREAL 需求分析

### 2.1.1 基本描述

在符合相关国内外技术标准和行业标准，满足软件平台、硬件平台的兼容要求的基础上，实时历史数据库平台应提供数据实时分布采集、压缩存储、方便管理以及可扩展的协同应用，支持生产过程数据实时采集、历史存储，建立数据平台的基础，按照标签点（测量点）的形式定时地收集、存储生产过程带有时间序列性质数据、手工数据、或者其他外部应用数据，并针对这些数据和数据库平台提供实时管理和应用能力。

实时/历史数据库平台的结构设计、系统配置、软件编制，应满足对生产过程的可靠运行要求，必须保证生产过程数据采集过程的实时性、数据的完整性；需要保证系统及其数据的安全，提供适当的数据备份措施和相应的数据安全保证措施；提供严格的用户认证、权限管理和审计手段，并考虑信息保密的时效性。

从开放性和可扩展性考虑，实时历史数据库平台应采用开放式体系结构和分布式系统设计，以满足未来企业信息综合应用的要求。

### 2.1.2 基本功能

本系统的服务器端运行环境为MS Windows 2003 SERVER，客户端运行环境为MS Windows XP PROFESSIONAL/HOME，MS Windows 2000 PROFESSIONAL，可采用不同系统平台版本予以发布。

本系统应实现的基本功能包括：

1. 数据归档与存储

- 本系统应提供面向过程数据经济存储或者优化存储的手段，并且提供数据压缩方法；
- 提供计算引擎或者计算接口，支持数据的二次计算-存储能力，既可直接存储过程数据又可存储过程数据中间计算结果。计算引擎以及接口支持统计功能包括最大值、最小值、平均值和累计值计算功能；
- 针对每条过程数据，记录包含如下内容： 标签点名称、标签点时间标签、标签点描述、数据源地址、数据点工程单位、数据类型、扫描周期、量程范围、数据时间标志、数据值、数据状态(含数据质量或者其它数据点状态信息)等；
- 本系统应能提供针对已有历史数据的移植和扩容方案，并实现多卷存储机制。

2. 数据库管理

- 数据库的建立、删除、修改；
- 设置系统运行参数和系统运行监视；
- 通过手工方式或者自动方式备份数据库系统配置信息和历史归档数据，并可从备份文件恢复数据。

3. 应用编程接口

支持以下采用程序方式对数据库进行维护、存取的编程接口：

- 建立和维护数据库；
- 建立和删除关于数据库的连接；
- 数据插入，以单点方式或批方式；
- 支持数据缓存恢复功能，比如，在服务器端中断工作后，接口机内可缓存一定时间的数据，这些数据在服务器恢复工作后，可以无损失的存入数据库；
- 数据查询和提取，可以单点或多点方式以单一时间、时间段方式提取；
- 数据的修改和删除，可以单点或多点方式以单一时间、时间段方式对数据库进行维护。

4. 事务机制

- 数据更新事务：包括事务开始、更新数据时间、更新数据值、检查数据状态、设置数据状态、事务结束。此功能尤其在数据恢复和备份功能中要有合理的实现；
- 数据计算事务：即二次计算点的数据更新，其包含的操作有：事务开始、获取二次计算所需的数据点值及其对应时间、产生计算结果、更新二次计算点的数据时间、更新二次计算点的数据值、检查数据状态、设置数据状态、事务结束。
- 数据回取事务：其包含的操作有：事务开始、按照指定条件获取数据、事务结束。

5. 权限管理机制

- 系统应支持基于角色的用户权限管理，可以对每个用户指定具体的操作权限，确保其只能进行限定范围内的操作；
- 日志功能，记录数据库系统运行中的操作信息、警告信息、错误信息等；
- 应实现审计功能，记录和追踪数据库配置信息与数据库修改信息。

### 2.1.3 技术指标

本系统的主要技术指标包括事务吞吐量、数据压缩能力、数据存储标签、每秒存取点数，以下指标应在当前的主流计算机系统硬件配置下实现，系统硬件的性能指标理论上应给以保证：

读事务吞吐量：10 TPS

写事务吞吐量：8 TPS；

数据压缩能力：数据压缩率为 1/10；

支持的数据存储标签量：100000；

连接客户端数量：50 个客户；

每秒存取点数：10000 点。

## 2.2 GDREAL 体系结构介绍

### 2.2.1 GDREAL 网络结构

实时历史数据库的网络结构如图 2-1 所示。

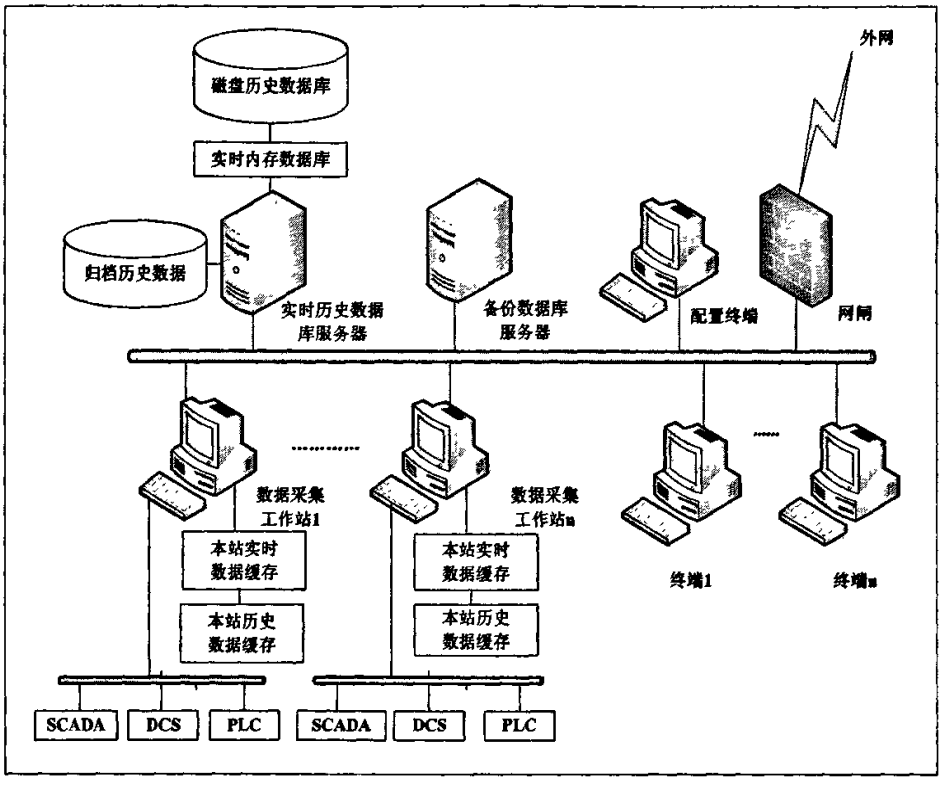


图 2-1 分布式实时数据库网络结构图

各数据采集工作站负责将属于本站的各标签点实时数据实时传输到实时数据库服务器中。服务器采用双机热备份：一个主服务器、一个备份服务器。备份服务器和主服务器同时运行，当主服务器出现故障时，备份服务器能迅速替代主服务器工作。配置终端负责对整个实时数据库的配置。终端为获取数据的最终目的地。如为实时数据则从实时数据库中获得；如为历史数据则从历史数据库中获得压缩历史数据，并在本地解压后得到最终的历史数据。历史数据库是数据存放的最终归宿地，同时也是查询数据的来源。

由图 2-1 可看出，实时历史数据库处于整个系统的核心，有一个好的实时历史数据库，就能快速地存取数据，满足系统的实时性要求。

### 2.2.2 GDREAL 模块划分

GDREAL 分为服务器和客户端。其功能模块的划分如图 2-2 所示：

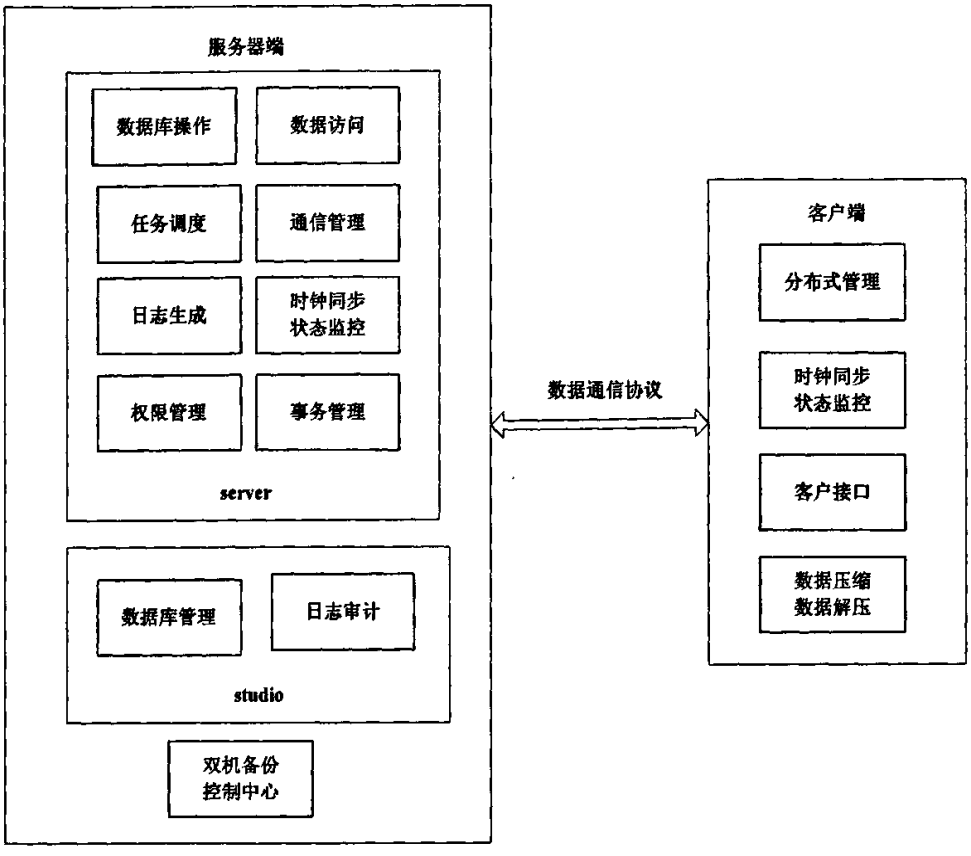


图 2-2 GDREAL 模块划分

对于服务器端各模块名称及其功能如表 2-1 所示：

表 2-1 服务器功能模块

| 模块名称 | 英文名称 | 功能 |
|---|---|---|
| 数据库操作 | DBOperation | 直接操作数据库文件，主要用于建立库，修改库，删除库等操作 |
| 数据访问 | DataInterface | 处理各种数据的访问。实时数据写入，数据查询，数据修改 |
| 通信管理 | CommunicationManage | 管理各种外部应用程序的接口，协调接口工作 |

| | | |
|---|---|---|
| 日志生成 | LogCreate | 产生系统日志，并写入日志数据库 |
| 任务调度 | TaskAttemper | 负责整个系统所有任务的调度和分配 |
| 时钟同步<br>状态监控 | SynchroClock | 同步系统中所有机器的时钟，记录整个系统中机器的工作状态 |
| 权限管理 | PurviewManage | 管理登陆用户的权限 |
| 数据库管理 | DBManage | 管理数据库各种状态信息，进行系统的参数设定 |
| 日志审计 | LogManage | 查看系统日志 |
| 双机备份控制中心 | DBCopy | 设置双机备份，在必要时进行双机备份。 |
| 事务管理 | TransactionProcessing | 管理数据库中的各种事务 |

对于客户端各模块的名称和功能如表 2-2 所示：

表 2-2 客户端功能模块

| 模块名称 | 英文名称 | 功能 |
|---|---|---|
| 分布式管理<br>时钟同步<br>状态监控 | ClientManage | 用于管理客户端的各种状态信息 |
| 客户接口 | ClientInterface | 用于提供所有客户端的接口功能 |
| 数据压缩<br>数据解压 | ClientCompress<br>ClientDecompress | 压缩实时数据<br>解压缩历史数据 |

### 2.2.3 GDREAL 的数据流程

GDREAL 的数据存储过程如图 2-3 所示。首先在接口机(客户端)上，从数据采集点上传的现场数据通过调用接口机上的数据提取模块，产生原始数据送往数据压缩模块，经过死区压缩和旋转门压缩后产生实时数据，将不同标签点的实时数据打包，经过网络传输到服务器。在服务器上，将接收到的数据包解包，经过数据分发模块，将不同标签点的实时数据存放在实时内存数据库的对应位置处。此时，在内存数据库中，数据只是暂时存储，以提高实时数据的查询速度和减少磁盘 I/O。当一个标签点的实时内存库中的数据填满以后，通过数据存储模块存放在磁盘历史数据库中。至此，实时数据便以历史数据的形式完成了永久存储。

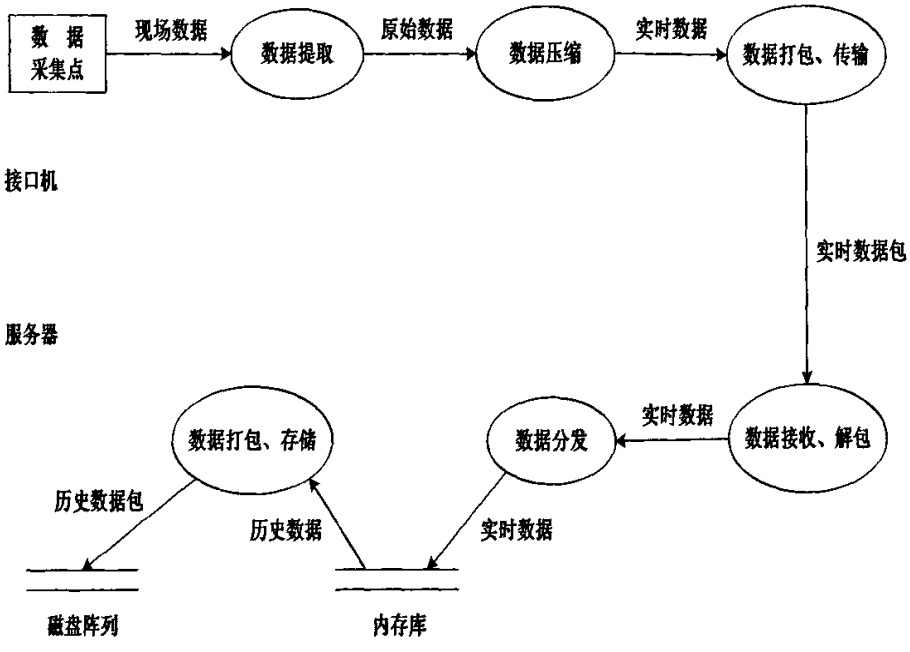


图 2-3 存储数据流图

GDREAL 的查询过程如图 2-4 所示。当用户从客户端发出的查询请求传到服务器上时，首先进行查询分类，判断其是实时数据查询还是历史数据查询。如果是实时数据查询，则直接从实时内存库中获得数据后返回。如果是历史数据查询，首先检查实时内存数据库中是否已存在有用户需要的数据，如有，则立即返回历史数据；否则需要查询磁盘历史数据库，并将读出的数据存放在历史数据缓存中，返回历史数据给用户。

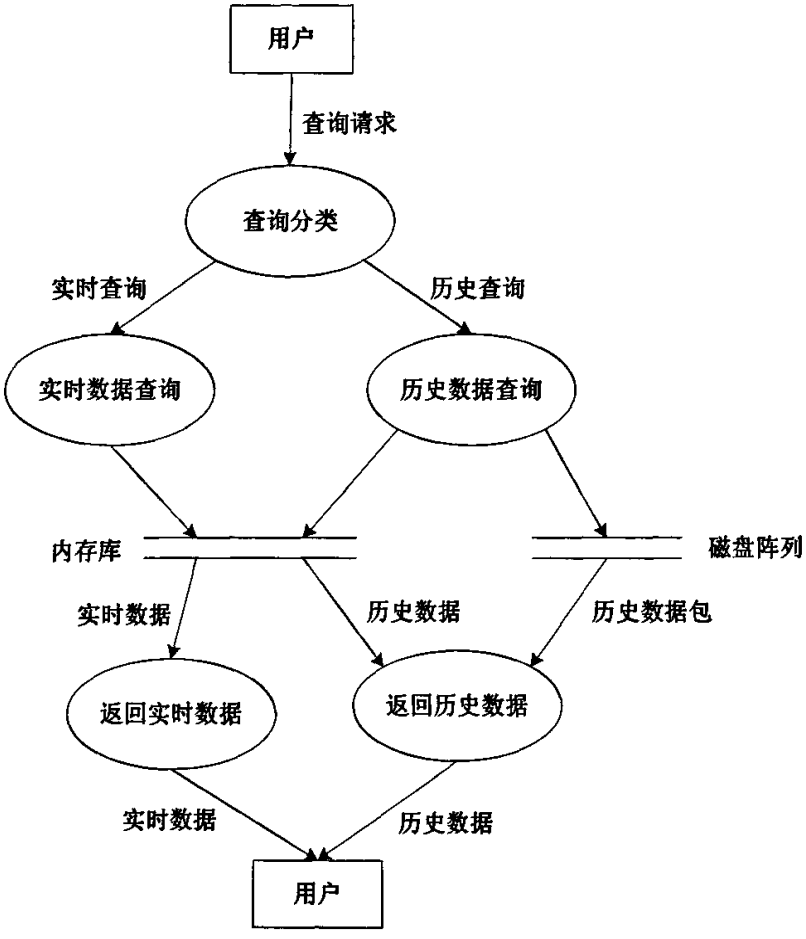


图 2-4 查询数据流图

## 2.3 小结

本章介绍了作者参与开发的 GDREAL 实时历史数据库系统，对其基本功能和所要达到的技术指标进行了说明，详细描述了 GDREAL 的体系结构，包括网络结构、模块划分和数据流图。

# 第三章 GDREAL磁盘历史数据库

磁盘历史数据库是实时历史数据库系统的重要组成部分。磁盘历史数据库永久保存实时数据的历史记录。

## 3.1 磁盘历史数据库设计中的问题

### 3.1.1 存储的数据对象

实时数据库的数据分为两类，实时数据和非实时数据即历史数据。实时数据包括从过程控制系统中实时采集的数据和对实时数据进行二次计算产生的中间数据。实时数据具有时间性，随着时间的推移，实时数据就失效了。历史数据是指过时的实时数据。历史数据对系统应用人员有极其重要的参考价值。它从实时数据库中得到且与时间有关，反映了测量的标签点在某一特定时刻的状态，如果离开了时间，历史数据也就失去了意义。

普通的关系型数据库的设计和应用主要是面对离散的、非连续的、不带有时间标识的数据对象，并通过二维表的方式来表现和建立数据之间的关系，例如订单信息、人事信息、销售数据[12]。实时历史数据库的主要存储对象为生产过程数据，它有以下主要特点：

(1) 历史数据存储的格式相对简单。如连续性、带时标的现场的温度、压力等模拟量；基于时间的连续的事件记录，如阀门开关、电机启停等开关量；离线手工数据，如通过二次计算得到的中间数据。但其占用的空间固定为 1 字节(布尔型开关量)、4 字节(整型、单精度浮点型模拟量)和 8 字节(双精度浮点型模拟量)。所以记录格式少而固定，没有复杂的关系。

(2) 海量数据存储。一套工业设备从投入使用起，一般都要经历 5 年以上时间，期间需要不断记录生产过程中的关键数据，作为改进生产工艺、提高生产效率、监视故障信息的重要参考。这个过程积累的数据一般要几百 G 甚至上 T 的存储空间。如本系统可允许最多 100000 个标签点的数据存储，且其时间标签为毫秒级。

(3) 数据保存的时间间隔相差很大。如在电厂的现场环境中，有些开关量最短几十毫秒保存一次，而一些模拟量最长 15 分钟才保存一次。

这些特点决定了历史数据库具有与关系型数据完全不同的体系结构。在

GDREAL数据库中主要有四种数据类型：布尔型（开关量）、整型、单精度浮点型和双精度浮点型数据。每条记录都是一个四元组：

d =（label，time，state，data）

label表示标签点名称，time表示数据的采集时间，state表示当时的数据状态和data表示采集的数据，一条数据记录的长度对应不同的数据类型分别是9B、12B、12B和16B。

### 3.1.2 实时性

实时数据是指从过程控制系统中实时采集的数据，也是在数据库中存放的当前最新的一条数据，由于其数量少，能立刻从内存中获得；历史数据是指除实时数据外的所有数据，它从实时数据库中得到且与时间有关，反映了实体在某一特定时刻的状态。历史数据的查询虽然不象实时数据那样要求很高的实时性，但作为外部系统的一个客观反映，它表示了外部系统的当前状态，只有数据与外部系统的实际情况相吻合时，数据才有意义。所以也要求历史数据的查询必须高效，能够实现快速响应。

### 3.1.3 存储和查询效率

由于标签点名称是一串字符串，在存储和查询时效率较低，因此，需要对标签点进行编号，在数据进入数据库之前，经过一定的哈希变换，将字符串转换成整个数据库内唯一的标签点编号。在数据库中所有对标签点名称的操作都转换成对标签点号的操作。在将数据返回给用户时，再转换成相应的标签点名称。

磁盘I/O速度是数据检索的最关键因素之一。当读一个文件中的内容时，磁盘驱动器的磁头所做的工作是先定位到该文件所在的磁盘位置，然后读取数据，最后做读后处理——将数据传送至磁盘高速缓存(Cache)和内存中。读盘时，系统先检查数据是否在高速缓存中，如果有则直接读取；如果没有则访问磁盘，也就是进行磁盘I/O。当需要多次读取同一份数据时，Cache的作用很大，但对于第一次读取某个文件，Cache就无能为力了。因此，对于读写操作，都应该建立自己的缓存机制，提高存储和查询效率，同时考虑到磁盘的物理特性。

### 3.1.4 数据插入和可扩展性

历史数据都是随着时间的一维线性变化而变化的，因此，正常情况下，存储在历史数据库中的后一条数据的时间标签一定比前一条数据晚。但是，在现实的生产过程中，可能遇到各种复杂的情况，如数据采集站与服务器端的网络中断。此时，新来的数据的时间标签就可能在已保存数据的前面，需要进行插入操作。以保证数据的时间一致性。

### 3.1.5 顺序和非顺序磁盘 I/O 操作

通常，一个硬盘驱动器由一组驱动器盘片组成。每个盘片都提供用于读取/写入操作的表面。一组带有读取/写入磁头的臂用于在这些盘片之间移动，并且从盘片的表面读取数据或者向其中写入数据。在设计文件和索引时需要关注有关硬盘驱动器的工作方式[13]：

第一，读取/写入磁头和相关的磁盘臂需要移动，以定位到请求的硬盘驱动器盘片上的位置并针对其进行操作。如果数据不按位置顺序分布到硬盘驱动器盘片上，则硬盘驱动器需要花更多的时间来移动磁盘臂（寻道时间）和旋转读/写头（旋转滞后时间）来找到数据。这与按位置顺序分布时的情形完全不同，在该情况下，所需的全部数据都位于硬盘驱动器盘片的一个连续物理扇区上，因此磁盘臂和读取/写入磁头在执行所需的磁盘 I/O 时移动量很少。

根据非顺序数据在磁盘上的分布距离、硬盘盘片可以旋转的速度(RPM)以及硬盘驱动器的其他物理属性，非顺序和顺序情形的时间相差很大：每次非顺序寻道大约花费约 50 毫秒，而顺序寻道则大约需要两三毫秒。顺序 I/O 有助于提高性能，而非顺序 I/O 会降低性能。

第二，读写 8KB 与读写 64KB 所需的时间几乎一样多。在 8KB 到大约 64KB 的范围内，磁盘臂以及读/写头的移动（寻道时间和旋转滞后时间）仍然占一次磁盘 I/O 传输操作所需时间的大部分。因此，从数学角度讲，在需要传输 64KB 以上的数据时，因为 64KB 与 8KB 的传输速度基本上一样，但每次传输所处理的数据却是后者的 8 倍，所以最好尝试尽可能多地执行 64KB 磁盘传输操作。在下文会看到，在 GDREAL 中，一次性读取 64KB 的区，包含 16 个 4KB 页面。

根据经验，大多数硬盘驱动器处理顺序 I/O 操作时所提供的性能是处理非顺序 I/O 操作时的 2 倍。即，需要非顺序 I/O 的操作所花费的时间是执行顺序 I/O 操作的两倍。因此，要尽可能避免可能导致数据库中出现随机 I/O 的情况。

应保证在进行数据查询时，一次能读取尽可能多的有用数据。对历史数据的

查询可能有多种情况，但最基本的一种是查询一个标签点一段时间内的数据。因此，要将一个标签点的数据按照时间先后顺序依次打包存放，读取时也是以包为基本单位进行读取。

### 3.1.6 系统的查询需求

系统对查询的需求如表 3-1 所示：

表 3-1 查询功能表

| 查询目标 | 查询功能 |
|---|---|
| 标签点信息 | 根据标签点号查询 |
| | 根据标签点名称查询 |
| 实时数据 | 查询单个标签点实时数据 |
| | 查询多个标签点实时数据 |
| 历史数据 | 查询一时间段内的所有原始数据 |
| | 查询一时间段内的一时间间隔内的插值数据 |
| | 查询一时间点（一个时刻）的数据 |
| | 查询一时间段内的统计数据，如最大值、最小值、平均值等 |

## 3.2 GDREAL 磁盘历史数据库的原理和结构

按照前面的分析，历史数据库文件结构的设计应遵循以下原则：

(1) 零空间浪费。即在磁盘上存储数据时，要按照页面大小顺序依次存放，不能有磁盘碎片产生；

(2) 零数据移动。数据一旦存储在磁盘上后，不再在磁盘内部进行数据移动操作，所有的操作都是存盘前在内存中完成；

(3) 尽量减少磁盘 I/O 次数。磁盘 I/O 是整个计算机系统最慢的操作之一，因此，要尽可能地减少磁盘 I/O 操作。

### 3.2.1 历史数据存储过程

系统中的历史数据库子系统采用了面向对象模型，将历史数据分为历史数据

缓冲池、当前历史数据文件和历史数据文件队列三级。它们分别记录了数据对象的近期、中期和远期数据。首先将数据存储在历史数据缓冲池中，待缓冲存满后，再将数据成块地存入当前历史文件中，历史数据文件以队列形式存放，当存满后，选择最早的历史数据文件作为可用的空历史文件。如图 3-1 所示。

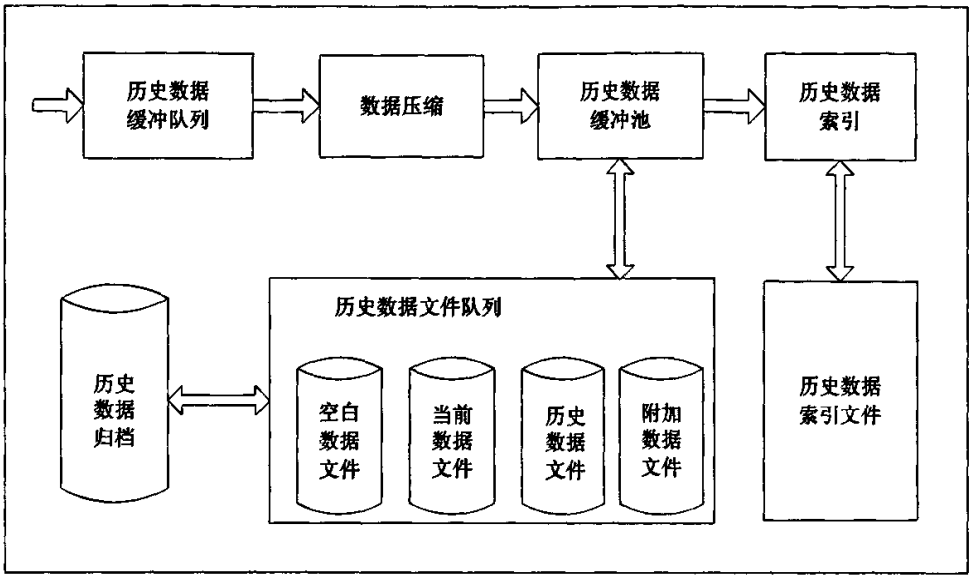


图 3-1 历史数据库数据流示意图

历史数据缓冲队列用于缓冲从现场采集来的数据。数据压缩是不可缺少的一步，通过压缩处理确定需要存储的关键数据点，然后将关键的数据点写进历史数据缓冲池中。历史数据缓冲池是历史数据在内存中的主要缓冲空间，用于存储近期历史数据，从而提高近期历史数据的存取效率，减少不必要的磁盘操作。

历史数据文件队列是历史数据存储的主要磁盘空间，在队列中起码要有一个历史数据文件，某时刻历史数据的追加只在当前历史数据文件进行。当历史文件存满以后，设定下一个可用的空历史数据文件为当前历史数据文件；当已无空文件时，则将最早的文件清空。

磁盘历史数据库在实现时采用异步存储方式，即数据存储操作只需将历史数据提交给后台线程就立即返回，再由系统的后台工作线程将历史数据缓冲队列中需要存储的数据写入磁盘文件中，当操作完成时再通知前台线程数据的插入成功与否。

### 3.2.2 多级索引和数据的存储

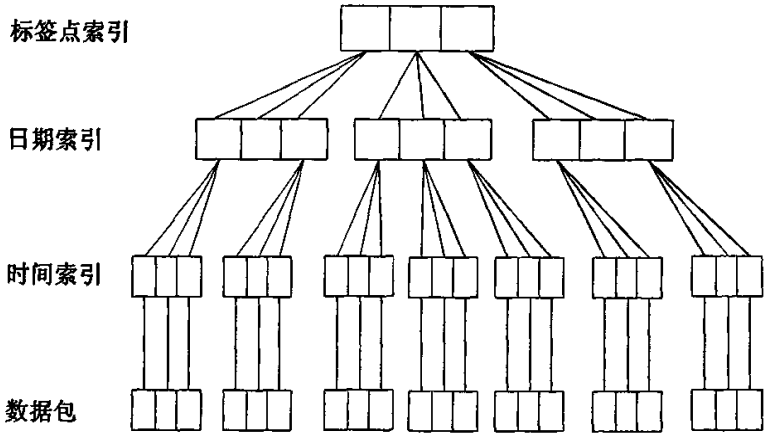


图 3-2 文件索引总体结构

根据历史数据的特点，我们可以设计出文件索引的总体结构，如图 3-2 所示。索引是按 B 树结构进行组织的，并且是聚集索引结构。索引分为三级，标签点索引节点为根节点，时间索引节点是页节点，其余的为中间节点。叶节点即磁盘数据块(页)。其各部分分别对应着不同的文件。

在这样一种存储结构中，要经过三级索引，且对每一次索引的检索都是一个费时的过程：首先找到标签点名称，由于标签点名称是由若干字符构成，因此搜索时要进行字符串的匹配，这将占用大量的 CPU 时间。通过标签点索引找到属于这个标签点的日期索引后，再通过二分法找到这一天的时间索引，最后读出相应的数据。因此，要找到所需的数据，至少要进行四次磁盘 I/O，还要在内存中查找，无疑效率是非常低的。

为此，我们提出了一个全新的数据结构，用以存储历史数据，称之为 Z 树。如图 3-3 所示：

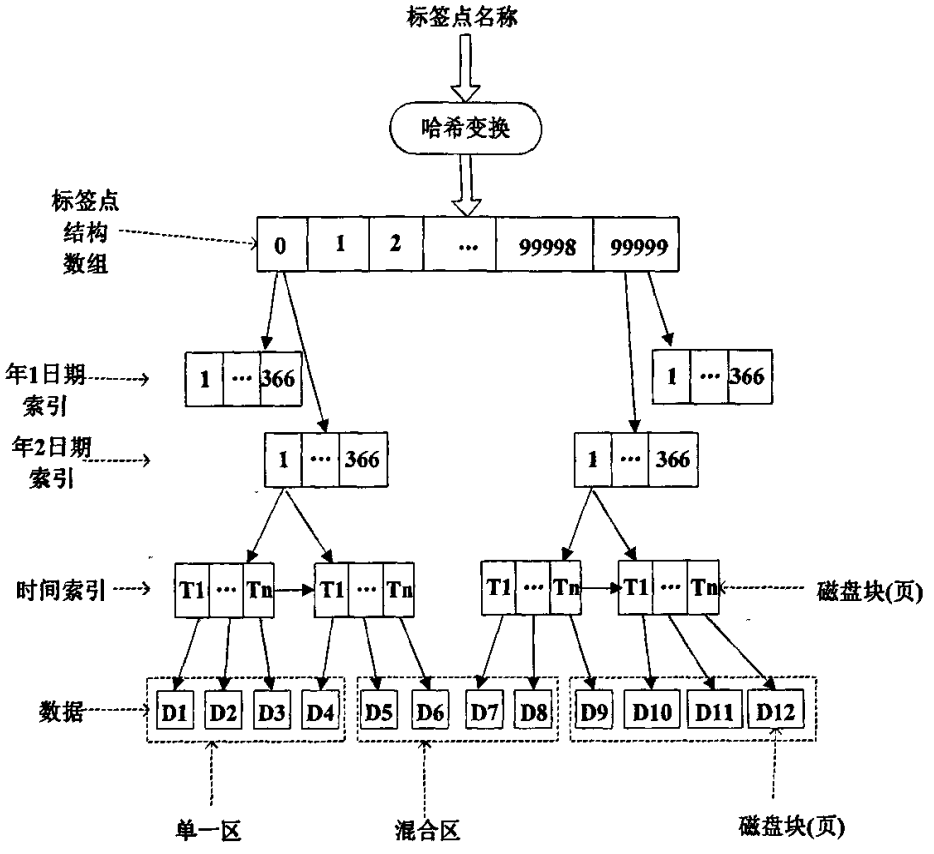


图 3-3 Z 树存储结构

在这种数据结构中，不用再按照图 3-2 所示的存储结构在磁盘中存放三种索引：标签点索引、日期索引和时间索引，以减少磁盘 I/O 次数。

在实时数据库系统中，标签点的名称都是以字符串的方式来命名，如在电厂实时监控系统中，就是以 KKS 编码。如果在整个数据库中都以标签点名称来访问数据，则效率很低，因为对字符串的搜索速度很慢，同时也会占用更多的存储空间。因此，我们在一进入数据库的时候，就通过哈希变换，将标签点名称映射为唯一的标签点编号，所有在数据库内部的访问都是以标签点编号来进行；当从数据库返回时，再将标签点号转换成标签点名称。这样，标签点号对用户来说是透明的，他们只要用标签点名称就能对数据库进行访问。哈希表常驻内存，减少了一次磁盘 I/O 操作。其时间复杂度由哈希算法决定，我们采用是 STL(Standard C++ Library，标准 C++模板库)中的 hash_map Class 来完成。

有了标签点号，再将日期经过变换后转换为代表一年中的某一天的一个整数，

就能直接找到指定的日期所在的磁盘位置，不用再进行查找操作，称这一级索引为直接索引。因此，日期索引节点可直接读取磁盘，一次 I/O 操作就可完成。这一步也称之为直接哈希定位，其时间复杂度为 O(1)。

通过日期索引节点，便可找到这一天的时间索引节点，读盘次数由其时间索引节点的个数决定。将属于这一天的所有时间索引节点读入内存中，通过二分法进行查找，最终找到数据包所在的位置，其时间复杂度为 $O(\log_2 n)$[14]。

为实现 Z 树存储结构，设计以下的索引文件和数据文件：标签点索引节点存放在标签点索引文件中，每个节点占用固定的空间，根据其标签点号依次顺序存放；日期索引节点对应日期索引文件，每个标签索引节点一年的日期索引节点个数固定为 366；时间索引节点对应时间索引文件，一个日期索引节点下对应若干个时间索引节点；历史压缩数据包对应数据文件，一个时间索引节点对应一个历史压缩数据包。图 3-4 是单独的一个标签点的完整结构：

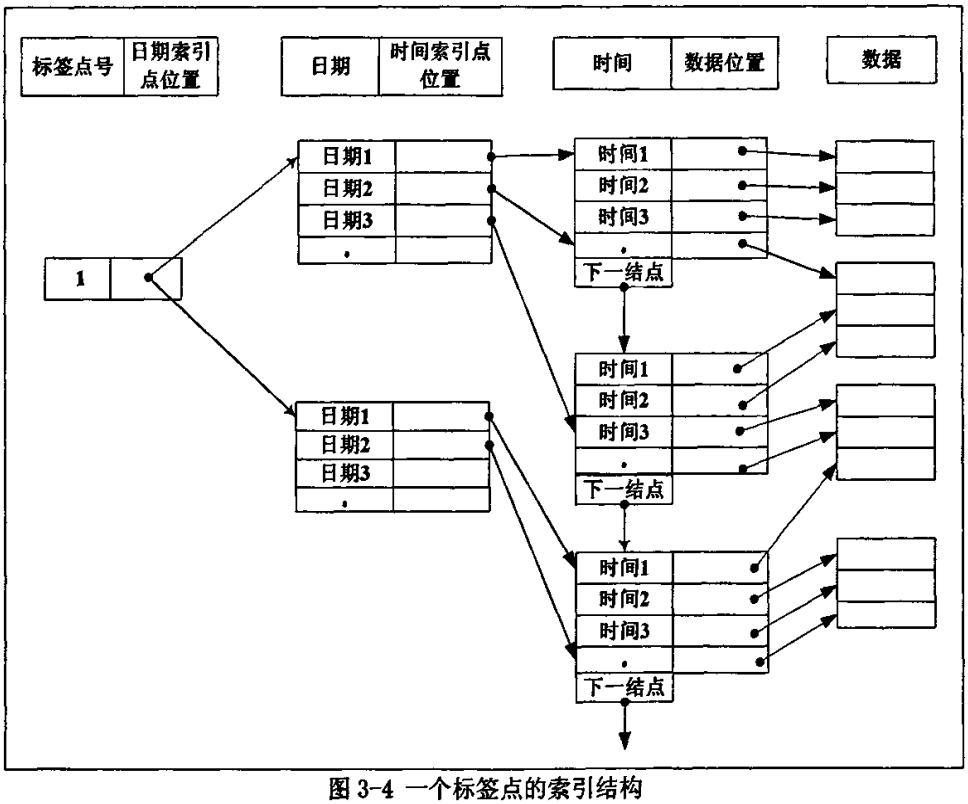


图 3-4 一个标签点的索引结构

下面依次分析。

### 3.2.3 数据文件

如前所述，由于在内存历史数据库中存有每个标签点的内存历史数据，当内存历史数据缓冲区填满后，再一次性地写入文件中，因此在文件中的标签点的数据也是分页(块)的，其大小是由其内存历史数据缓冲区所决定，通过实验分析，将其定为 4KB。以字节序列一个数据包紧接着前一个数据包存放，这样能最大化利用存储空间，几乎没有浪费。同时，由于是连续存放，对磁盘的读写效率都会有很大的提升。数据库中的数据存储在 4KB 块(也称为页或包)中。每组 16 个邻接块是一个 64KB 区。高速缓冲存储器也划分为每页 4KB。一块能存放的数据记录条数分别是 4096/9=455(布尔型)、4096/12=341(整型)、4096/12=341(浮点型)、4096/16=256(双精度浮点型)。磁盘 I/O 操作在区级进行。

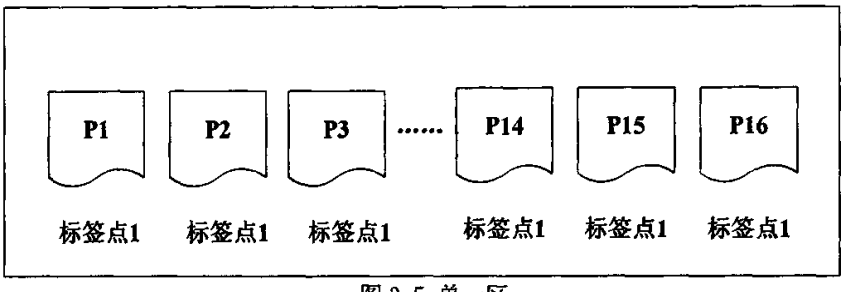


图 3-5 单一区

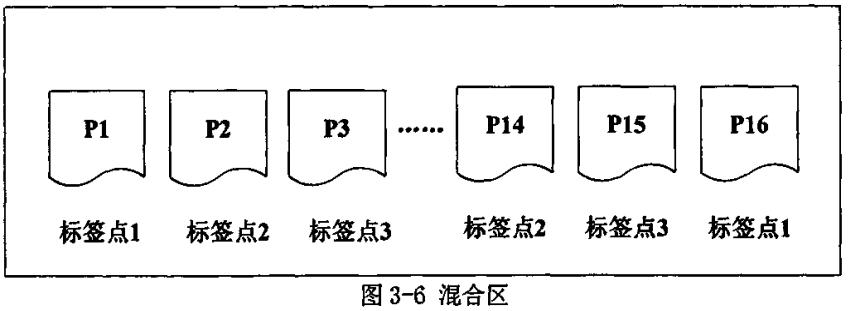


图 3-6 混合区

区用来有效地管理页，所有页都属于一个区。一个区是 16 个物理上连续的块。区分为两种：单一区，由单个标签点所有，区中 16 个块都只能由这个标签点使用，如图 3-5 所示；混合区，最多可由 16 个标签点共享，区中的每一块可由不同的标签点所有，如图 3-6 所示。如果一个标签点的数据较多，则使用单一区，混合区正好相反。通常先从混合区向新的标签点分配页，如果这个标签点的数据在一段时间内(如一天)增长到 16 块，将变成使用单一区进行后续块的分配。系统会根据数据量的变化动态分析决定是在单一区还是在混合区进行分配。

### 3.2.4 时间索引及其文件

由于时间索引随着数据量的增加而动态增加，不能预估其值，因此时间索引文件的空间分配也是动态增加的。只要是一个标签点有数据来，就为其分配一固定大小的块(经实验，块大小定为 4KB，索引中的页不会组合成区)，顺序存放这个标签点的时间索引信息。另一个标签点(不一定按顺序了)再从上一块的末尾分配块，依次存放。当一块写满后再分配一新块，块与块之间通过指针连接起来。每一条时间索引记录占用 15 个字节，所用的标签点索引数据结构如下：

```
struct TIME_INDEX
{
    DWORD startTime;
    DWORD endTime;
    WORD length;
    BYTE fileName;
    DWORD location;
};
```

其中，startTime 是页面的起始时间(一天内的毫秒数)；endTime 是页面的结束时间(一天内的毫秒数)；length 指的数据的长度；fileName 是压缩数据包所在的文件；location 是压缩数据包在 fileName 所指文件中的存放位置。由 fileName 和 location 就可找到数据的存放位置。

从现场采集来的数据一般都是一个 4 字节的 UTC 时间，另外再加一个 2 字节的无符号整数用来表示采集时的毫秒时刻，如果每条记录都以这种格式来保存，则一共要占用 6 字节来保存时间。因此，我们专门把时间里面的日期提取出来，存放在日期索引节点中，这样在时间索引节点中就只存放一天内的毫秒数。24 小时内的毫秒数小于 86400000，因此用一个 4 字节的无符号整数就能表示，节省两个字节的空间。

### 3.2.5 日期索引及其文件

一年建立一个，包含所有的标签点的日期索引信息（预留空间）。按照标签点的编号顺序存放(这样，即使没有标签点指针指向其日期索引节点，但由标签点号就能直接找到其日期索引节点所在的位置，不用再进行查找操作)，每个标签点每

一天占用一个日期索引记录共 19 个字节，一年占用 366(为简化操作，非闰年有一条记录空间的浪费)条记录共 366*19=6954B 的磁盘空间。要查询某个标签点某一天的日期索引记录，可直接定位到指定的位置，其计算公式如下：

$$Offset = (N-1)*T+(day-1)*R$$

Offset 表示当前记录在文件中的位置，N 表示标签点号，T 表示一个标签点一年日期索引信息占用的字节数，固定为 6954B，R 表示一个日期索引记录占用的空间共 19B，day 表示日期是一年内的第几天。如要查询标签点 2 的某年 1 月 20 日的日期索引记录，其位置位于(2-1)*6954+(20-1)*19=7125 处。每个日期索引记录有个指针指向其时间索引节点所在的位置(由时间索引节点的文件名和偏移量表示)。

建立日期索引文件的目的有二：一是为了加快查询，可以把从一个标签点的所有数据中查询改变成迅速定位到在一天内的查询，每个标签点的日期索引节点都有固定的存储位置，通过标签点号可以直接根据地址空间折算出来；二是节省了存储空间——把日期值从时间值中分离出来，不用在每一个压缩包上都存放日期值。所用的日期索引数据结构如下：

```
struct DATE_INDEX
{
    BYTE date[3];
    DWORD timeIndexCount;
    BYTE fileName;
    DWORD location;
    WORD offset;
    BYTE affixationFileName;
    DWORD affixationLocation;
};
```

其中，date[3]是日期，年月日各一个字节；timeIndexCount 是已写入磁盘文件的时间索引的个数，即一天内有多少个数据包；fileName 是时间索引节点所在的文件名；location 是时间索引节点信息所在时间索引节点块块首在文件中的存放位置；offset 是时间索引点信息在当前时间索引节点块中的存放位置；affixationFileName 是指这一天内是否有插入的数据，有则为文件的名字，没有则为－1；affixationLocation 是数据在附加文件中的位置。labelIndex 和 timeIndex 不保存在磁盘上。

### 3.2.6 标签点索引及其文件

整个数据库里只有这一个。由于整个历史数据的组织都是以标签点为主索引，因此每来一压缩数据包需要写文件时，都会修改标签点的索引。为此，将标签点的索引信息常驻内存。而为了备份和恢复的目的才建立标签点索引文件，每改动一次标签点的索引值，就要将在内存中的改动保存到文件中，以防系统出现故障后文件信息丢失。查询时不用读取标签点索引文件。标签点索引按照标签点的编号顺序存放，每个标签点记录占 12 个字节。所用的标签点索引数据结构如下：

```
struct LABEL_INDEX
{
    BYTE firstYear;
    WORD remindCount;
    BYTE lastFileName;
    DWORD timeLastLocation;
};
```

其中，firstYear 是本标签点建立的年份，remindCount 是指当前块的剩余字节数；lastFileName 是为本标签点分配的时间索引节点的最后一块的能被写入的位置，即当前标签点时间的写入位置所在的文件；timeLastLocation 是为本标签点分配的时间索引节点的最后一块的能被写入的位置，即当前标签点时间的写入位置。每当把一个内存页面写入磁盘时，这三个值都会发生相应的变化。

### 3.2.7 附加文件

当某一天有插入数据时，数据不再成块保存，而是进行一定的组合后存储在附加文件中，其位置保留在日期索引节点中。如有多个插入数据，以链表的形式连接起来。由此可以看出，如有插入数据时，查询的效率将明显降低。但考虑到历史数据的插入较少发生，以性能来换取灵活性是可行的。

## 3.3 GDREAL 磁盘历史数据库的实现

### 3.3.1 Windows 的文件操作

设计一个系统，是建立在一操作系统平台的基础上，需要对操作系统有深入的理解，特别是分析其提供给用户的服务，选择最好的方式来实现。实时数据库系统也不例外。因此，有必要介绍一下在 Windows 平台下的文件操作。

与计算机执行的大多数操作相比，设备 I/O 是最慢的之一，读取文件时涉及到磁盘的 I/O。Windows 提供了两种设备 I/O 方式：同步方式和异步方式[15]。在同步 I/O 方式中，如果要读写磁盘文件，则 CPU 必须等到数据读取完成后才能继续工作，即使 CPU 在这段时间内是空闲的，也不能做其它工作；但在异步 I/O 方式中，可以使用 Windows 的多线程结构，在系统对磁盘文件进行读写操作时，应用程序的其余代码可以继续异步执行。

Windows 有 4 种技术用于进行异步 I/O。当进行异步 I/O 时，必须向操作系统发出一个 I/O 请求。操作系统队列化了各种 I/O 请求，在内部处理它们。当系统在处理 I/O 请求时，线程可以返回继续执行。当操作系统处理完 I/O 请求后，会通知一个结果：发送、接收或错误。发出 I/O 请求的方式对 4 种技术几乎是相同的，区别只在于被通知结果的方式。

表 3-2 简单地总结了 4 种不同的异步操作技术。

表 3-2 异步操作技术

| 技　术 | 特　点 |
|---|---|
| 使一个设备内核对象变化为有信号 | 只能对单个设备进行单个 I/O 请求。一个线程发出 I/O 请求，另一个线程进行处理。 |
| 使一个事件内核对象变为有信号 | 能对单个设备进行多个同时的 I/O 请求。一个线程发出 I/O 请求，另一个线程进行处理。 |
| 告警 I/O | 能对单个设备进行多个同时的 I/O 请求。发出 I/O 请求的线程也必须处理它。 |
| I/O 完成端口 | 能对单个设备进行多个同时的 I/O 请求。一个线程发出 I/O 请求，另一个线程进行处理。 |

1．使设备内核对象有信号

在异步调用 ReadFile 和 WriteFile 时，要分配一个 OVERLAPPED 结构作为参

数，并初始化其成员。如果I/O请求是异步执行，函数立即返回FALSE，指出I/O请求没有完成，可以继续执行下一条代码，同时开始异步文件操作。当线程代码执行到一定点后，会等待数据被完成装入缓冲区后才能继续执行。Windows把文件句柄看作是可同步的对象——处于有信号或无信号。在函数执行前，将文件句柄设为无信号态。当所有的数据都被读出或写入文件后，系统将文件句柄设为有信号态。通过调用GetOverlappedResult函数，检查文件句柄是否为有信号态，并判断文件操作是否成功。

2. 使一个事件内核对象变为有信号

第一种方法不能处理多个I/O请求。如果要同时对同一个文件进行多个异步操作，不能通过等待文件句柄变为有信号来同步线程，因为通过文件句柄无法判断是哪一个异步操作完成了。

在调用读写函数时，如果想同时执行多个异步I/O请求，通过设置OVERLAPPED结构的hEvent成员对应不同的异步操作，则每次I/O操作完成后，设置其为有信号。在通过GetOverlappedResult函数获取返回结果时，设置fWait为TRUE，这样就是通过OVERLAPPED结构的hEvent成员来判定操作是否完成，从而实现同时多个异步操作。

3. 告警I/O

Alertable I/O(告警I/O)提供了更有效的异步通知形式。当一个线程被创建时，系统还创建了一个队列，并把它同线程关联起来，称为异步过程调用(APC，Asynchronous Procedure Call)队列。

ReadFileEx/WriteFileEx在发出I/O请求的同时，提供一个回调函数（APC过程），当I/O请求完成后，一旦线程进入可告警状态，回调函数将会执行。

线程进入告警状态时，内核将会检查线程的APC队列，如果队列中有APC，将会按FIFO方式依次执行。如果队列为空，线程将会挂起等待事件对象。以后的某个时刻，一旦APC进入队列，线程将会被唤醒执行APC，同时等待函数返回WAIT_IO_COMPLETION。

4. I/O完成端口

使用告警I/O的主要缺点是发出I/O请求的线程也必须是处理结果的线程，如果一个线程退出时还有未完成的I/O请求，那么应用程序将永远丢失I/O完成通知。但I/O完成端口没有这个限制。

I/O完成端口是一种机制，通过这个机制，应用程序在启动时会首先创建一个线程池，然后该应用程序使用线程池处理异步I/O请求。这些线程被创建的唯一目

的就是用于处理 I/O 请求。对于处理大量并发异步 I/O 请求的应用程序来说，相比于在 I/O 请求发生时创建线程来说，使用完成端口就可以做的更快且更有效率。

CreateIoCompletionPort 函数会使一个 I/O 完成端口与一个或多个文件句柄发生关联。其原型如下：

- HANDLE CreateIoCompletionPort(
  HANDLE FileHandle， // 以异步方式打开的文件句柄
  HANDLE ExistingCompletionPort， // 完成端口句柄
  ULONG_PTR CompletionKey， // 完成键。可以自己定义的参数，把一个结构的地址赋给它，供用户使用
  DWORD NumberOfConcurrentThreads )； // 线程数目

当与一个完成端口相关的文件句柄上启动的异步 I/O 操作完成时，一个 I/O 完成包就会进入到该完成端口的队列中。对于多个文件句柄来说，这种机制可以用来把多文件句柄的同步点放在单个对象中。每当一个文件中的异步操作完成，就会把一个 complete package 放到队列中，这样我们就可以使用这个来完成所有文件句柄的同步。

调用 GetQueuedCompletionStatus 函数，某个线程就会等待一个完成包进入到完成端口的队列中，而不是直接等待异步 I/O 请求完成。线程就会阻塞于它们的运行在完成端口（按照后进先出队列顺序的被释放）。这就意味着当一个完成包进入到完成端口的队列中时，系统会释放最近被阻塞在该完成端口的线程。其原型如下：

- BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort， // 完成端口句柄
  LPDWORD lpNumberOfBytesTransferred， // 传输的字节数
  LPDWORD lpCompletionKey， // 完成键
  LPOVERLAPPED *lpOverlapped， // 指向 LPOVERLAPPED 的指针
  DWORD dwMilliseconds)； // 等待的时间

调用 GetQueuedCompletionStatus，线程就会与某个指定的完成端口建立联系，一直延续至该线程的存在周期，或被指定了不同的完成端口，或者释放了与完成端口的联系。一个线程只能与最多不超过一个的完成端口发生联系。

完成端口最重要的特性就是并发量。对于并发量设置为计算机中 CPU 的数目的两倍。如果事务处理需要一个漫长的计算时间，一个比较大的并发量可以允许更多线程来运行。虽然完成每个事务处理需要花费更长的时间，但更多的事务可

以同时被处理。

总结起来，完成端口在磁盘操作中有以下优点：后台线程专门用来处理和磁盘 I/O 有关的操作；支持多个线程并行地完成异步 I/O 操作；在创建完成端口时，就创建了指定数目线程的线程池，而在使用完成端口的过程中不用创建新的线程，尽管线程的创建的开销比进程要小得多，但也会不是没有开销，而这些线程在其它程序执行时是空闲的，性能就能进一步提高；线程的个数一般为 CPU 数目的两倍，没有多余的活动线程，这减少了线程的上下文件切换所浪费的 CPU 周期。因此，在我们所开发的系统中所使用的就是 I/O 完成端口，减少 I/O 操作所占用的时间，提升整个服务器的性能。以下代码实现了异步文件写：

```
hDataFile = CreateFile(".\\fileName.dat", GENERIC_WRITE|GENERIC_READ,
                    FILE_SHARE_READ, NULL, OPEN_EXISTING,
                FILE_FLAG_NO_BUFFERING|FILE_FLAG_OVERLAPPED,NULL);
hCompletionPort = CreateIoCompletionPort(hDataFile, hCompletionPort,
                                    dwCompleteKey,   iNumOfCPU*2);
```

首先以 FILE_FLAG_OVERLAPPED 标志调用 CreateFile 函数建立文件以进行异步操作，然后通过 CreateIoCompletionPort 创建 I/O 完成端口，并将文件句柄与 I/O 完成端口联系起来。此时，就可发送异步文件操作了。_WriteFile 函数封装了异步文件写操作：

```
_WriteFile(HANDLE handle, DWORD offset, char *& t_pbBuffer,
        DWORD length);
```

各参数分别表示文件句柄、偏移量、返回的数据缓冲区和数据长度。其主要实现步骤如下：

```
// 填充结构
LPIORequest lpIORequest;
lpIORequest->Overlapped->Offset = offset;
lpIORequest->Overlapped->OffsetHigh = 0;
lpIORequest->pBuffer = t_pbBuffer;
// 写文件
fOk = ::WriteFile(handle, lpIORequest->pBuffer, length,
    &nTempCount, lpIORequest->Overlapped);
if (!fOk || (GetLastError() != ERROR_IO_PENDING))
    return error;
```

```
// 获取返回数据
fOk = GetQueuedCompletionStatus (m_hCompletionPort, &dwCount,
    &dwCompleteKey, (LPOVERLAPPED *)&lpIORequest, INFINITE);
if (fOk || (GetLastError() == NO_ERROR))
{   if (dwCount == length)   return success;   }
else     return err;
```

填充 lpOverlapped 结构中的文件偏移量后，调用 WriteFile 函数提交文件写操作后立即返回，程序可继续执行，通过调用 GetQueuedCompletionStatus 函数判断文件操作是否成功。为获取返回数据，需要定义一数据结构：

```
typedef struct
{   OVERLAPPED Overlapped; // 异步文件操作的结构
    char *pBuffer; // 异步文件操作的读写缓冲区
} IORequest, * LPIORequest;
```

在异步操作时，将其做为参数传入，通过 GetQueuedCompletionStatus 中的返回参数 LPOVERLAPPED，将其强制转换为 LPIORequest 类型，pBuffer 中存放的就是读写的数据。

### 3.3.2 GDREAL 磁盘历史数据库的初始化

对于 GDREAL 磁盘历史数据库通过类 CHDiskDatabase 实现，CHDiskDatabase 的主要数据成员如下：

```
class CHDiskDatabase
{   // 标签点索引节点指针，是所有内存和磁盘存储结构的顶层节点。
    LABEL_INDEX * m_pLabelIndex;
    //整个库唯一的完成端口。所有的文件操作通过完成端口完成。
    HANDLE m_hCompletionPort;
    HANDLE m_fGlobalLabel;    //标签索引文件。
    HANDLE m_fDateIndex[DATE_FILE_COUNT];   //日期索引文件。
    HANDLE m_fTimeIndex[DATE_FILE_COUNT][TIME_FILE_COUNT]; //时间索引文件。
    HANDLE  m_fData[DATE_FILE_COUNT][DATA_FILE_COUNT];   //数据文件。
```

// 与每个文件相关联的完成键。

DWORD m_dwCKData[DATE_FILE_COUNT][DATA_FILE_COUNT];

// 索引文件末尾标记。不同于实际上的 EOF，是逻辑上的文件结尾,用来分配一新的时间索引块的起始位置。

DWORD m_nLastLocation;

// 数据文件末尾标记。同上。当前数据文件的当前位置,即文件末尾，也就是下一条数据应写入的位置。

DWORD m_nLastDataLocation;

BYTE m_nCurrentIndex; // 当前值最大的时间索引文件的序号(即文件名)。

BYTE m_nCurrentData; // 当前数据文件。

BYTE m_nCurrentYear; // 当前年。当“年”发生变化时，需要重建日期索引文件。}

系统启动时创建一CHDiskDatabase类的对象，通过构造函数CHDiskDatabase()完成初始化操作，包括对完成端口、完成键和当前文件进行初始化，创建新的文件组和分配Z树的根节点（标签点索引节点），最后调用__Initialize函数完成标签点索引节点的初始化：

```
// 获得系统 CPU 个数
SYSTEM_INFO SystemInfo;
GetSystemInfo(&SystemInfo);
// 创建完成端口。此时未关联任何设备。最后一个参数表示线程的个数，此处设为 CPU 数目的两倍。
m_hCompletionPort = CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0, SystemInfo.dwNumberOfProcessors*2);
if (m_hCompletionPort == NULL)
    return err;
// 初始化每个文件的完成键，用以完成端口唯一标识各个不同的文件。
for (INT i=0; i<DATE_FILE_COUNT; i++)
    for (INT k=0; k<TIME_FILE_COUNT; k++)
            m_dwCKTime[i][k] = j++;
// 初始化当前文件。
m_nCurrentYear = (BYTE)(COleDateTime::GetCurrentTime().GetYear()-2000);
m_nCurrentIndex = (BYTE)0;
```

```
m_nCurrentData = (BYTE)0;
// 创建新的文件组。
__CreateFile(m_nCurrentYear, 'l');          //标签索引文件。
__CreateFile(m_nCurrentYear, 'y');          //日期索引文件。
__CreateFile(m_nCurrentIndex, 't');         //时间索引文件。
__CreateFile(m_nCurrentData, 'd');          //数据文件。
//  分配标签索引节点空间(常驻内存)。
m_pLabelIndex = new   LABEL_INDEX[LABEL_COUNT];
__Initialize();          //初始化。
```

__Initialize 函数完成对当前文件位置、标签点索引节点进行初始化：

```
//初始化时间索引节点和数据页面位置。
m_nLastLocation = 0;
m_nLastDataLocation = 0;
//初始化标签索引节点并分配一天(当日)的日期索引节点。
for (INT i=0; i<LABEL_COUNT; i++)
{  //此处初始化是为了在建立第一个时间索引节点的时候用作比较用，而其它的数据无关紧要。
    m_pLabelIndex[i].firstYear = -1;
    // 从内存池中申请一日期索引节点。每一索引节点只在内存中保留当日的日期索引节点信息
    while (-1 == GetDateNode(m_pLabelIndex[i].dateIndex)); }
```

在新建文件组以及建立新的文件时，要用到__CreateFile 函数，将文件名和文件类型做为参数传入：

```
bool __CreateFile(BYTE nParaFileName, BYTE nParaFileType);
```

根据不同的文件类型来创建相应的文件，并将其与完成端口相关联。如建立时间索引文件：

```
sprintf(cgTempFileName, ".\\%d%d.iti", m_nCurrentYear, nParaFileName);
m_fTimeIndex[m_nCurrentYear][nParaFileName]
    = CreateFile(cgTempFileName, GENERIC_WRITE|GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        ILE_FLAG_NO_BUFFERING|FILE_FLAG_OVERLAPPED, NULL);
```

// 将文件句柄与完成端口建立联系，m_hCompletionPort 是在初始化时建立的完成端口的句柄。最后一个参数表示线程的个数，初始化时已设置，此处设为 0 表示默认值。

```
CreateIoCompletionPort(m_fTimeIndex[m_nCurrentYear][nParaFileName],
    m_hCompletionPort, CKTime, 0);
```

### 3.3.3 追加历史数据

追加操作做为一个事务来完成。在将历史数据页面写入磁盘文件时的操作过程如图 3-7 所示。

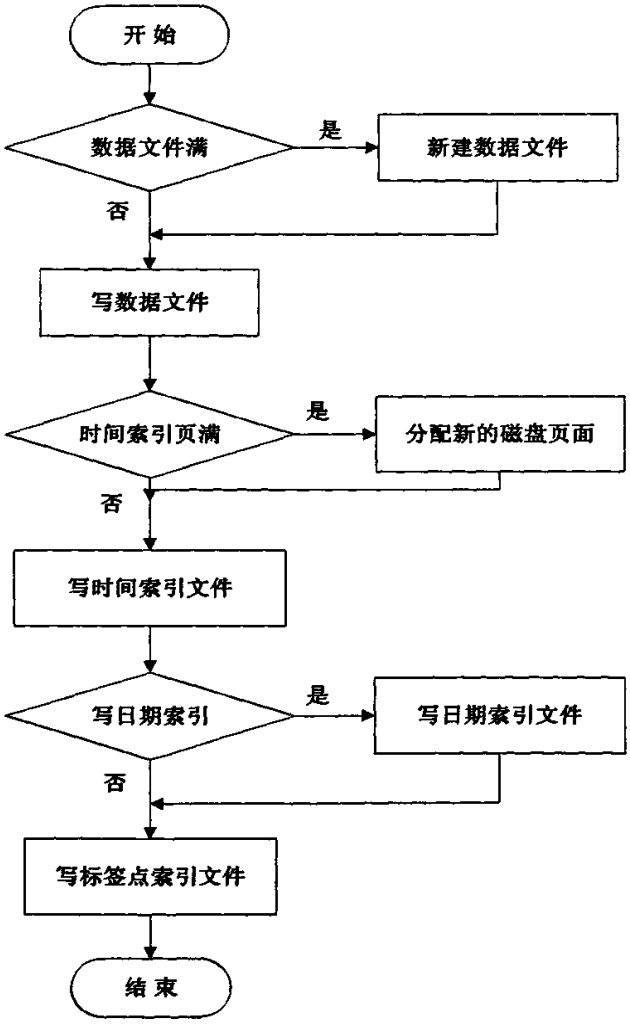


图 3-7 追加流程图

当需要写磁盘文件时，首先检查数据文件是否超出设置的长度。若超出，则新建一文件，将历史数据页面写入磁盘文件。若数据未能成功写入磁盘，则回滚，事务操作失败。否则，再写时间索引文件。由于每一标签点是按照页面来组织时间索引的，因此系统要记录时间索引的写入位置，若此页面已满，则新申请一个磁盘页面，将时间索引写入，并更新时间索引的写入位置，供下一次使用。若时间索引未能成功写入磁盘，则回滚，事务操作失败。否则，检查是否是新的一天的数据，若是，则需要将前一天的日期索引信息写入磁盘。对于日期索引的位置，前已述及，可直接定位。若日期索引信息未能成功写入磁盘，则回滚，事务失败。否则，将标签点索引信息写入磁盘。成功，则事务成功，追加操作正确执行。否则，也要回滚，事务失败。因此，追加操作做为一个事务来完成，要么不做，要么全做，数据的状态必须处于一致状态。追加操作的函数原型：

INT Write(DWORD p_nLabelIndex, DWORD nParaDataLength, WORD p_nRecordLength, bool p_bWriteDate);

参数 nParaIndexNum 表示索引节点号，nParaDataLength 是要写入的数据长度，p_nRecordLength 是一条数据记录的长度，p_bWriteDate 表示是否需要写日期索引。

主要实现算法如下：

```
//判断数据文件是否超出长度。
if(m_nLastDataLocation+nParaDataLength > FILE_LENGTH)  //文件长度超出，新建一数据文件。
{  if (!__CreateFile(m_nCurrentData+1, 'd'))
       m_nLastDataLocation = 0; }
//写数据文件。
_WriteFile(m_fData[m_pLabelIndex[p_nLabelIndex].dateIndex->date[0]]
   [m_nCurrentData], m_nLastDataLocation,
   m_pLabelIndex[p_nLabelIndex].dateIndex->pBufNode->lpBuffer,
   nParaDataLength);
//获取时间索引
__CombineTimeIndex(p_nLabelIndex, lpTempStr);
// 记住写入时间索引节点的位置。
nTempFileName = m_pLabelIndex[p_nLabelIndex].lastFileName;
t_nAffixLocation =  m_pLabelIndex[p_nLabelIndex].timeLastLocation;
// 写入后当前时间索引节点块满。最后有 5 个字节写下一个时间节点的位置。
```

```
if (LENGTH_TIME == (m_pLabelIndex[p_nLabelIndex].remindCount-5))
{  // 分配新的时间索引节点页面
    _AllocateTimeIndex(p_nLabelIndex);
    // 写入下一时间索引节点的位置。
    lpTempStr[LENGTH_TIME] = m_pLabelIndex[p_nLabelIndex].lastFileName;
    *((DWORD *)(lpTempStr+LENGTH_TIME+1)) =
        m_pLabelIndex[p_nLabelIndex].timeLastLocation; }
//写时间索引文件。此处应使用的是分配索引节点块前的值。
_WriteFile(m_fTimeIndex[m_pLabelIndex[p_nLabelIndex].dateIndex->
date[0]][nTempFileName],  t_nAffixLocation, lpTempStr, LENGTH_TIME+5);
if (p_bWriteDate)  // 写日期索引节点。
    _WriteDateIndex(p_nLabelIndex);
_WriteLabelIndex(p_nLabelIndex); // 写标签索引节点信息到文件。
```

只有当标签点索引文件正确写入后，才更新以下值，表示写入成功；否则所有的操作都无效，事务操作失败。

```
//更新本标签点的时间索引节点的当前位置及剩余块容量。
m_pLabelIndex[p_nLabelIndex].timeLastLocation += LENGTH_TIME;
m_pLabelIndex[p_nLabelIndex].remindCount -= LENGTH_TIME;
// 更新内存数据库
m_pLabelIndex[p_nLabelIndex].memRecordCount = 0;
m_pLabelIndex[p_nLabelIndex].memCurrentPosition =
    PACKAGE_HEADER_LENGTH;
m_pLabelIndex[p_nLabelIndex].dateIndex->timeIndexCount ++;
//更新数据文件逻辑结尾标志。
m_nLastDataLocation += nParaDataLength;
// 如果历史内存库中存在这一天的数据，则把此节点加入到链表中。
DATE_INDEX * t_pstDateIndex = m_pLabelIndex[p_nLabelIndex].
    DateIndexAvl.GetNode(m_pLabelIndex[p_nLabelIndex].dateIndex);
if (t_pstDateIndex != NULL)
    t_pstDateIndex->DataAVL.AppendNode
        (m_pLabelIndex[p_nLabelIndex].dateIndex->pBufNode);
```

### 3.3.4 查询历史数据

查询操作也是做为一个事务来完成。从磁盘文件读取页面数据时的操作过程如图 3-8 所示。首先读取日期索引，从日期索引中获得这一天的时间索引存放位置，将这一天的时间索引全部读入内存后，进行二分查找，找到起始时间和结束时间的时间索引节点，获得每个数据页面的存放位置，最后读出数据页面，并将整个页面做为一个节点插入到历史数据缓存的 AVL 树中，以便以后查询同类数据时从历史数据缓存中就可找到，不用再进行磁盘 I/O 操作。历史数据缓存的组织请参见 4.4 节 “历史数据缓冲区”。

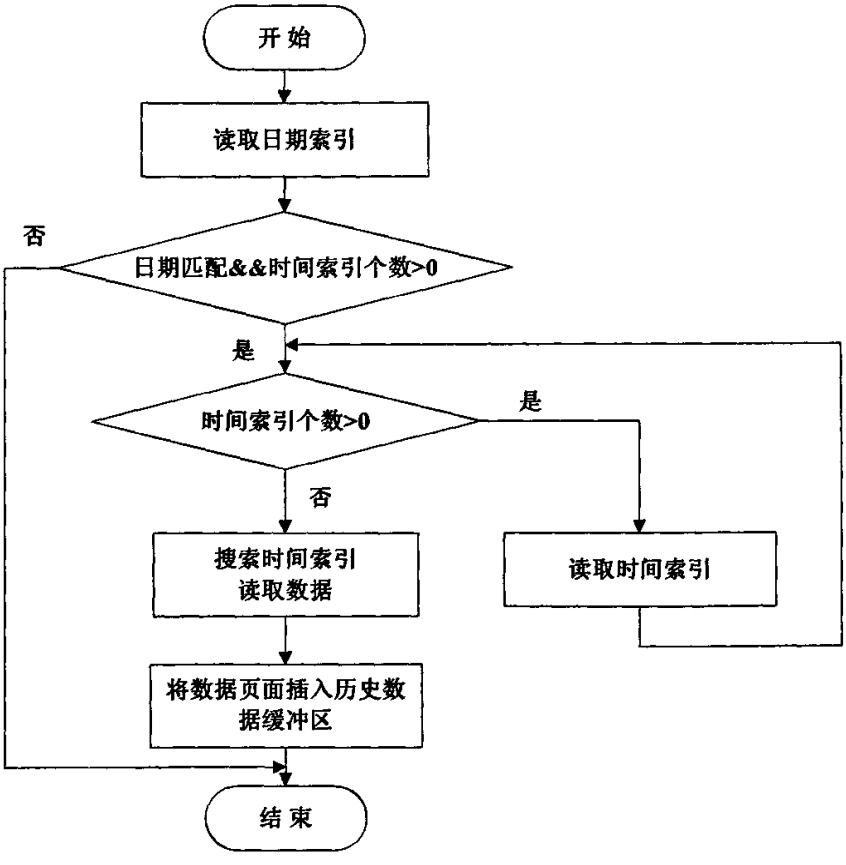


图 3-8 查询流程图

查询操作通过函数 Query 实现，需要四个参数：DWORD p_nLabelIndex：标签点号，BYTE nParaThisYear：年份，INT nParaDayOfYear：年内天数，DATE_INDEX *& v_pIndexBufNode：返回的数据。关键的算法如下：

//初始化日期索引节点

```
while (-1 == GetDateNode(pQueryDateIndex));
pQueryDateIndex->timeIndexCount = 0;
// 直接定位到指定的日期索引节点位置
offset =
p_nLabelIndex*366*LENGTH_DATE+(nParaDayOfYear-1)*LENGTH_DATE;
fOk = _ReadFile(m_fDateIndex[nParaThisYear], offset, t_pbBuffer,
    LENGTH_DATE);
if (!fOk)
{// 读取日期索引失败，释放空间返回错误
    g_IndexBufAVL.Release(pQueryDateIndex);
    return err;   }
// 获得日期索引节点
*pQueryDateIndex = *((DATE_INDEX *)t_pbBuffer);
//日期不匹配或无时间索引节点,这一天没有数据。
if ((pQueryDateIndex->date[0] != nParaThisYear)
     || (0 == pQueryDateIndex->timeIndexCount))
    {   g_IndexBufAVL.Release(pQueryDateIndex);
        return err;   }
```

获得日期索引节点后，就可读取这一天的时间索引，但要区分两种情况：一是时间索引在一个时间索引节点内，则一次就可读出；如果在多个节点内，则需多次读取，并将其连接起来：

```
//保留原来时间索引个数值，下面还会用到。
nTempCount = pQueryDateIndex->timeIndexCount;
_ReadFile(m_fTimeIndex[pQueryDateIndex->date[0]]
pQueryDateIndex->fileName], pQueryDateIndex->location,   t_pbBuffer,
    pQueryDateIndex->offset);
//指向时间索引块末尾并去掉下一节点的位置。
t_pbBuffer = t_pbBuffer +   (pQueryDateIndex->offset - 5);
nTempFileName = *t_pbBuffer;
t_nLocation = *((DWORD *)(t_pbBuffer+1));
nTempCount = nTempCount – (pQueryDateIndex->offset - 5)/LENGTH_TIME;
// 读出的第一块的包数只取实际的包数。
```

```
t_pstBufNode->nCount = (pQueryDateIndex->offset-5)/LENGTH_TIME;
while (nTempCount > (BLOCK_LENGTH-5)/LENGTH_TIME)
{  // 超过一块的长度。
    while (-1 == GetDataBuf(t_pstTempBufNode));
    t_pstCurBufNode->pNextNode = t_pstTempBufNode;
    t_pstCurBufNode = t_pstCurBufNode->pNextNode;
    t_pbBuffer = t_pstCurBufNode->lpBuffer;
    //继续读出一天剩下的时间索引信息。
    _ReadFile(m_fTimeIndex[pQueryDateIndex->date[0]][nTempFileName],
        t_nLocation，t_pbBuffer, BLOCK_LENGTH);
    //指向时间索引块末尾并去掉下一节点的位置。
    t_pbBuffer = t_pbBuffer + (BLOCK_LENGTH - 5);
    nTempFileName = *t_pbBuffer;
    t_nLocation = *((DWORD *)(t_pbBuffer+1));
    t_pstCurBufNode->nCount = (BLOCK_LENGTH-5)/LENGTH_TIME;
    nTempCount = nTempCount - t_pstCurBufNode->nCount;}
最后，取得返回数据。
ret = __GetData(pQueryDateIndex, t_pstBufNode);
if (err == ret)
{  g_DataBufAVL.Release(t_pstBufNode);
    g_IndexBufAVL.Release(pQueryDateIndex);  }
else {// 将临时节点(时间索引块)连接到日期索引节点中。
    pQueryDateIndex->pBufNode = t_pstBufNode;}
```

## 3.4 性能分析

### 3.4.1 页面大小的确定

为了确定合适的页面大小，建立以下的实验环境：以写入一个标签点一天的全部数据为例，在 P41.7G，512M 内存的电脑上进行实验，一秒钟产生一条数据，则共有 85600 条记录，取页面大小分别为单个记录一页、4K、16K 和 64K，标签点总数分别为 1000，5000 和 10000，所得的写入时间如表 3-3 所示：

表3-3 不同页面的写入执行时间

| 时间标签点 \ 页面 | 单个记录 | 4KB | 16KB | 64KB |
|---|---|---|---|---|
| 1000 | 7900ms | 175ms | 165ms | 150ms |
| 5000 | 8700ms | 180ms | 170ms | 160ms |
| 10000 | 9100ms | 185ms | 170ms | 163ms |

由上表可看出，本系统的写入效率不会随着标签点的增加而有明显的增加，有微小的差别只与内存的使用量有关。前已述及，读写磁盘是以块为单位，如果是单条记录组成一个块，其所用时间会明显增加，因此，应将数据组合成块。当块增大到一定程度时，如 4KB，写入时间不再随着页面大小的增加而有明显的下降，考虑到磁盘空间的使用，页面大小确定为4KB是合适的。

### 3.4.2 查询效率分析

由于标签点索引文件的建立是为了备份和恢复的目的，查询时不用读取标签点索引文件，其所用的结构常驻内存中。当查询一个标签点一年范围内的数据时，无论多少天的数据，由上述，读一次磁盘就可确定所有的日期索引信息。为便于分析，假设共需查询n天的数据，每一天用(d1，d2.....dn)表示，一条数据记录的长度为L，一条时间索引的长度LT=15B，一天的存储的数据条数为RN，每一天的数据记录条数用(r1，r2.....rn)表示，一天的时间索引记录个数为TN，每一天的时间索引记录个数用(t1，t2.....tn)表示，一天的时间索引记录占用的磁盘块个数为BTN，每一天的时间索引记录占用的磁盘块个数用(bt1，bt2.....btn)表示。磁盘块大小为B(=4K)，每天的数据记录所用的磁盘块个数即时间索引记录个数为$t_i=\lceil r_i * L/B \rceil$，每天的时间索引记录占用的磁盘块个数为$bt_i=\lceil t_i * LT/B \rceil$。则查询所要的总的读磁盘的次数DN为：

$$DN=1+\sum_{i=1}^{n} bt_i+\sum_{i=1}^{n} t_i$$

如读取一个标签点一天的数据，数据类型是整型，一条数据记录的长度L=12，假设一秒种产生一条数据，则一天共有r1=85600条数据记录。$t1=\lceil r_1 * L/B \rceil=\lceil 85600*12/4096 \rceil=251$，$bt1=\lceil t_1 * LT/B \rceil=\lceil 251*15/4096 \rceil=1$，则总的

读盘次数为：

DN = 1+1+251=253。

以上是极端情况下才会发生。首先，当数据存储时都会经过一定的压缩，存储量会减少很多。其次，在实际应用中，多数标签点的块都可能是在一个单一区里，这样一次就可读取 64KB 的数据，也就是 16 块的数据。则：

DN = 1+1+16 = 18。

## 3.5 小结

本章以磁盘历史数据库的开发为背景，从存储策略，存储结构，数据组织方式等方面综合研究，为从根本上提高存储频率和数据存储点数，分析了磁盘历史数据库设计中的问题，提出了一种新的磁盘存储结构是——Z 树结构，减少了磁盘 I/O 操作次数，极大地提高了磁盘存储性能。结合 Windows 操作系统的文件操作，采用文件组的方式，设计了一个高效的文件集管理系统。最终完成了磁盘了历史数据库的设计和实现，并对性能做了分析。

# 第四章 GDREAL 内存数据库

由于实时控制和先进控制需要近期的和实时的数据，对查询的实时性有很高的要求。但磁盘历史数据库因为要进行 I/O 操作，无法满足需求。因此，将实时数据和近期的历史数据存放在内存中，满足快速查询和存取要求。

在此，内存数据库是一个广义的名词。一方面，它将实时数据和近期的历史数据存放在内存中，一是为了满足快速查询和存取要求，从狭义上说，只有这一功能才称得上是实时内存数据库，二是为了减少写盘 I/O 次数，把已经成为历史数据的实时数据填满缓冲区后，一次性写入磁盘，此功能作为写入磁盘前的一个缓冲作用，第三章中，称此部分为历史数据缓冲池，在本章，统一称为实时内存数据库；另一方面，在从磁盘读取数据后，也应对数据进行一定的缓存，以备后面的查询使用，减少读盘次数，这实际上就是历史数据缓冲区。

## 4.1 内存数据库概况

内存数据库是支持实时事务的最佳技术，其本质特征是其“主拷贝”或“工作版本”常驻内存，即活动事务只与实时内存数据库的内存拷贝打交道。显然，它要求较大的内存量，但并不要求任何时刻整个数据库都能存放在内存，即内存数据库系统还是要处理 I/O。虽然如此，但它已不是传统磁盘数据库的概念，传统数据库适用的数据结构、事务处理机制及恢复等技术对内存数据库不一定合适。所以，实时内存数据库的设计应该考虑内存直接快速存取的特点，以 CPU 和内存空间的有效利用为目标来重新设计开发各种策略与算法、技术、方法及机制[16]。

内存数据库管理方式同磁盘数据库管理方式有显著的不同[17]：

a)内存和磁盘在存取时间上有若干数量级的差别。内存的存取时间在 10E-8s 的数量级，而磁盘在 5*10E-3 数量级；

b)内存数据具有易失性，关于故障和备份需要着重加以考虑；

c)数据存储格式不同。内存是字节或字编址的，而磁盘是块存储设备；

d)数据的存储组织方法对性能影响不同。不同的组织方式对磁盘而言的性能影响远比对内存影响大，如顺序存取与随机存取的时间对内存没有什么变化，而对磁盘则几乎有数量级的差别；

e)存取方式不同。内存可由处理机直接存取，磁盘则不能；但内存比磁盘更易于受到来自程序错误的直接数据破坏。

目前数据库技术中的数据索引结构主要有数组、AVL 树、B-树、B+树。在 IBM 的内存数据库系统 OBE 中，采用数组作索引结构。通过预知大小，或者通过像虚拟存储映射这样的技术而使其能妥当增长，使数组索引使用的空间最小。它的缺点是不能动态维护，每一维护操作所引起的数据移动量是 O(N)。

AT&T Bell 试验室的“硅数据库机”，以 AVL 树作为内存索引的结构。AVL 树操作的时间复杂度为 $O(\log_2 N)$，具有较高的存取性能。但其突出的缺点是其内存的有效利用率很低，每个节点只有一个数据元素，却有两个指针和有关的控制信息[18]。

B-树的操作性能好，且能动态维护。但它的内存利用率较低。在内存情况下，B+-树不具有比 B-树更大的优越性。

Hash 表是当前内存数据组织常用的方式。它是一种基于泊松分布统计计算的索引方法。具有组织方便，访问性能高的特点[19]。

对各种数据应采用不同的方式，才能发挥各种算法的长处，提高系统的整体性能。

## 4.2 数据存储流程

为保证系统对海量数据管理的支持，在设计实时数据库的数据流程时，采取如下措施：

(1)当接口机上有新数据来时，这个新值并不直接传到服务器中，系统把该值存入一级缓冲中，然后接收下一条数据，当一级缓冲填满后，再一次性通过网络传输至服务器。一级缓冲设在接口机上。见图 4-1。

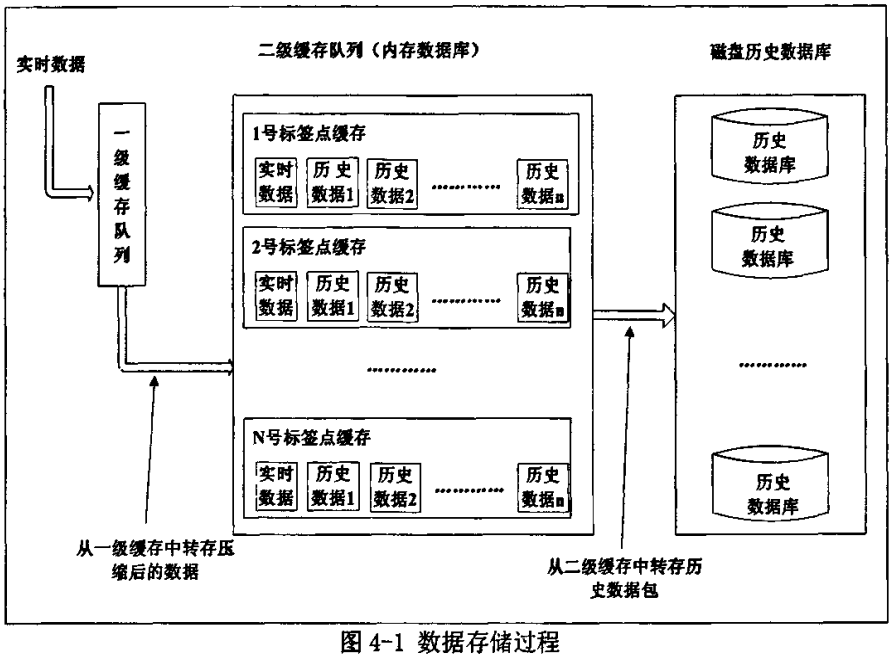


图 4-1 数据存储过程

(2)设立二级缓存。而二级缓存是为了防止频繁的文件写入，提高数据存储和查询的效率。系统为每个标签点设立一个缓存区，存储线程从一级缓存中取得数据，并不直接写入文件，而是写入该标签点对应的缓存区，只有当缓存区已满的时候，系统才会一次性地把缓存区中的所有数据写入数据文件。二级缓存就是实时内存数据库。在二级缓存中，把最新的一条数据作为实时数据，其余的都是近期的历史数据。

## 4.3 GDREAL 实时内存数据库的结构

在 GDREAL 实时数据库系统中，我们分别采用了数组、B-树和 Hash 表等数据结构，将三者的优点很好的结合起来。如在将标签点名称映射为标签点编号时采用了 Hash 表，在组织标签点节点时采用了数组结构，对日期索引节点和时间索引节点及数据页面采用了 B-树结构。

### 4.3.1 GDREAL 实时内存数据库的结构组织

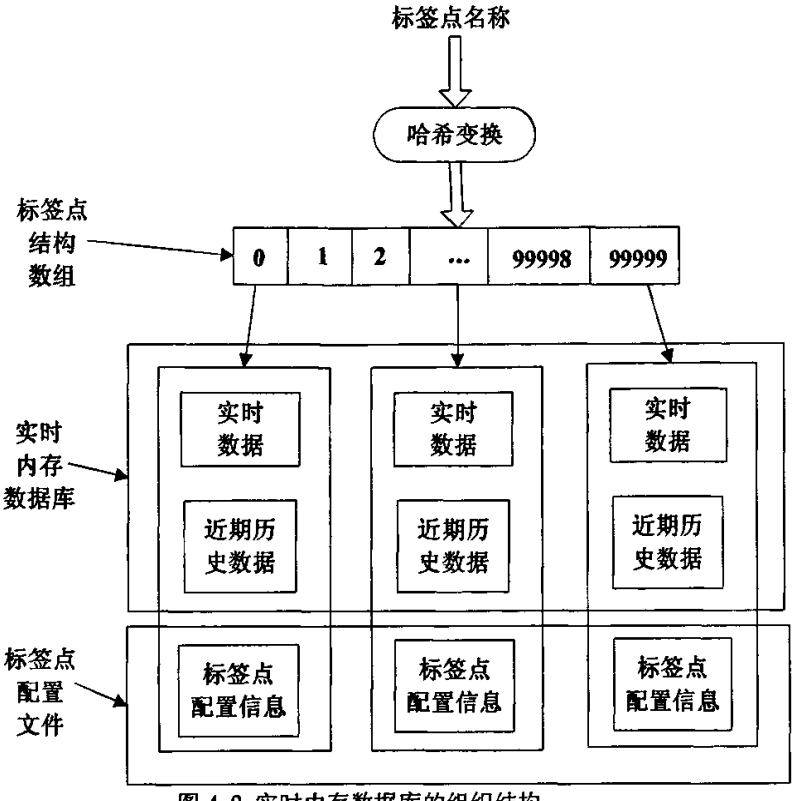


图 4-2 实时内存数据库的组织结构

图 4-2 是实时内存数据库的组织结构图。从中可以看出，当外部应用通过接口访问数据库的时候，通过哈希变换，将标签点名称映射为唯一的标签点编号，所有在数据库内部的访问都是以标签点编号来进行；当从数据库返回时，再将标签点号转换成标签点名称。

而标签点号在数据库内部以结构数组的形式来组织，按照事先的定义，标签点的个数最多可支持 100000 个，但可以根据服务器的性能动态调整。将标签点号作为标签点数组的下标，就可找到到标签点结构，从而实现对数据的快速访问。

每一标签点有三种类型的数据：实时数据、近期历史数据和标签点配置信息。其中实时数据和近期历史数据依次连续存放在标签点内存页面中，参见图 4-1，二者的形式并无不同，只是将最新的一条数据记录做为实时数据。数据记录此时由原始的四元组 d =（label,，time，state，data）变成一个三元组：

d =（time，state，data）

标签点号已经提取出来做为一级索引，日期也提取出来做为二级索引，数据记录中就不用再重复记录，节省存储空间和查询时间。在这三元组中，Time 表示

数据采集时间，以一天内的毫秒数表示；state 表示数据采集时的状态，包括数据质量或者其它数据点状态信息；data 表示数据值，由不同的数据类型决定其存储空间的大小，前两项的存储空间都是固定不变的。当标签点内存页面由最新的数据填满以后，将其一次性地写入磁盘页面中。因此，这一功能就是写入磁盘前的缓冲作用。通常采用双机热备份的方法来处理故障状态下缓冲区数据的安全问题。用户访问历史数据时，系统首先检查实时内存数据库中是否有用户需要的历史数据，如果没有再查询磁盘历史数据库。

标签点配置信息常驻内存，在标签点配置文件中存放备份，包括标签点名称、标签点时间标签、标签点描述、数据源地址、数据点工程单位、数据类型、扫描周期、量程范围、数据时间标志、数据值、数据状态(含数据质量或者其它数据点状态信息)等。

### 4.3.2 GDREAL 实时内存数据库的数据结构

为实现以上存储结构，将标签点索引结构进行扩充：

```
struct LABEL_INDEX
{  ......
   DWORD memLastTime;          // 实时数据时间
   int memLastState;           // 实时数据状态
   BYTE memLastType;           // 实时数据类型
   union VALUE memLastValue;   // 实时数据值
   WORD memRecordCount;   // 实时内存数据库中实时数据记录条数
   WORD memCurrentPosition;          // 实时内存数据库当前页面位置
   struct DATE_INDEX * dateIndex;          // 日期索引
};
```

各字段的涵义已在注释中标出，其中 memCurrentPosition 记录了写入实时数据的页面位置，dateIndex 表示日期索引节点，其类型 DATE_INDEX 结构的部分关键字段定义如下：

```
struct DATE_INDEX
{  ......
   BYTE date[3];                                  // 日期。年月日各一个字节。
   struct HDB_BUF_NODE * pBufNode;          // 实时数据页面
```

```
};
```

日期索引节点只保留当天的数据，也就是说，在当前的实时内存数据库中的数据记录是一天范围内的数据。pBufNode 是数据页面，此处用以存放实时数据和近期的历史数据。其结构的完整定义如下：

```
struct HDB_BUF_NODE
{
    struct TIME_INDEX timeIndex;        // 时间索引节点
    BYTE bFlag;                         // 页面标志
    WORD nCount;                        // 数据记录个数
    BYTE lpBuffer[PAGE_SIZE];           // 数据页面
    struct HDB_BUF_NODE * pPrevNode;    // 上一页面的位置
    struct HDB_BUF_NODE * pNextNode;    // 下一页面的位置
};
```

PAGE_SIZE 是页面大小，根据 Windows 操作系统的内存分页系统的页面大小以及磁盘页面大小的设置(参见 3.4.1 节)，将其大小设置为 4KB。timeIndex 是时间索引节点，用以表示当前实时内存数据库中的数据记录的起止时间。其结构的部分关键字段定义如下：

```
struct TIME_INDEX
{
    DWORD startTime;            // 页面起始时间(一天内的毫秒数)
    DWORD endTime;              // 页面结束时间(一天内的毫秒数)
    WORD length;                // 数据页面的实际长度
    ......
};
```

在系统初始化时，按照系统设计的最大量，初始化标签点索引节点数组，一次性分配其存储空间，以方便管理：

```
m_pLabelIndex = new LABEL_INDEX[LABEL_COUNT];
```

其中 LABEL_COUNT 就是系统能支持的最大标签点数目。而对日期索引节点和页面节点的分配，考虑到系统并不总是满负荷的运行，因此是在系统运行期间，当有数据来时，再分配空间。并且当删除标签点时，应释放所占用的空间。

## 4.4 历史数据缓冲区

### 4.4.1 关系数据库的缓冲池

对于关系数据库来说，一个主要的设计目标就是尽量减少磁盘 I/O，因为磁盘的读写操作是最慢的的操作之一。关系数据库一般在内存中生成一个缓冲池来存放从数据库读取的页，需要大量代码专门用于尽量减少磁盘与缓冲池之间的物理读写次数，并设法在以下两个目标之间达到平衡：

防止缓冲池变得过大，从而导致整个系统内存不足；尽量增加缓冲池的大小，以便尽量减少数据库文件的物理 I/O。

由此可见，关系数据库会占有系统尽可能多的内存空间用作缓冲池，另一方面，当系统的内存使用量过大时，会动态调整内存的使用量。

### 4.4.2 磁盘历史数据库查询缓冲区

在实时数据库中，实时事务要求系统能比较准确地估计和控制或规定事务的运行时间，但对磁盘数据库而言，由于磁盘存取、内外存数据传递、缓冲区管理、排队等待等使得实时事务实际平均执行时间不易控制在一个较短的时间内[20][21]。而如果将选定数据库的“主拷贝”或“工作版本”常驻内存，形成内存数据库，即活动事务只与实时数据库的内存拷贝打交道，每个事务在执行过程中实现 I/O。那么系统就可以将事务的运行时间控制在一个较短的时间内[22]。

为了减少磁盘 I/O 次数，有效利用数据复用功能，借鉴关系数据库和实时内存数据库的做法，在磁盘历史数据库中也设置了历史数据缓冲区[23]，使每一次查询后的数据的大部分甚至全部都保留在内存。在下次查询同样的数据时，就可直接从历史数据缓冲区中读取，不用再进行磁盘 I/O 操作，提高了查询速度。为此，我们采用了 AVL 树来组织数据，其搜索时间最短，时间复杂度为 $O(\log_2 N)$，N 为 AVL 树节点个数。

#### 4.4.2.1 查询缓冲区的结构

由于分为两级索引：日期索引和时间索引，时间索引是日期索引下的二级索引，AVL 树也有两级：日期索引节点的 AVL 树(D-AVL)和时间索引节点的 AVL 树(T-AVL)，每一个 D-AVL 树的节点都是一棵 T-AVL 树的根节点，而每一个标签点都包含一个日期索引节点的 AVL 树。如图 4-3 所示：

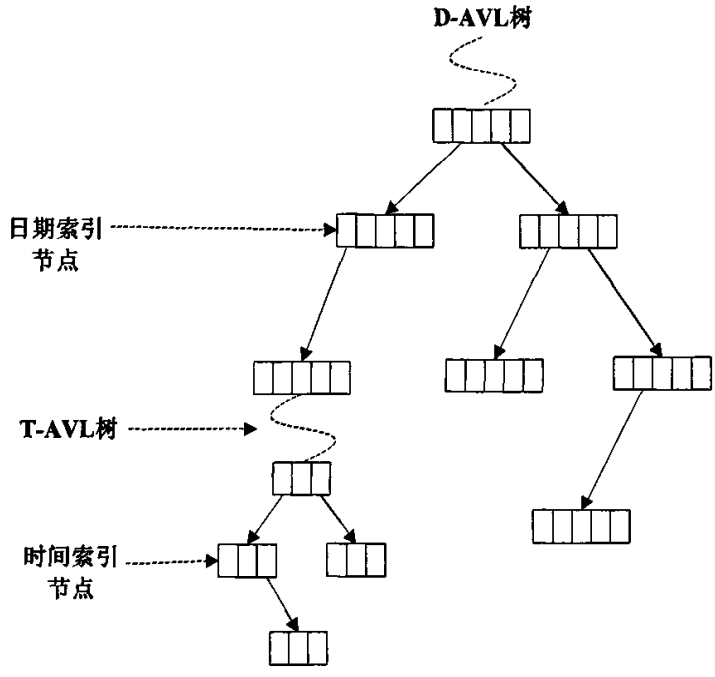


图 4-3 查询缓冲区两级 AVL 树结构

根据以上的分析，可扩展以下的数据结构：

```
struct LABEL_INDEX
{  ……
   AVL<struct DATE_INDEX> *pDateIndexAVL;  // 日期索引节点的 AVL 树
};
```

在标签点索引结构中增加一项 pDateIndexAVL，指向日期索引节点的 D-AVL 树的根节点。 AVL 是一模板类，封装了对 AVL 树的各种操作。设计成模板，可以为各种不同的数据类型提供访问，减少了编写代码的工作量。D-AVL 树中的每个节点都是一个日期索引 DATE_INDEX 结构，扩展其定义如下：

```
struct DATE_INDEX
{  ……
   DWORD labelIndex;                    // 标签点号。
   BYTE date[3];                        // 日期。年月日各一个字节。
   AVL <struct HDB_BUF_NODE> *pTimeAVL;    //时间索引节点的 AVL 树
   struct DATE_INDEX *pLChild;          // D-AVL 树的左孩子
```

```
    struct DATE_INDEX *pRChild;        // D-AVL 树的右孩子
};
```

在日期索引结构中增加一项 pTimeAVL，指向时间索引节点的 T-AVL 树的根节点。在 T-AVL 树中的节点都是 HDB_BUF_NODE 结构类型，其完整的定义请参见 4.3.2。T-AVL 树的每一个节点包括了三个基本字段：数据域、左孩子和右孩子。

查询某个标签点的数据时，先搜索其 D-AVL 树，只有当数据不存在时，再从磁盘历史数据库中读取数据。

#### 4.4.2.2 基于双队列的页面淘汰算法

页面置换算法依据一定的规则，把在内存中的某个页面置换出去，从而在空出一个物理页面后，装入导致缺页失败的页面[24]。

在所有的页面置换算法中，optimal 页面置换算法是最优的算法[25]，因为它淘汰下次访问距当前最远的那些页中序号最小的一页。OPT 算法的页面失败次数在所有页面置换算法中是最低的。然而 OPT 算法需要预先知道页面访问序列，这在真实环境下是不可能实现的，但 OPT 算法的意义是可以做为标准来衡量其它页面置换算法的优劣。

FIFO(First In First Out 的简写，先进先出)页面置换算法把最先进入内存中的页面置换出去，即选择淘汰在内存中驻留时间最久的页面。但有些在内存中停留时间最长的页往往经常被访问到，马上被替换出去后又会马上访问到，导致进程发生“抖动”(Thrashing)。

LRU(Least Reference Used 的简写，最近最小使用)页面置换算法把在内存中过去最长时间未被用到的页面置换出去。程序运行时页面访问序列会表现出当时性[26]，即当前访问的页面在最近的页面访问序列中会被经常访问到。LRU 算法只需知道在内存中的页面最后一次访问的顺序，然后根据该信息把最久未被访问到的页面置换出去，很好地适应了页面访问序列的当时性。在真实的环境下，LRU 算法要远远优于 FIFO 算法[27]。

在实时数据库系统中，有些标签点可能很重要，需要经常被访问到，为了避免产生抖动，我们采用 LRU 算法来淘汰过时的页面。LRU 算法要实现，必须解决两个问题：1) 内存中的各个页面各有多久时间未被进程访问过；2) 如何快速地知道哪一页是最近最久未使用的页面[29]。

对于操作系统来说，实现 LRU 算法需要硬件的支持，包括寄存器和栈[28]。如

果用软件来实现，这种方式却变得效率低下，因为每次要对寄存器置位和对所有页面进行比较运算或者搜索栈中所有页面并进行压入和弹出操作。为此，提出了基于双队列的页面淘汰算法，其思想是：一个页面同时从属于两个队列，一个是正在使用的页面队列(由应用决定，不一定是队列，如在本系统中就是一个 AVL 树的节点)，一个是管理页面的 LRU 队列。如图 4-4 所示。

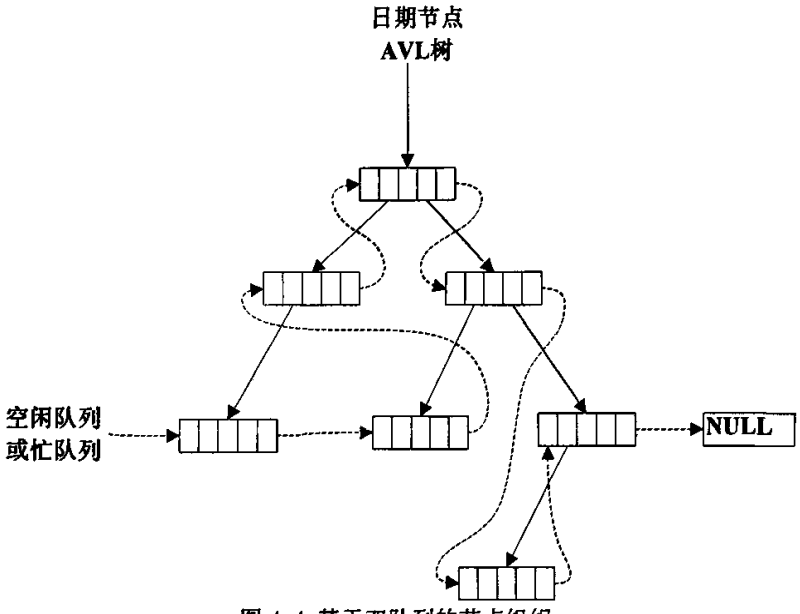


图 4-4 基于双队列的节点组织

其中 LRU 管理队列分为两种：空闲队列或忙队列。一个节点要么属于空闲队列，要么属于忙队列。当已无空的页面时，从空闲队列头部取出一个节点，同时在 AVL 树中删除相应的节点，释放到内存池中，再由内存池将页面分配给申请页面的用户。若此时在空闲队列中已无节点，则表示所有的节点都在忙队列中，因此，应从忙队列头部取出一个节点，删除相应的 AVL 树中的节点，放回内存池中，此时表示系统非常繁忙。淘汰节点时，是以日期索引节点为单位，而不是以内存页面为单位。从图 4-2 中可看出，日期索引节点有 5 个域，而数据页面只有 3 个域，日期索引节点中多出的两个域用以组织 LRU 队列。因此，在日期索引结构中，要增加以下的字段，以管理 AVL 树和 LRU 队列：

```
struct DATE_INDEX
{  ……
   BYTE nReferencedCount;        /节点的引用计数。
```

```
    struct DATE_INDEX *pLChild;        //左孩子
    struct DATE_INDEX *pRChild         // 右孩子
    struct DATE_INDEX *pLRUNextNode;       // LRU下一个日期索引节点。
    struct DATE_INDEX *pLRUPrevNode;       // LRU上一个日期索引节点。
};
```

nReferencedCount 表示查询时本节点的引用个数，如果其值大于等于 0，则表示此节点在空闲队列中，否则表示在忙队列中。表面上看，日期索引结构中有 4 个指针，存在着浪费，但由于一个日期索引节点下包含若干个页面，其所占用的空间非常小，由此换来的性能上的提升。

#### 4.4.2.3 断点续传

有时，用户查询的数据量会特别大，如在电厂制作的年报表中，其查询的数据的时间跨度很长，数据可能达到数万条。在这种情况下，如果一次性将所有的数据从磁盘读取到历史数据查询缓冲区中，可能会出现两种情况，一是很快将内存资源耗尽，最终没有足够的内存页面供使用，查询失败；二是即使查询成功，用户在客户端会等待很长时间，降低了实时响应时间和服务器的性能。因此，应分批返回数据，返回的数据条数为一固定值 ReturnedRecordNum。

有三种方式实现分批返回：

(1) 每一次查询都执行 I/O 请求，从磁盘查询数据，返回指定数量的数据。这是效率最低的一种。磁盘历史数据库设计的目标就是要尽可能减少磁盘 I/O，而这种方式与设计初衷背道而驰。我们采用磁盘历史缓冲区来减少磁盘 I/O；

(2) 一次从磁盘读取一天的数据到缓冲区中，并以 AVL 树组织起来。如果数量达到了返回条数 ReturnedRecordNum，则返回；否则继续查询下一天的数据，直到满足条件为止。而下一次则从磁盘历史缓冲区中查找所需数据，从上一次数据的返回点处返回数据。这种方式减少了磁盘 I/O，但由于每次都要搜索 AVL 树，因此，其效率也是不高的；

(3) 对第二种方法进行改进，记录上一次数据的返回点，下次只要根据记录从返回点的下一条记录返回数据即可，称这种方式为断点续传。无疑，采用了断点续传技术的查询方式效率是最高的，避免了重复的磁盘 I/O 和在内存中执行搜索操作。

为了使用断点续传技术，要在标签点索引结构中增加一项：

```
struct LABEL_INDEX
{  ……
   AVL<struct BROKEN_POINT> *pBrokenPointAVL;    // 断点 AVL 树
};
```

断点节点按照 AVL 树组织起来，pBrokenPointAVL 指向 AVL 树的根节点。断点 BROKEN_POINT 定义如下：

```
struct BROKEN_POINT
{
   long nSequenceNum;                    // 查询序列号。
   int nStartIndex;                  // 数据缓冲区的起始位置。
   struct DATE_INDEX * pDateIndexNode; // 日期索引节点。
   struct HDB_BUF_NODE * pBufNode;    // 数据缓冲区节点。
   struct BROKEN_POINT *pLChild;   // 左孩子
   struct BROKEN_POINT *pRChild;   // 右孩子
   struct BROKEN_POINT *pLRUPrevNode;        //LRU 上一断点位置。
   struct BROKEN_POINT *pLRUNextNode;        //LRU 下一断点位置。
};
```

断点也是一种内存资源，因此也要用 LRU 队列进行管理。系统为每一个查询分配一个全局唯一的查询序列号，每一次调用查询接口函数时通过参数传入，初始查询时设为-1，每一次查询分批返回时将查询序列号通过值参返回给调用者，下一次查询时直接将上次分配的查询序列号传入，与本标签点的断点中的 nSequenceNum 字段进行比较，验证断点是否存在。其算法如图 4-5 所示：

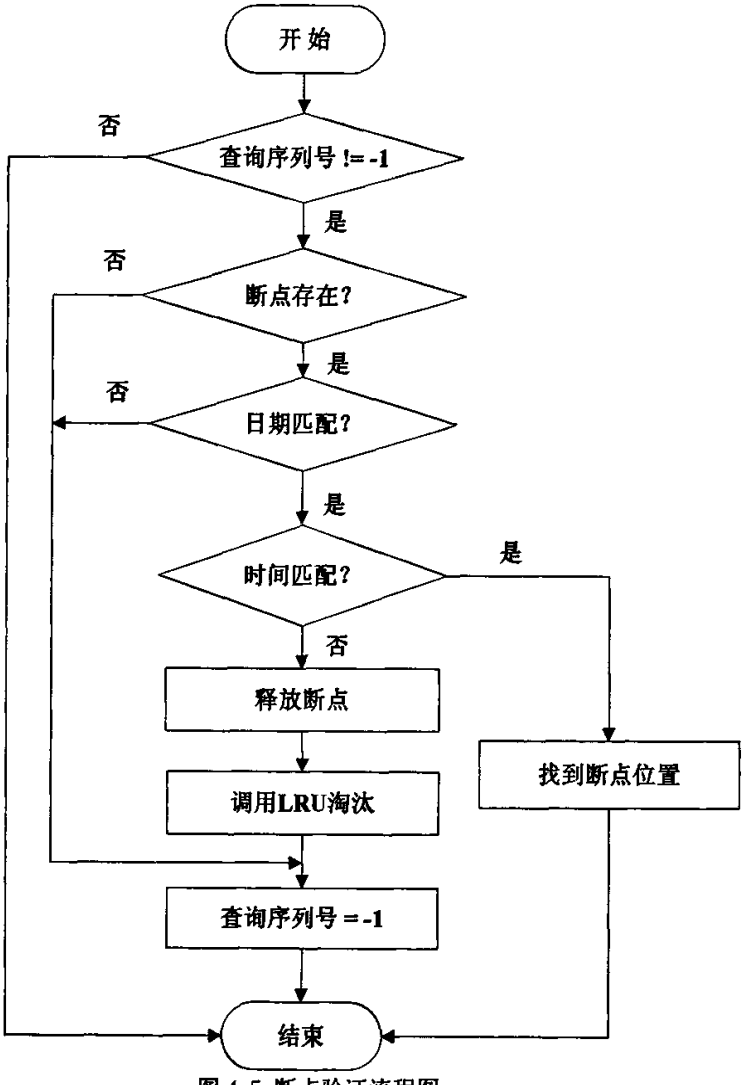


图 4-5 断点验证流程图

首先判断查询顺序号是否为初始值-1，若不是-1，则搜索本标签点的断点 AVL 树，看是否有节点的 nSequenceNum 与之相等，若相等，则表示查询可能存在，需进一步验证日期和时间是否匹配；否则将查询序列号置为-1，结束验证。若最后的时间不匹配，则还要释放断点并调用 LRU 页面淘汰函数。当找到正确的断点后，就可从断点位置继续返回数据，省去了每次重复的磁盘 I/O 和在内存中执行搜索操作，极大地提高了系统性能。以下代码验证断点的有效性：

```
// 找到顺序号所指向的数据位置。
t_pstBrokenPoint =
```

```
m_pLabelIndex[p_nLabelIndex].BrokenPointList.GetNode(v_nSequenceNum);
if (t_pstBrokenPoint != NULL)
{  // 找到顺序号,验证顺序号的有效性,初始化起始位置。
   t_pstQueryDateIndex = t_pstBrokenPoint->pDateIndexNode;
   if (p_stStartTime.time==t_pstQueryDateIndex->date)
   {  // 日期匹配，验证时间
      if (*((DWORD *)(t_pbBuffer+t_nStart*t_nRecordLength))
         == t_nStartMillisecondOfDay)
        // 断点有效
         return true;
      else
      {  // 日期节点存在，但断点无效，减少引用数
         t_pstQueryDateIndex->nReferencedCount--;
         if (t_pstQueryDateIndex->nReferencedCount <= 0)
         {  //释放断点
            __ReleaseBP(t_pstBrokenPoint, p_nLabelIndex);
            // 将日期节点从忙队列移到空闲队列。
            g_BusyIndexList.RemoveNode(t_pstQueryDateIndex);
            g_IdleIndexList.AppendNode(t_pstQueryDateIndex);} }}
   // 未找到断点
   v_nSequenceNum = -1;
   return false;}
```

断点的位置由 pDateIndexNode、nStartIndex 和 pBufNode 三者决定。在设置断点时需要记录这三个值。首先要处理序列号，只有设置断点时才需要序列号。

```
if (v_nSequenceNum == -1 || p_pstBrokenPoint == NULL)
{  // 获得全局唯一的序列号
   v_nSequenceNum = InterlockedIncrement(&g_nSequenceNum);
   if (v_nSequenceNum == -1)
      v_nSequenceNum = InterlockedIncrement(&g_nSequenceNum);
   p_pstBrokenPoint->nSequenceNum = v_nSequenceNum;
   // 加入到本标签点的断点 AVL 树中，同时加入到全局断点链表中
   m_pLabelIndex[p_nLabelIndex].BrokenPointAVL.
```

```
    AppendNode(p_pstBrokenPoint);
    g_BusyBPList.AppendNode(p_pstBrokenPoint); }
// 将断点处的时间重设为起始时间
t_nTempTime = *((DWORD *)(p_pCurBufNode->lpBuffer
    +p_nBrokenPoint*p_nRecordLength+PACKAGE_HEADER_LENGTH));
// 重设断点位置
p_pstBrokenPoint->pDateIndexNode = p_pstIndexBufNode;
p_pstBrokenPoint->pBufNode = p_pCurBufNode;
p_pstBrokenPoint->nStartIndex = p_nBrokenPoint;
```

## 4.5 GDREAL 内存数据库的实现

用户通过内存数据库访问实时数据和历史数据。为此，系统建立一内存数据库类 CHMemDatabase，其父类是磁盘历史数据库类 CHDiskDatabase，用户只需使用内存数据库提供的接口就可访问内存数据库和磁盘历史数据库，如果数据在内存数据库里，则直接访问；否则，向磁盘库发送缺页请求，将所需数据放入内存数据库后再进行访问。所需数据是在内存库中还是在磁盘库中，对用户是完全透明的。

### 4.5.1 追加实时数据

由接口机上传的数据包里包括不同标签点的数据，需要将其放入实时内存数据库中的不同标签点的缓冲中。当一个标签点的缓冲填满以后要将其写入磁盘历史数据库保存。图 4-6 是详细描述了追加实时数据的过程。

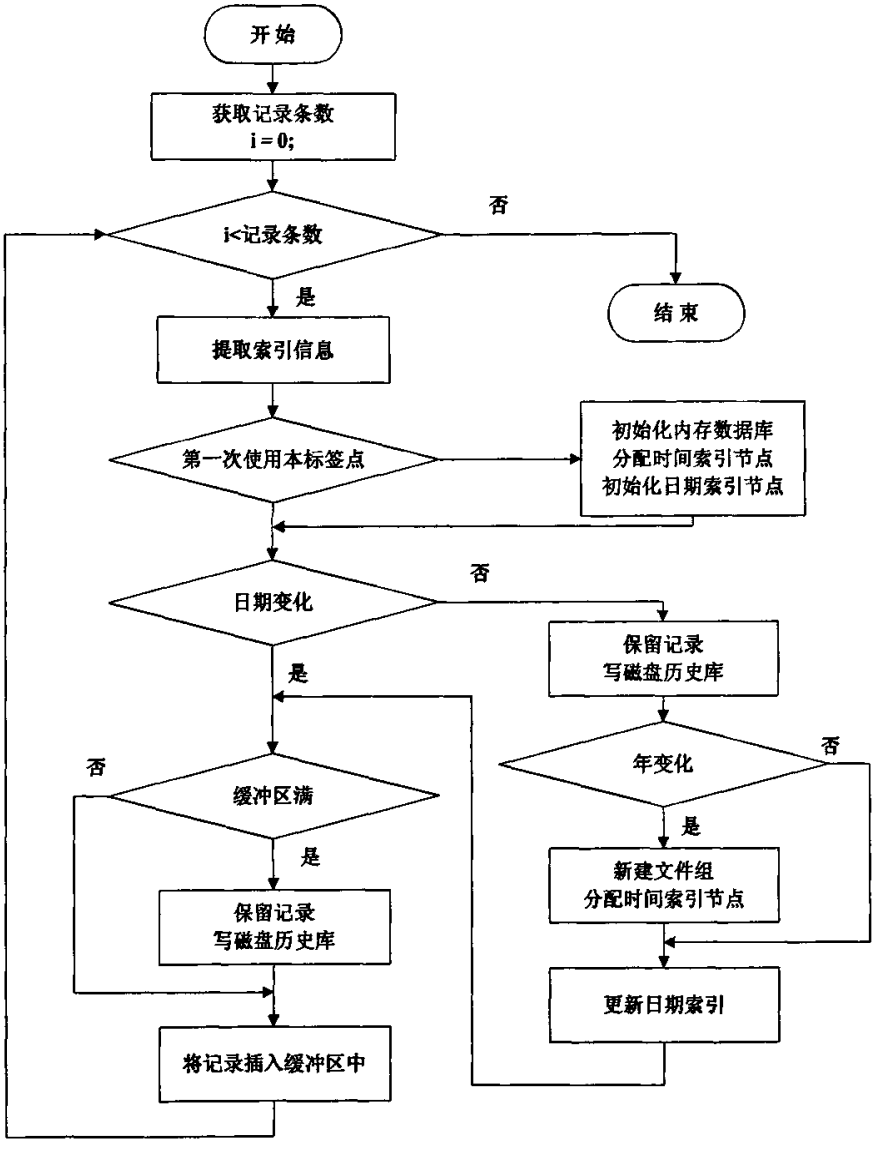


图 4-6 追加实时数据流程图

通过追加函数 Append(BYTE * p_pbBuffer)来实现以上算法：

首先获得数据记录个数：

```
t_nRecordCount = *((WORD *)p_pbBuffer);
p_pbBuffer += 2;
// 循环处理需要存储的数据包及其索引,以每条数据为循环单位。
for (WORD i = 0; i<t_nRecordCount; i++)
{   // 标签索引号。
```

```
t_nLabelIndex = *((DWORD *)p_pbBuffer);
p_pbBuffer = p_pbBuffer+4;
// 得到一天的毫秒数。
t_nMillisecondOfDay = __GetMillisecondOfDay(t_stTime.hms[0],
    t_stTime.hms[1], t_stTime.hms[2], t_stTime.millisecond);
// 第一次为标签索引结点新分配时间索引结点的块。
if((BYTE)-1 == m_pLabelIndex[t_nLabelIndex].firstYear)
{   // 初始化内存历史数据库。
    m_pLabelIndex[t_nLabelIndex].memRecordCount = 0;
    m_pLabelIndex[t_nLabelIndex].memCurrentPosition =
        PACKAGE_HEADER_LENGTH;
    // 分配新的时间索引结点。
    _AllocateTimeIndex(t_nLabelIndex);
    // 初始化日期索引结点。
    m_pLabelIndex[t_nLabelIndex].dateIndex->fileName
        = m_pLabelIndex[t_nLabelIndex].lastFileName;
    m_pLabelIndex[t_nLabelIndex].dateIndex->location
        = m_pLabelIndex[t_nLabelIndex].timeLastLocation;
    m_pLabelIndex[t_nLabelIndex].dateIndex->timeIndexCount = 0;
    m_pLabelIndex[t_nLabelIndex].dateIndex->offset
        = m_pLabelIndex[t_nLabelIndex].remindCount;}
```

检查是否是一个新的日期索引点即新的一天开始。要把前一天的日期索引结点的信息写入文件。如果年月日有任何一个发生了变化，都认为是这一天变化了。在其内部再判断是否是年发生了变化，做相应的处理。此处把年的处理放在了内部，因为二者有需要共同处理的部分。如果是年不同，则处理年变化后的情况。

```
if (m_pLabelIndex[t_nLabelIndex].dateIndex->date != t_stDate)
{   // 保留上一条数据
    __ReserveLastRecord(t_nLabelIndex, 4+t_nLength,
        COUNT_OF_MILLISECOND_OF_ONE_DAY);
    // 把缓冲区中的所有数据全部与入文件中。
    Write(t_nLabelIndex, m_pLabelIndex[t_nLabelIndex].memCurrentPosition,
        8+m_sznSizeOfType[ m_pLabelIndex[t_nLabelIndex].dateIndex
```

```
->pBufNode->lpBuffer[0]], 1);
```

新的一年开始，需重建新的日期索引文件。此处用以下判断而不能用“!=”，因有些数据跨年后，而另一些慢速标签点数据则可能晚于它到达。m_nCurrentYear > t_stDate.year 就属于这种情况，但由于以前的文件已存在，所以不用重建。直接通过年就能访问到。

```
if(m_pLabelIndex[t_nLabelIndex].currentYear+1 == t_stDate.ymd[0])
```

{ // 如果是第一个标签点的数据到了新的一年，但 currentYear == m_nCurrentYear，还要建立相应的新的一年的文件。但如果不是第一个标签点的数据，则表示新的一年的数据已经建立起来了，不用再建立了。

```
    if (m_pLabelIndex[t_nLabelIndex].currentYear == m_nCurrentYear)
    {   __CreateFile(m_nCurrentYear+1, 'y');
        //初始化时间索引结点位置。
        m_nLastLocation = 0;
        m_nLastDataLocation = 0;}
    //分配新的时间索引结点。
    _AllocateTimeIndex(t_nLabelIndex); }
// 更新日期索引结点信息。
m_pLabelIndex[t_nLabelIndex].dateIndex->timeIndexCount = 0;
// 新的一天的时间索引结点所在文件。
m_pLabelIndex[t_nLabelIndex].dateIndex->fileName
    = m_pLabelIndex[t_nLabelIndex].lastFileName;
// 新的一天的时间索引结点位置。
m_pLabelIndex[t_nLabelIndex].dateIndex->location
    = m_pLabelIndex[t_nLabelIndex].timeLastLocation;
// 新的一天的时间索引结点在当前时间索引结点块中的位置。
m_pLabelIndex[t_nLabelIndex].dateIndex->offset
    = m_pLabelIndex[t_nLabelIndex].remindCount;}
```

当前缓冲区已满不能够容纳本包的数据，需要将缓冲区数据写入磁盘历史数据库保存。

```
    if (t_nLength+4 >
(DATA_BUFFER_LENGTH-m_pLabelIndex[t_nLabelIndex].memCurrentPosition))
    {   //保留上一条数据
```

```
__ReserveLastRecord(t_nLabelIndex, 4+t_nLength);
// 把缓冲区中的所有数据全部与入文件中。
Write(t_nLabelIndex,
    m_pLabelIndex[t_nLabelIndex].memCurrentPosition,
    8+m_sznSizeOfType[ m_pLabelIndex[t_nLabelIndex].dateIndex
    ->pBufNode->lpBuffer[0]]);  }
```

最后，将一条记录插入到标签点的页面缓冲区中。

```
__AppendOneRecord(t_nLabelIndex, t_nMillisecondOfDay, p_pbBuffer,
    t_nLength);
```

## 4.5.2 查询数据

在查询数据时，系统会依次搜索实时内存库，历史内存库和磁盘历史库。如图 4-7 所示。

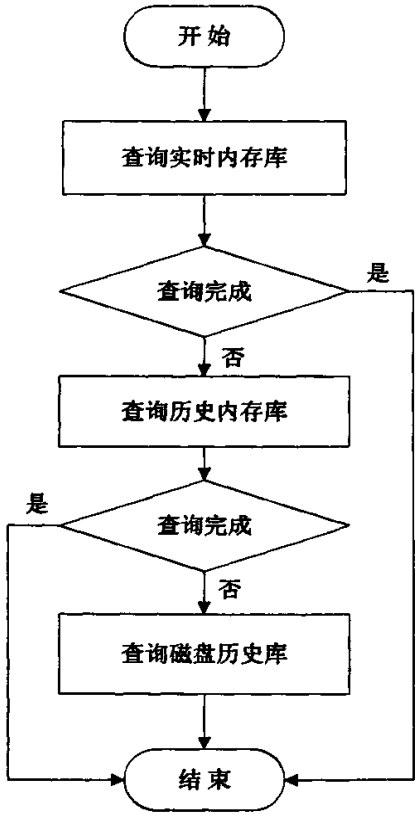


图 4-7 查询数据流程图

通过 Query 函数实现查询操作，其原型如下：

```
int Query(DWORD p_nLabelIndex,                    // 标签点号
        QUERY_TIME_INDEX & p_stStartTime,   // 起始时间
        QUERY_TIME_INDEX p_stEndTime,       // 结束时间
        BYTE * v_pbBuffer,                  // 数据缓存，由用户分配、释放
        int & v_nBufLength,                 // 缓冲区(v_pbBuffer)长度
        long & v_nSequenceNum);             // 查询序号,初始为-1
```

由于可能要查询的数据量特别大，需要多次调用查询函数，实现断点续传，用 v_nSequenceNum=-1 来标记这是第一次查询。以后再查询时不用改动任何参数，用户直接调用原来的函数继续查询下一位置处的数据就可实现多次查询。函数内部通过 v_nSequenceNum 的值判断断点的存在与否。通过返回值判断查询是否结束：0 表示所有查询都执行完毕，且返回正确的值；1 表示一天的数据执行完毕，且返回正确的值。但还有数据要继续查询；-1 表示出错，不再继续。

```
//是否是当前天，只有当前天才有内存历史数据库。
if (p_stStartTime.time == m_pLabelIndex[p_nLabelIndex].dateIndex->date)
{  // 是否在内存数据库中
   flag = __IsInMemory(p_nLabelIndex,      t_nStartMillisecondOfDay,
                t_nEndMillisecondOfDay);
   if (flag)
   {  // 从实时内存库中查询数据
      ret = __GetDataInMemory(p_nLabelIndex, t_nStartMillisecondOfDay,
          t_nEndMillisecondOfDay, v_pbBuffer,  t_pstBrokenPoint);  }}
// 是否在历史内存库中
DATE_INDEX * t_pstQueryDateIndex
      = m_pLabelIndex[p_nLabelIndex].DateIndexAVL.GetFirstNode();
if (v_nSequenceNum != -1)
{  // 不是第一次查询,验证断点是否存在
   __VerifyBP(v_nSequenceNum); }
if (v_nSequenceNum == -1)
{  //获取日期索引节点
   t_pstQueryDateIndex   =
m_pLabelIndex[p_nLabelIndex].DateIndexAVL.GetFirstNode(p_stStartTime);
```

```
if (t_pstQueryDateIndex != NULL)
{  // 是第一次查询。但在查询内存库中。需要根据 LRU 淘汰算法移动节点位置
    if (t_pstQueryDateIndex->nReferencedCount == 0)
    {  // 在空闲队列中，将其取出放到忙队列末尾
        g_IdleIndexList.RemoveNode(t_pstQueryDateIndex);
        g_BusyIndexList.AppendNode(t_pstQueryDateIndex); }
    else
    {  // 在忙队列中，将其取出放到忙队列末尾
        g_BusyIndexList.RemoveNode(t_pstQueryDateIndex);
        g_BusyIndexList.AppendNode(t_pstQueryDateIndex); }
    // 新查询要增加此日期索引节点的引用计数。
    t_pstQueryDateIndex->nReferencedCount++; }
else
{  // 不在查询内存库中，需重新从磁盘上查找所需数据。
    ret = _GetDataInDisk(p_nLabelIndex,  p_stStartTime.time[0],
            t_nStartDayOfYear,  t_pstIndexBufNode);
    if (ret == err) return err;
    else
    {  // 将数据结点加入到内存历史数据库链表中。
        // 初次查询的引用计数值为 1。
        t_pstIndexBufNode->nReferencedCount = 1;
        t_pstIndexBufNode->labelIndex = p_nLabelIndex;
        // 基于双队列的 LRU 淘汰算法。
        //将节点加入到日期节点的 AVL 树中
        m_pLabelIndex[p_nLabelIndex].DateIndexAVL.
            AppendNode(t_pstIndexBufNode);
        //将节点加入到忙队列中
        g_BusyIndexList.AppendNode(t_pstIndexBufNode);}} }
if (-1 == v_nSequenceNum)
{  // 通过二分法找到数据起始位置和结束位置
    t_nStart = __BinSearch (t_nRecordLength, // 一条记录长度
```

```
            t_pCurBufNode->nCount,              // 包里的数据条数
            t_nStartMillisecondOfDay,           // 起始时间
            t_pbBuffer);                        // 数据包
t_nEnd = __BinSearch (t_nRecordLength, t_pCurBufNode->nCount,
            t_nEndMillisecondOfDay, t_pbBuffer);
// 复制的数据长度
t_nFoundCount = t_nEnd-t_nStart+1;
t_nCopyDataCount = (t_nRemaindedCount <= t_nFoundCount)
            ? t_nRemaindedCount : t_nFoundCount;
// 断点位置
t_nBrokenPoint = t_nCopyDataCount+t_nStart;
// 有数据复制
memcpy(v_pbBuffer+ONE_PACKAGE_HEADER,
    t_pbBuffer+t_nStart*t_nRecordLength,
    t_ t_nCopyDataCount*t_nRecordLength);
t_nRemaindedCount -= t_nCopyDataCount;
if (t_nEndMillisecondOfDay < *((DWORD *)(t_pCurBufNode->pNextNode
    ->lpBuffer   +PACKAGE_HEADER_LENGTH)))
    // 所有数据处理完
        return 0;
else
{   // 设置断点后返回
    __SetBP(p_nLabelIndex, p_stStartTime,   t_nBrokenPoint,
        t_nRecordLength, t_pstIndexBufNode,  t_pstBrokenPoint,
            t_pCurBufNode, v_nSequenceNum);
    return 1; }
```

## 4.6 基于内存池的内存分配

在当今的实时监控系统中，由于可靠性和实时性的要求，几乎所有的实时数据库系统都是24*7*365的运行。从硬件上来说，大都采用了双机热备份的方式来保障系统的可靠性，即一个主服务器用作当前的工作服务器，一个备份服务器用

于当主服务器出现故障时，代替主服务器工作；从软件上来说，应不断提高软件的可靠性，保证系统不间断运行。其中最容易出现问题的就是对资源的使用，特别是对内存的使用。

为了适应实时数据库系统对实时性和可靠性的要求，应采用预先内存分配，即内存池，原理是：在内存中开辟一些内存区域(称为缓冲池或内存池)，当系统中有内存调度时，不需要向操作系统提出请求，而是向内存池提出内存块的申请和释放[5]。

由缓冲池来统一管理内存，一般仅在内存池无可用空间时才向操作系统提出一次内存申请，但这个申请不应是无限制的。当今的操作系统中，采用了虚拟存储器技术，对内存的分配已远远超越了物理内存的容量，从某方面来说扩大了应用的范围。但虚拟内存会降低系统的性能，延迟实时反应，因此在实时数据库系统中就避免使用虚拟存储器。物理内存的总的使用量不应超过总的物理内存空间的 80%，通过以下的代码可获得系统的当前内存状态并分配新的空间：

```
MEMORYSTATUSEX statex;
statex.dwLength = sizeof (statex);
GlobalMemoryStatusEx (&statex);
if (statex.dwMemoryLoad<80)
   { 申请新的内存块；}
```

使用内存池机制的优点主要有：

(1) 通过引入内存池机制，将动态堆内存的分配转化成为程序运行预先进行一次性堆分配的方式，这样当申请新的内存块的时候，就可以直接从预先分配好的池中选出一块可用的内存，这样就避免了动态临时分配所带来的开销。由于决大部分内存操作都通过内存池接口进行，有效地减少了对 OS 提出内存请求带来的系统调用开销（通常系统调用的开销都比较大），系统就可以在很短的时间内做出响应，这样内存分配和释放的性能就是高度可预测的。同时，对于内存池可以采用统一的管理，简化了动态存的管理，减少了内存泄漏发生的几率；

(2) 减少了内存碎片。动态内存分配和释放中，可能产生大量的内存碎片，最终可能的一种情况是：即使有够多的物理内存，但由于都是一个个的小碎片，要分配一个大的连续区域时，系统会提示内存不足，最终导致系统崩溃。采用预先不会使资源耗尽。为了减少内存碎片，要设计不同的池粒度，以适应不同大小的内存分配需要。虽然池的粒度不可能适应各种大小的粒度（从这个角度来说，对于大小可变的内存分配需求来说，池式分配也易于产生内存碎片，但是可以通过

动态的适应性和静态的最接近的粒度构造，来减少这种碎片的产生），但是采用池分配实际上还是将原来的直接动态内存分配所产生的“块间碎片”转化成了“块内碎片”，这实际上是一种对于内存碎片的分摊机制。由于实时内存数据库中的数据格式比较单一，大多数页面都是以 4K 为单位，因此，能够避免页内碎片的产生。

由此可见，采用池式分配机制有效地改善了系统的动态响应能力，提高了运行时刻的时间、空间性能，减少了运行时刻的资源开销。

为了能实现对不同的数据类型进行缓冲池管理，我们采用了模板类。

```
template <class T>
class CHDatabaseBuffer
{
    CHDatabaseBuffer(int num);
    ~CHDatabaseBuffer(void);
    // 从数据缓冲区链表中取出头节点。
    int Get(T *&p_pNode);
    //  将节点归还数据缓冲区缓冲区链表
    int Release(T *& p_pNode);
    //  增加一个缓冲区。
    int IncreBuf(int);
    T* pFirstNode;          // 链表头节点
    T* pLastNode;           // 链表尾节点
    T* iterator;            // 迭代器
    // 初始化时分配的缓存个数。
    int m_nBufCount;
};
```

将所有预先分配的内存页面以队列的形式组织起来，当有用户要申请内存空间时，不再进行内存的动态分配，而是通过 Get 函数一次取出一个页面，释放时通过 Release 函数将页面归还给队列，不再进行动态释放。当内存池中的所有页面都申请完并且还有足够的物理内存时，可通过 IncreBuf 函数动态增加新的页面，如果没有，则通过基于双队列的 LRU 淘汰算法从历史数据缓冲区中淘汰页面，重新放入内存池中供使用。

## 4.7 小结

在本章中，以实时内存数据库的开发为背景，分析了实时数据的存储流程，完成了实时内存数据库和历史数据缓冲区的设计和实现：为了避免内存碎片和减少动态内存分配和释放的开销，设计了一个内存池，改动态内存分配为预先分配，提高了系统性能；结合历史数据的特点，设计实现了基于双队列的内存页面 LRU 淘汰算法，将之用于历史数据缓存页面的管理；采用断点续传机制，很好地实现了大量数据的分批返回。

# 第五章 GDREAL 的事务和并发控制

实时历史数据库并发性用来解决多个用户对同一数据进行操作的问题。由于实时数据库是数据和事务都有定时特性或定时限制的数据库[31]，这个问题更加突出。提高数据库的处理速度，单单依靠提高计算机的物理速度是不够的，还必须充分考虑数据库的并发性问题，提高数据库并发性的效率。通过使用事务和锁机制，可以很好地解决实时数据库的并发性问题。

## 5.1 实时数据库中的事务和并发控制

实时数据库的事务同关系数据库的事务定义相同。一个或多个数据库操作组成一组，称作事务。事务是必须被作为一个原子、外观上孤立于其它事务执行的单位[32]。

实时数据库系统可以看作是常规数据库管理系统与实时系统的结合体，像 DBMS 一样，它必须处理事务并保证 ACID 数据特性[33]。ACID 即原子性、一致性、隔离性和持续性[34]。原子性是指事务必须是一个自动的单元工作，要么执行全部数据的修改，要么全部数据的修改都不执行。一致性是指当事务完成时，必须使所有数据都具有一致的状态。在实时数据库中，所有的规则必须应用到事务的修改上，以便维护所有数据的完整性。所有的内部数据结构，例如树状的索引与数据之间的链接，在事务结束之后，必须保证正确。隔离性是指并行事务的修改必须与其他并行事务的修改相互独立。一个事务看到的数据要么是另外一个事务修改这些事务之前的状态，要么是第二个事务已经修改完成的数据，但是这个事务不能看到正在修改的数据。这种特征也称为串行性。持久性是指当一个事务完成之后，它的影响永久性的产生在系统中，也就是这种修改写到了数据库中。

实时数据库系统中的事务与传统事务之间有较大区别，其事务可以有定时限制(典型的为截止期)，系统追求的目标不是系统的吞吐量，而是单个事务定时限制的满足，以使满足定时限制的事务比率最大；传统事务的原子性、一致性、隔离性及永久性在实时环境下变得太严格或不可能。

实时数据库中的事务并发运行和存取共享数据，彼此存在相互干扰，因此需要解决数据库的实时并发控制问题[35]。并发控制就是要控制并发事务之间的相互

作用，使数据库的一致性不被破坏。可串行化是传统数据库中并发控制最普遍的正确性原则，但在实时事务中会影响事务的定时性。在实时数据库中，可采用放宽可串行化。

实时数据库系统不能沿用关系数据库系统的并发控制协议，因为实时数据库中大量的事务具有时间限制，强调这些事务具有平等的地位和相同的被调度运行的机会。为了使尽可能多的事务满足它们的截止时间约束，在考虑调度策略时，应将事务的时间约束作为一个重要的因素，越紧急的事务，在不影响数据系统正确性的原则下，应较早地被调度运行。当一个事务正在运行时，如果新到了一个更为紧急的事务，该执行事务应该被抢占[36]。

迄今为止，已有的并发控制和事务调度策略要么假定实时数据库由一种单一事务形式如硬实时事务[37]组成，要么假定其由两种混合事务如硬实时/软实时事务[38]或软实时/非实时事务[39]组成，具有应用局限性。但在实时数据库中同时具有这三种类型的混合事务。因此，应采用一种混合并发控制模式，对不同事务类间冲突采用不同策略而对不同类事务的类内冲突采用同类策略[40]，保证：

(1) 保证硬实时事务的可调度性；

(2) 使违反软实时事务截止期的事务数达到最小；

(3) 使非实时事务有一个小的平均响应时间，不会使过多事务发生饿死现象。

因此，并发控制的核心是在保证硬实时事务完成的基础上尽可能提高系统的吞吐率。

目前在实时数据库理论界有两类并发控制协议：一种是基于锁的并发控制协议，主要有两段锁协议(经典的并发控制协议)、优先级继承机制处理并发控制协议；一种是乐观并发控制协议[41][42]。

两阶段锁(Two-Phase Locking Protocol，2PL)协议：在每个事务中，所有封锁请求先于所有解锁请求。两阶段指的是封锁阶段和解锁阶段。凡是服从两阶段锁条件的事务被称为两阶段锁事务[43]。可串行化是并发执行事务正确性的准则，两段锁协议可保证并发事务的可串行化。两段式锁协议中规定：在对任何数据进行读、写操作之前，事务首先要获得对该数据的锁，而且在释放一个锁之后，事务不能再获得任何其它的锁。

乐观并发控制协议(OCC)：基本思想是将事务执行的步骤分为读取、验证、写入阶段。事务执行过程中对数据的操作在自己的独立工作区完成，而不是直接在原数据上进行，数据冲突问题的解决将被搁置到一个事务完成，其所有操作进入验证阶段。可串行化问题在验证阶段进行检测，一旦出现冲突，将会有任务被迫

重启；如果通过了验证，那么事务对事务的操作将会从独立工作区落实到原数据上。

在软实时系统下，如果物理资源有限，则锁协议优于乐观并发控制协议。然而，在硬截止期系统，当可用资源数量很丰富，并且系统性能主要由并发控制而不是其它资源调度决定时，则乐观协议优于锁协议[44]。

## 5.2 Windows 操作系统的同步机制

Windows 为我们提供了多种同步机制方便多线程编程，需要对其有深入的理解才能选择正确的方法应用到的实时数据库中以方便、快速和高效地构建各类锁。

线程的同步可分用户模式的线程同步和内核对象的线程同步两大类。用户模式中线程的同步方法主要有原子访问和临界区等方法。其特点是同步速度特别快，适合于对线程运行速度有严格要求的场合。内核对象的线程同步则主要由事件、信号量以及事件等内核对象构成。由于这种同步机制使用了内核对象，使用时必须将线程从用户模式切换到内核模式，而这种转换一般要耗费近千个 CPU 周期，因此同步速度较慢，但在适用性上却要远优于用户模式的线程同步方式。

### 5.2.1 临界区

临界区（Critical Section）是一段独占对某些共享资源访问的代码，在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

临界区在使用时以 CRITICAL_SECTION 结构对象保护共享资源，并分别用 EnterCriticalSection（）和 LeaveCriticalSection（）函数去标识和释放一个临界区。所用到的 CRITICAL_SECTION 结构对象必须经过 InitializeCriticalSection（）的初始化后才能使用，而且必须确保所有线程中的任何试图访问此共享资源的代码都处在此临界区的保护之下。否则临界区将不会起到应有的作用，共享资源依然有被破坏的可能。

在使用临界区时，一般不允许其运行时间过长，只要进入临界区的线程还没有离开，其他所有试图进入此临界区的线程都会被挂起而进入到等待状态，并会

在一定程度上影响。程序的运行性能。尤其需要注意的是不要将等待用户输入或是其他一些外界干预的操作包含到临界区。如果进入了临界区却一直没有释放，同样也会引起其他线程的长时间等待。换句话说，在执行了 EnterCriticalSection（）语句进入临界区后无论发生什么，必须确保与之匹配的 LeaveCriticalSection（）都能够被执行到。虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可用来同步多个进程中的线程。

### 5.2.2 互斥内核对象

互斥（Mutex）是一种用途非常广泛的内核对象。能够保证多个线程对同一共享资源的互斥访问。只有拥有互斥对象的线程才具有访问资源的权限，由于互斥对象只有一个，因此就决定了任何情况下此共享资源都不会同时被多个线程所访问。当前占据资源的线程在任务处理完后应将拥有的互斥对象交出，以便其他线程在获得后得以访问资源。

互斥对象的行为特性与临界区相同，但也有区别：

(1) 互斥对象属于内核对象，而临界区则属于用户对象。这意味着互斥对象的运行速度比关键代码段要慢；

(2) 互斥对象可以跨进程使用，临界区只能在一个进程中使用；

(3) 等待一个互斥对象，可以指定结束等待的时间长度，但临界区不行。

### 5.2.3 信号量内核对象

信号量（Semaphore）内核对象对线程的同步方式与前面几种方法不同，它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用 CreateSemaphore（）创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减 1，只要当前可用资源计数是大于 0 的，就可以发出信号量信号。但是当前可用计数减小到 0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过 ReleaseSemaphore（）函数将当前可用资源计数加 1。在任何时候当前可用资源计数决不可能大于最大资源计数。

### 5.2.4 管理事件内核对象

在所有的内核对象中，事件内核对象是个最基本的对象。它们包含一个使用计数（与所有内核对象一样），一个用于指明该事件是个自动重置的事件还是一个人工重置的事件的布尔值，另一个用于指明该事件处于已通知状态还是未通知状态的布尔值。

事件能够通知一个操作已经完成。有两种不同类型的事件对象。一种是人工重置的事件，另一种是自动重置的事件。当人工重置的事件得到通知时，等待该事件的所有线程均变为可调度线程。当一个自动重置的事件得到通知时，等待该事件的线程中只有一个线程变为可调度线程。

当一个线程执行初始化操作，然后通知另一个线程执行剩余的操作时，事件使用得最多。事件初始化为未通知状态，然后，当该线程完成它的初始化操作后，它就将事件设置为已通知状态。这时，一直在等待该事件的另一个线程发现该事件已经得到通知，因此它就变成可调度线程。这第二个线程知道第一个线程已经完成了它的操作。

## 5.3 GDREAL 的并发控制

### 5.3.1 GDREAL 中的事务

在 GDREAL 实时数据库系统中实时约束较弱，软实时性事务较多，系统资源有限，但抢占问题不如强实时系统那样紧张，且数据采集的优先级要高于数据访问的优先级。这种情况下，并发控制和实时事务的调度策略，需要在现有实时数据库系统理论基础上进行调整，由于系统硬截止期特性很容易满足，所以不应当作为硬截止期系统来处理[45]，应当以采用锁协议为基础进行并发控制协议的选择。

在实时历史数据库中，单纯的硬实时事务比较少，大多数是软实时事务，其时间要求没有硬实时事务严格。硬实时事务应具有最高的优先级，通常写操作大多数来自于这类事务[46]。将实时历史数据库的事务分为四类：

实时更新事务：属于硬实时事务。记录现实世界的状态或发生的时间到数据库中。它是简单的只写事务：为了保持数据库的“外部一致”，它是短暂的、周期性的，且是应被立即执行的(不能等待和阻塞)硬实时事务[47]。

实时查询事务：对于如电厂这样的流程工业企业来说，实时数据的查询应能以尽可能快的速度获得。如绘制实时变化曲线图时用到的实时数据查询，需要尽快获得当前的实时数据，但其限制却与军事系统有着本质的不同，超过截止期后不会对系统产生危害，数据的短暂延迟也是允许的，但事务的价值会急剧下降。因此，对于实时查询事务，没有一个严格的时间限制，但应尽可能满足其事务截止期要求。因此这是一个软实时事务。

历史数据更新事务：对于历史数据的更新，由于涉及到磁盘的 I/O，因此其也没有严格的时间限制，属于非实时事务，但要防止数据的丢失。

历史数据查询事务：对于历史数据的查询，也会涉及到磁盘的 I/O，且数据量会比实时数据查询多很多，因此也属于非实时事务。

另外还有一些常规事务，如备份事务等也属于非实时事务。

归结起来，这些事务可以被抽象为数据的读、写事务，从而使用统一的方法来处理。当然，根据事务调度的规则，实时数据事务的优先级高于历史数据事务的优先级。那么，事务间的并发控制可以抽象为数据读、写的并发控制。在流程工业实时数据库系统中，实时性限制不强，系统对并发控制性能和效率的要求很容易满足，数据的一致性就变得更为重要。在并发控制方面我们采用基于优先级继承的有序共享的两段锁协议。

### 5.3.2 GDREAL 的并发控制协议

锁是防止其他事务访问指定的资源控制、实现并发控制的一种主要手段。在采用传统的锁协议时可能会出现优先级反转，即出现高优先级事务被低优先级事务阻塞的现象。对于 RTDBS 而言，优先级反转是一个严重的问题，它可能导致的结果是高优先级事务被无限期的阻塞。为此，采用优先级继承协议[48]，将优先级继承的策略引入到实时数据库中。基本的优先级继承协议的思想是当优先级倒置发生时，持有锁的低优先级事务将以在等待锁的事务中的最高优先级执行，直到它释放锁。由于优先级的升级，它能够比没有优先级升级时执行得快，这样释放锁就很快，高优先级事务的阻塞时间可能降低。

在读-写锁模型中，每一个数据对象 x 有两种锁类型：读锁 rLock[x]和写锁 wLock[x]。事务 Ti 在执行一个读(写)操作 ri[x](wi[x])前必须申请获得对 x 的一个读(写)锁 rLi[x](wLi[x])，在执行完后须释放锁 rUi[x](wUi[x])。锁与锁之间有两种基本关系：排它关系和共享关系[49]。

定义 1：若事务 Ti 为操作 p 对数据 x 申请并获得了锁 pLi[x]，则其它任何事务 Tj 为某操作 q 对 x 的任何封锁请求 qLj[x]都不能成功，直至 T 释放 x 上的锁，称 pLi[x]和 qLj[x]间具有排它关系，记为 pLock≠>qLock[50]。

定义 2：若事务 Ti 和事务 Tj(Ti≠Tj)可同时对数据二拥有锁 pLi[x]和 qLj[x]以执行在 x 上的操作 p 和 q，则称 pLi[x]和 qLj[x]间具有共享关系，记为 pLock⇔qLock。

在锁协议中，锁定资源的方式有两种基本形式，一种形式是读操作要求的共享锁，另一种形式是写操作要求的排它锁。共享锁允许并行事务读取同一种资源，这时的事务不能修改访问的数据。当使用共享锁锁定资源时，不允许修改数据的事务访问数据。当读取数据的事务读完数据之后，立即释放所占用的资源。排它锁就是在同一时间内只允许一个事务访问一种资源，其他事务都不能在有排它锁的资源上访问。在有排它锁的资源上，不能放置共享锁，也就是说不允许可以产生共享锁的事务访问这些资源。只有当产生排它锁的事务结束之后，排它锁锁定的资源才能被其他事务使用。

定义 3：设事务 Ti 为操作 p 在在数据对象 x 上申请一个锁 pLi[x]，事务 Tj 为操作 q 在 x 上申请的另一锁 qLj[x]，若 pLi[x]的获得在 qLj[x]之前，则 pi[X]对数据库的操作亦必须在 qj[x]执行之前，称 pLi[x]和 qLj[x]具有有序共享关系，记为 pLock=>qLock。

有序共享就是说，两个不同的事务，可以并发地对同一对象拥有各自的锁，只要拥有有序共享关系的锁的持有者在操作执行次序上亦按照锁的获得顺序执行对数据库的操作即可。

在有序共享两段锁协议中有如下规则：

锁规则：事务在执行某一读(写)操作前必须获得一读(写)锁。

两阶段规则：事务一旦释放某个锁则不能再获得任何其它锁；

有序共享锁的获得和释放规则：如果事务 Ti 在数据 x 上获得了与事务 Tj 有有序共享关系的一个锁，则事务 Ti 和事务 Tj 在 x 上操作的执行次序必须与获得锁的次序相同。

在实时数据库系统中，并发事务间操作的执行是以获得锁的先后次序来执行的，而申请获得锁之间不再有排它关系，因而可以大大提高并发事务的执行效率，提高系统的性能。

在 GDREAL 实时数据库系统中，锁之间存在如下的关系：读锁之间是共享关系，读锁和写锁、写锁和读锁、写锁和写锁之间为有序共享关系，如表 5-1 所示，其中事务 Ti 为锁的拥有者，事务 Tj 为锁的申请者。

表 5-1 实时数据库中锁的共享关系

| Ti \ Tj | R | W |
|---|---|---|
| R | ⇔ | => |
| W | => | => |

## 5.3.3 GDREAL 的锁的粒度

为了提高系统的性能，加快事务的处理速度，缩短事务的等待时间，应该使锁定的资源最小化；但若过小，由于锁的申请和释放也需要时间，会影响系统的性能。为了控制锁定的资源，应该首先了解系统的空间管理。在 GDREAL 中，可以锁定的资源包括记录、页面、标签点和数据库。如图 5-1 所示：

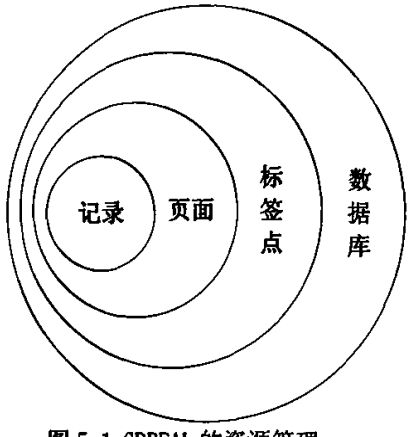


图 5-1 GDREAL 的资源管理

记录是数据的基本单位，一条记录由若干数据项组成。记录作为一个整体进行考虑和处理。最小的空间管理单位是页，一个页有 4K，由若干条记录组成。每个标签点下包含若干个页面，而全部的标签点及其数据页面就组成了数据库。

锁定比较小的对象，例如锁定记录，虽然可以提高并发性，但是却有较高的开支，如果锁定许多记录，那么需要占有更多的锁。锁定比较大的对象，例如锁定数据库，会大大降低并发性，因为锁定整个数据库就限制了其他事务访问该数据库的其他部分，但是成本开支比较低，因为只需维护比较少的锁。

## 5.3.4 GDREAL 的锁的实现

综合考虑锁的成本和事务并发性的要求，我们采用两级锁，对实时事务采用分组标签点锁，对历史事务采用标签点级的锁。

分组标签点锁：由于实时事务的操作很快就能完成，如果将锁的粒度划分得过小，则花费在申请和释放锁上的时间可能会超过事务的执行时间，虽然提高了并发度，但系统性能也会急剧下降。但如果将整个实时内存库封锁，则系统的并发度又较低，不能满足事务的截止期要求。因此，我们将标签点组织成逻辑上的组，一个组包含若干个标签点。根据系统内部的标签点个数动态分配组。这样，事务的并发操作就在组间进行，组与组间互不影响。

标签点级的锁：由于历史事务会有磁盘的 I/O 操作，因此其速度就会较慢，采用标签点级的锁能很好地满足并发性和性能要求。

将锁的管理实现为一个类：

```
class CDBLock {
public:
    CDBLock(int count);
    ~CDBLock();
    int GetRLock(int num);          // 获得读锁
    int GetWlock((int num);         // 获得写锁
    int Release((int num);          // 释放锁
private:
    CRITICAL_SECTION m_cs;          // 对其它成员的互斥访问
    HANDLE m_hsemReaders;           // 等待的读事务的信号量
    HANDLE m_hsemWriters;           // 等待的写事务的信号量
    int    m_nWaitingReaders;       // 等待的读事务个数
    int    m_nWaitingWriters;       // 等待的写事务个数
    int    m_nActive;               // 正在访问的事务个数
    BYTE *pTable;                   // 访问的资源表
};
```

系统要建立两个锁管理对象，一个是实时锁管理对象，另一个是历史锁管理对象。在初始化的时候传入一个整数 count，作为分配访问资源表 pTable 时的长度。实时锁中，count 为组的个数；历史锁中，count 为标签点的个数。对于 pTable 的每一个元素当其为 1 时表示此组或标签点已加了写锁，为 2 时为读锁，为 0 时表示未加锁。

当一个写事务锁定资源时，对于新来的读事务和写事务如果请求的是同一资源，则要等待。当前一个写事务完成时，从等待队列中优先调用写事务，只有当所有写事务完成时，才能调用等待队列中的读事务。

当一个读事务锁定资源时，对于新来的是读事务且在等待队列中没有写事务，则读事务间可以申请获得读锁，读事务间并发访问资源。但如果一个写事务请求同一资源时，则马上将正在执行的读事务的优先级提升为写事务的优先级，新来的读事务不能获得共享读锁，以使正在运行的读事务尽快完成，然后调用等待队列中的写事务，只有当所有写事务完成时，才能调用等待队列中的读事务。这样就实现了基于优先级继承的有序共享的两段锁协议。

## 5.4 小结

在本章中，详细介绍了实时历史数据库的事务和并发控制，结合 Windows 操作系统提供的同步机制和传统的事务并发控制协议，区分实时事务和非实时事务，设计和实现了基于优先级继承的有序共享的两段锁协议。

# 第六章 结论和展望

实时历史数据库是工业企业自动化体系中的一个关键技术，研究和开发具有自主知识产权的实时历史数据库产品具有重大的理论研究和现实意义。本课题在自主开发的 GDREAL 系统的基础上，完成了以下工作：

以磁盘历史数据库的开发为背景，分析了磁盘历史数据库设计中的问题，对制约历史数据库发展的瓶颈——磁盘 I/O、文件的索引和数据的组织结构等进行了深入的研究，提出了一种新的磁盘存储结构是——Z 树结构，减少了磁盘 I/O 操作次数，极大地提高了磁盘存储性能；采用文件组的方式，设计了一个高效的文件集管理系统；对写入页面的大小和查询性能做了分析。

以实时内存数据库的开发为背景，分析了实时数据的存储流程，完成了实时内存数据库和历史数据缓冲区的设计和实现：为了避免内存碎片和减少动态内存分配和释放的开销，设计了一个内存池，将动态内存分配改为预先分配；结合历史数据的特点，设计实现了基于双队列的内存页面 LRU 淘汰算法，将之用于历史数据缓存页面的管理；采用断点续传机制，很好地实现了大量数据的分批返回。

详细介绍了实时历史数据库的事务和并发控制，结合传统的事务并发控制协议，区分实时事务和非实时事务，提出并设计了基于优先级继承的有序共享的两段锁协议。

综上所述，本文紧密结合工程实际，分析了当前实时历史数据库系统的理论和技术，设计和开发出一套适用于电力企业等流程工作实际的大型实时数据库系统中的磁盘历史数据库和实时内存数据库。但实时数据库的出现毕竟还不到 20 年时间，很多研究工作还未深入，相关技术也还不成熟，GDREAL 一期工程中还有不足的地方。随着 GDREAL 二期工程即将展开，在一期工程的基础上还有以下需要改进的地方：对于非时间顺序数据的插入，系统的性能会有较大的下降，对于这一部分有较多的工作要做；对于事务和并发控制方面的研究要进一步加强；对于数据库的安全性和稳定性还有待进一步的提高。

# 致 谢

首先，我要向我的导师郝玉洁副教授表示最衷心的感谢和崇高的敬意。郝老师严谨认真的治学精神、耐心宽厚的师长风范和诲人不倦的精神一直影响和激励着我。在大学时，郝老师就给了我极大的鼓励和帮助。在研究生学习期间，郝老师为我创造了良好的学习和研究环境，提供了难得的实践机会。她不但在学业中给了我极大的帮助和精心的指导，还关心我的生活，对于个人的成长和发展指明了方向。遇到了郝老师，使我的人生有了重大的转折，我将永远铭记在心。

衷心感谢郭建东老师。郭老师有着深刻的洞察力和忘我的工作精神。他在我的研究方向上给予我很多指导性意见，在做课题的一年多时间里，郭老师更是对我的课题进行了直接的指导和帮助，没有他，我的毕业课题不可能顺利完成。郭老师在生活中平易近人，兄长般关心、爱护他的学生。不能继续跟着郭老师做项目，不得不说是一种遗憾。

谢谢张禹、陈鑫铎、马成、麻建华、郭建新，我们在一个项目组中互相学习、互相帮助，度过了一段愉快的时间。还要谢谢教研室的同学高翔、刘涛庆、王晓、江广顺、杜英、王榕。谢谢北京一同工作过的同事蒋禾青、程睿君、马令智和武芳。谢谢关心支持我的朋友。

谢谢我的父母和姐姐、哥哥及家人，他们平凡而伟大的爱一直激励着我不断努力。谢谢我的女朋友，她的爱一直陪伴着我，使我坚持走到今天。

谢谢抽出宝贵时间对论文进行评阅和答辩的专家、教授和老师，谢谢他们的辛勤工作。

研究生阶段的学习是我一生最宝贵的精神财富，将使我终生受益。谢谢电子科技大学!

# 参考文献

[1] 褚健,孙优贤,流程工业企业综合自动化技术发展的思考.Http://www.cipsea.com, 2002

[2] OZGUR U., Alejandro Buchmann, A Real-time Concurrency Control Protocol for Main-Memory Database Systems. Information Systems, 1998, 23(2): 109-125

[3] 杨玲,漆永新,实时数据库概述.冶金自动化,1996, 11(3):510

[4] 王能斌,数据库系统教程.北京：电子工业出版社, 2005, 1-10

[5] 郝欣,牟长信,郑伟.应用实时数据库实现电厂生产过程信息共享.东北电力技术, 2006,(1): 17-19

[6] 韩玫瑰,史明华,马涛,董学仁.DCS 组态软件实时数据库系统的设计.自动化仪表, 2006,27(1): 18-21

[7] 周东球,杜殿林,左信等. 先进控制软件系统实时数据库的设计. 微计算机信息, 2003, (10): 20-21

[8] 赵淑芳,任建平.分布式实时数据库并发控制.机械管理开发.2005, 82(1):83-84

[9] 肖迎元,刘云生,廖国琼.主动实时内存数据库系统的数据交换策略及实现.计算机工程与应用,2004,29

[10 刘云生,胡国玲,舒良才.实时主存数据库事务的预处理.软件学报,1997,03

[11] 刘云生,卢炎生,李国徽.实时数据库系统及其特征, http://www.embed.com.cn

[12] 刘笑天.生产信息化建设实时历史数据库平台 iHistorian. 测控技术,2003, 22(6)：69-71

[13] http：//www.microsoft.com

[14] 严蔚敏,吴伟民,数据结构(第 2 版), 北京:清华大学出版社,1992

[15] Jeffrey Richter. Windows 核心编程.北京:机械工业出版社,2000

[16] Garcia-Molina H., Salem K., Main memory database systems:an overview, Knowledge and Data Engineering, IEEE Transactions. 1992, 4(6):509-516

[17] 卢炎生,韩琪,主动实时数据库事务及其处理.计算机研究与发展.1998, 35(2):188-191

[18] 刘云生,潘琳,实时数据库系统的内存数据库组织与故障恢复.小型微型计算机系统, 2001,22(5):611-613

[19] Aranha R. F. M., Narayanan V G.S., Muthukrishnan C. R., S Prasad S. T., Ramamritham K.. Implementation of a Real-Time Database System. Information Systems.1996,21(1): 55-74

[20] 钟远明,奚建清,分布式实时数据库系统中事务处理的研究.计算机应用研究. 2002, (2):

75-78

[21] Haritsa J. R., Ramamritham K., Real-Time DataBases in the New Millenium. Real-Time Systems, 2000

[22] 黄梆,数据库的延时问题及技术解决办法.图书情报工作, 2001,3: 73-76

[23] Anindya D., Sarit M., Igor R. V. Buffer Management in Real-Time Active Database Systems. The Journal of Systems and Software, 1998,42:227-246

[24] Silberschatz A , Galvin P., Gagne G. Applied operating systemconcepts. 北京:高等教育出版社, 2001

[25] VAho A, Denning P, Ullman J. Principles of Optimal Page Replacement. In Journal of ACM, 1971 ,18:80-93

[26] Borodin A, Irani S , Raghavan P , et al. Competitive Pagingwith Locality of Reference. In Journal of Computer and System Sciences, 1995, 50:244 - 258

[27] Chrobak M, Noga J. LRU is Better than FIFO. In Proc 9th Annual ACM - SIAM Symp on Discrete Algorithms. Philadelphia, PA , USA:SIAM, 1998, 78-81

[28] 汤子瀛. 操作系统原理(第 2 版). 北京:清华大学出版社, 2001

[29] 张俊花. 对 LRU 页面置换算法的理论改进, 太原师范学院学报, 2004, 3(2): 30-31

[30] 李涛, 李慧, 谷建华, 潘慧芳,基于 ACE 的并发编程模式和池式内存分配的研究, 计算机工程与设计,2006,27(1): 26-28

[31] 林伟璐,钱宇,李秀喜,郑秀玉,化工过程集成运行系统的研究.化工自动化及仪表, 2000, 27(1):1-3

[32] Garcia-Molina H., Jeffery D. U., Jennifer W.著,杨冬青,唐世渭,徐其钧等译,数据库系统实现. 北京:机械工业出版社,2001

[33] OZGUR U., Research Issues in Real-Time Database Systems, Information Sciences, 1995, 87: 123-151

[34] Jim Gray, Andreas Reuter 著,孟小峰,于戈等译,事务处理概念与技术.北京:机械工业出版社, 2004

[35] 方来华,吴爱国,何熠,组态软件核心技术研究,化工自动化及仪表,2004,31(1):33-35

[36] Quazi N.A., Susan V V, Maintaining Security and Timeliness in Real-time Database System. The Journal of Systems and Software, 2002, (61):15-29

[37] L Sha, R Rajkumar, J P Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Trans on Computers, 1990, 39 (9):1175-1185

[38] Kam-yiu Lam, Tei-Wei Kuo, Tsang N W H, et al. The Reduced Ceiling: Protocol for

Concurrency Control in Real-Time Databases with Mixed Transactions. The Computer Journal, 2000, 43 (1):65-80

[39] Kam-Yiu Lam, Tei-Wei Ku, Ben Kao, et al. Evaluation of Concurrency Control Strategies for Mixed Soft Real-Time Database Systems. Information Systems, 2002, 27 (2):123-149

[40] 卢炎生,刘玮,赵小松.并行实时数据库系统中的一种自适应并发控制模式[J].计算机工程与科学. 2006, 28(2):86-89

[41] Ming X., Krithi R., Jayant R.H., John A.S.. A State-conscious Concurrency Control Protocol for Replicated Real-time Databases. Information Systems, 2002, 27:277-297

[42] 夏家莉,实时数据库系统中的并发控制协议及分析.计算机工程与应用, 2002, (7):200-202

[43] 祁鑫,王文海.实时数据库系统并发控制机制综述.工自动化及仪表.2006,33(1):47-50

[44] Yu-Wei C., Le G, Effects of Deadline Propagation on Scheduling Nested Transactions in Distributed Real-Time Database Systems. Information Systems, 1996, 21(1):103-124

[45] 雷晓平,罗海天,尹传高,实时内存数据库的并发控制模型.微计算机应用, 1998, 19(3): 158-161

[46] Ricardo M. F., Antonio P., Kishor S. T., Performance analysis of distributed real-time databases. Performance Evaluation, 1999, 35:145-169

[47] 万常选,夏加莉,一种实时数据库系统的基于时间戳的多版本并发控制协议.计算机工程与应用, 2001,15:22-24

[48] 罗蕾. 嵌入式实时操作系统及应用开发.北京：北京航空航天大学出版社,2005

[49] 萨师煊,王珊,数据库系统概论.北京：高等教育出版社. 2002

[50] 王成光.流程工业大型实时数据库理论、技术与应用[博士学位论文],浙江：浙江大学, 2003, 60-62

# 攻硕期间取得的研究成果

[1] 曾强,郝玉洁,郭建东.实时历史数据库库文件结构设计与分析.微计算机信息.已录用

[2] GDREAL 实时历史数据库系统中的磁盘历史数据库和实时内存数据库的设计和实现，投入现场运行

[3] MINET 网络监控系统的设计和实现,某安全部门项目