第 3 章　嵌入式数据库的存储与组织

CHAPTER 3

在嵌入式系统中，由于处理器速度、体积、移动性、访问速度等方面的特殊要求，在存储器的组织结构和数据的访问组织方面与传统数据库有着显著的差异。本章首先介绍了嵌入式数据库的物理层，其中包括嵌入式数据库的存储体系和主要的存储介质及其特性，重点介绍了 Flash 存储介质的优点、存储中需要解决的主要问题和其中使用的文件系统，接着介绍了文件型数据库中使用的散列技术和索引技术的原理，最后介绍了内存型数据库的数据组织方式，如区段式存储、T 树、N-Array 模型等。

3.1　嵌入式数据库的存储体系

传统的数据库中存储结构分为内存、磁盘、磁带三层。内存为程序运行和数据处理的工作场所，它存取速度快，但容量有限、信息易失。硬盘容量大，可以永久存储信息，但存取速度慢。在嵌入式数据库系统中，数据与事务可能有显式的截止期限限制，特别是一些紧急数据和事务，读写磁盘的 I/O 操作很可能使事务与数据错过它的截止期限。而且，向磁盘写日志也会影响事务处理的效率。另外，磁盘体积大、防震和抗恶劣环境的能力差，不适合用于嵌入式系统中。在各种存储介质中，Flash 的价格相对较低且信息可永久保留。与硬盘相比不仅存取速度快、体积小，而且便于插拔，是代替硬盘作为嵌入系统永久存储设备的最佳选择。但 Flash 读写速度不对称，读取速度快，读数据的时间$<10^{-7}$s，与 RAM 相当，且可按地址读取；但是写入速度相对慢，写入一次大约需 10^{-5}s。NVRAM 兼有内存读写速度快和在后备电池维护的情况下非易失的特点，从而广泛用于嵌入式系统中，但它价格昂贵，不便插拔，通常与 Flash 配合使用，充当 Flash 写缓冲区。对于实时系统中的紧急且需要永久保存的数据，当写 Flash 会错过它的截止期限时，可先写入 NVRAM，紧急事务的日志也可写入 NVRAM 中。典型的嵌入式数据库的物理存储结构如图 3-1 所示。

在层次结构中，RAM、NVRAM 构成嵌入式数据库管理系统的内存，Flash 则作为嵌入式数据库管理系统的外存。在内存与外存之间，以及内存的 RAM 与

NVRAM之间都存在数据交换。数据库可采用嵌入式设备提供的通信接口或者I/O接口，在外存和磁盘、光盘、磁带等大容量的数据转储设备之间进行数据交换。在RAM数据处理过程中，NVRAM作为它的快速保存数据场所，只要NVRAM有后备电池支持，就可以将频繁存取的数据存放在NVRAM中，而不必担心突然断电造成的数据丢失。在适当的时候，可以将NVRAM中的数据存回Flash。在内存缓冲区设置得足够大的情况下（内存缓冲区的大小通常可以通过配置文件进行设定），甚至可以将整个数据库装载入内存，这时，数据库就成为一个内存数据库。NVRAM除了保存高频度使用数据，还能保存事务运行日志。在数据库的工作版本常驻内存的情况下，保存事务日志尤为重要，因为日志将成为突然掉电情况下数据库恢复的重要手段。需要说明的是，由于NVRAM造价昂贵，不可能要求将来大众化的嵌入式移动设备中都配备NVRAM；其次，在便携式PC中也不可能配备NVRAM。因此，图3-1是典型的嵌入式数据库的物理存储体系结构，在实时性、安全性要求高的嵌入式移动设备中使用，而在一般的情况下，则是在内存中开辟缓冲区，在适当的时候将数据写回外存。

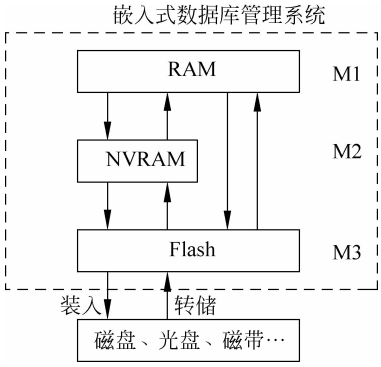

图3-1　物理存储体系结构图

对于嵌入式数据库而言，数据特征和事务特征决定着数据的存放位置。系统中数据的各种特征如图3-2所示。

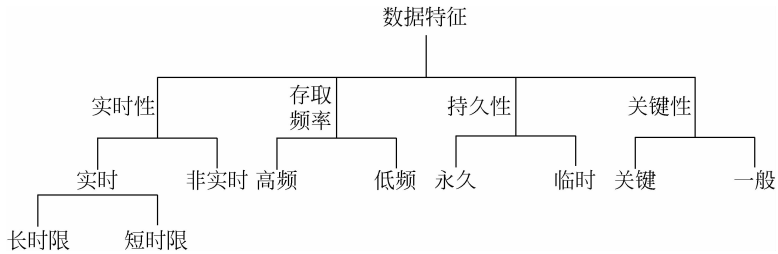

图3-2　嵌入式数据库的数据特征分类

对于图3-1所示的存储体系结构，各种不同特征的数据的放置策略如下。

1. 实时性

实时性指数据具有时间限制。数据的访问有着截止期限限制，访问完成时间落在此数据的截止期限内时该数据才有效。按照实时性，数据可分为非实时数据和实时数据。

（1）非实时数据无截止期限与之相连，有效性不受时间的影响。这类数据可存放在存取速度慢的介质上，如M3中。

（2）实时数据可分为长时限数据与短时限数据。长时限数据与非实时数据可存储到M3上，由于它的长时限，事务对存在M3上的数据读、写、修改不会使它失效；短时限数据必须保存在M1或M2中，当需要存档保留时才存放到M3中。

2. 存取频率

按照存取频率，数据可分为高频数据和低频数据。

（1）高频数据即常说的“热点”数据，因为需要经常读取和修改，所以最适合放置于 M2 中。

（2）低频数据很少被存取，所以可常驻 M3，当需要时才取到内存。

3. 永久性

按照永久性，数据可分为永久数据和临时数据。

（1）永久数据需要长期反复使用，它们必须存于 M2 或 M3。

（2）临时数据是短期使用的数据，如中间结果等，这类数据只需保留在内存中。当不再使用时就将所占的内存空间释放。

4. 关键性

关键性是指数据对事务处理的重要性，它可以是数据本身的固有特性，也可以是由事务存取该数据的关键性而引起的特性。

各类不同关键度数据在事务处理过程中的要求如下：

（1）关键数据的有效期与一致性必须确保，一旦发生故障，它们必须尽快恢复。为了确保其事务的高性能要求，关键数据存放在 M2 中。

（2）一般数据的有效性对事务的影响不大，根据其特征来决定其位置。

（3）事务的特征对其处理数据的放置有很大的影响，如周期的和紧急事务要求数据存放于 M1 中，高优先级事务要求数据存放于 M1 中，低优先级事务要求数据存放于 M3 中。

除了物理体系结构之外，对于一个计算能力有限、存储资源紧张的嵌入式终端而言，运行在其上的嵌入式数据库管理系统应采取不同于传统数据库的设计风格。它应当尽量利用操作系统提供的便利，避免提供与操作系统重复的机制，以及通过增加模块来强化某些操作系统本身已经提供的操作，以保持较小的内存占用。例如，嵌入式数据库通常避免直接操作外存，而使用操作系统提供的文件系统接口实现外存处理。

图 3-3 为嵌入式数据库的逻辑存储体系结构图。外存中的数据包括数据字典、索引和数据三类。因为数据字典占用空间很小且存取频繁，所以将其常驻内存。其他两类数据则使用高活跃度数据存储区域存储和管理。高活跃度数据存储区域以记录为单位存储和组织被频繁访问的数据，并将发生改变的数据写回到操作系统文件缓冲区。实际统计表明，数据库经常访问的数据一般不超过其数据总量的 20%，因此高活跃度数据存储区域的存在可以极大地降低外存 I/O 的次数。

在图 3-3 中，操作系统在内存中提供了文件缓冲区，用以减少外存 I/O 的次数，提高外存读取的效率。它与高活跃度数据存储区域的作用并不重合，高活跃度数据存储区域直接针对数据库系统的特点而设计，可以根据数据库的特点为其设计调入调出算法。操作系统的数据缓冲区在各方面都更类似于传统数据库系统的数据缓冲区，而且它不是为某种特定类型的应用而设计的，它是全局的内存缓冲区域，为运行在操作系统中的所有应用访问外存时提供频繁读取块的缓冲。

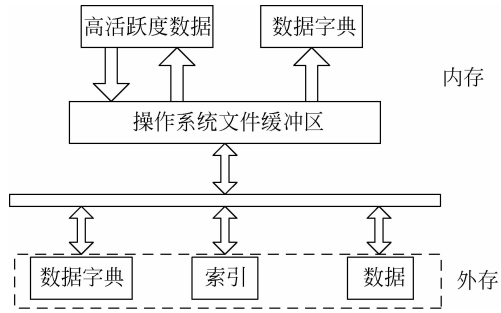

图 3-3　逻辑存储体系结构图

3.2　嵌入式数据库的存储介质

传统桌面计算机存储系统从内向外，依次为寄存器、Cache、内存、外存，其访问速度越来越低，访问的粒度和存储容量越来越大。嵌入式系统也遵循以上的规律，但嵌入式系统在持久性存储介质（外存）方面较传统计算机设备有着很大的区别，它通常由 Flash 组成。嵌入式系统中的各种存储器类型及其比较如表 3-1 所示。

表 3-1　嵌入式设备中存储器的特征比较

种类	易失性	可写性	擦除单位	擦除次数	速度	价格
SRAM	是	是	字节	无限制	快	昂贵
DRAM	是	是	字节	无限制	适中	适中
MaskRAM	否	否	—	—	快	便宜
PROM	否	一次	—	—	快	适中
EPROM	否	是	整个	有限次	快	适中
EEPROM	否	是	字节	有限次	读快、写慢	昂贵
Flash	否	是	扇区	有限次	读快、写慢	适中
NVRAM	否	是	字节	无限次	快	昂贵

可见，在内存方面嵌入式移动设备和传统计算机的差别不大，但在外存的配置上，却有多种选择。一般来说，在众多类型的存储器中，嵌入式移动设备通常使用的是 Flash，也称为闪存或 Flash 卡。目前，Flash 卡越来越成为嵌入式移动设备的主流持久性存储设备。根据不同的生产厂商和不同的应用，闪存卡分为 SmartMedia（SM 卡）、CompactFlash（CF 卡）、MultiMediaCard（MMC 卡）、Secure Digital（SD 卡）、Memory Stick（记忆棒）、XD-Picture Card（XD 卡）和微硬盘（Microdrive）等种类。这些闪存卡虽然外观、规格不同，但是技术都是相同的。

严格地说，Flash 存储器是 EEPROM 的一种，它主要分为 NOR 和 NAND 两类。NOR Flash 带有 SRAM 接口，有足够的地址引脚寻址，可以很容易地存取其内部的每一个字节。NAND Flash 使用复杂的 I/O 口串行地存取数据，读和写操作均采用 512B 的块，这一点与

硬盘操作类似。NOR 与 NAND Flash 的性能比较如表 3-2 所示。

表 3-2　NOR 和 NAND Flash 的性能比较

类　　型	NOR	NAND
存储容量	1～16MB	8～512MB
读取方式	按字节随机读取	按页读取
写入速度	慢	快
擦除单位	8～128KB	16KB,每块分为 32 个 512B 的页面,每页面有 16B 附加数据用于存放元数据或者错误校验码
擦除次数	约 10 万次	约 100 万次
擦除时间	370ms/块	2ms/块
可靠性	高,出厂时无坏块	低,出厂时有坏块

Flash 存储器的主要特点如下：

(1) Flash 存储器可以像其他 ROM 那样直接读取数据,但写入数据的操作与传统的 EEPROM 不同。Flash 的写操作只能有选择地把一些位从逻辑“1”置为逻辑“0”,而无法将逻辑“0”置为逻辑“1”。由于这个特性,如果向同一个单元写入多次则该单元最终的值将是曾经写入的所有值逻辑与的结果,而非最后一次写入的值。

(2) 为了向已经执行过写操作的单元写入新的内容但不会因原来的内容而改变,必须事先对单元执行擦除操作。Flash 的擦除操作是将指定区域内的全部逻辑“0”复位到逻辑“1”,擦除以区块(Block)为单位,必须整块擦除。由于擦除过的区块中所有位均被置为逻辑“1”,而任何值和逻辑全“1”的值相与都不会改变,因此接下来写入的内容就会被完整地记录下来。

(3) 一般来说,NOR Flash 区块在一次擦除后可以进行任意次随机写入,且写入操作以字为单位(通常为 16 位或 32 位),而 NAND Flash 区块在一次擦除后只能进行有限次写入,写入操作以页面(Page)为单位(通常为 512B 或 2048B),可以一次只写入页面的一部分。NAND Flash 每页都有额外的带外空间(Out Of Band,OOB),512B 的页面配有 16B 的 OOB,2048B 的页面配有 64B 的 OOB。

(4) Flash 中每块的可重复擦除次数是有限的(通常为 100 000 次),当达到寿命后,对该区块的擦除可能会失效,有些位将无法从逻辑“0”复位到逻辑“1”。

在访问速度方面,NOR Flash 在 10μs 内可以写 1 个字的数据,这样在 32 位总线的情况下,写 512B 数据需要 1280μs。而 NAND 写的速度是 50ns/B,加上在不同页之间的移动时间,在 8 位总线的情况下,写 512B 数据需要 236μs。可见,NAND 具有较快的写入速度,同时 NAND 型 Flash 的成本较低,所以对于需要大容量非易失性存储的应用,比如智能手机、数码相机以及 PDA 或是移动存储设备(如 U 盘)等,NAND 型 Flash 是比较好的选择。

NAND Flash 在具有众多优点的同时,也有自身的缺陷,目前 NAND Flash 在应用中的缺点以及克服这些缺点的方案主要有以下几个。

(1) 坏块处理：某些区块不能保证一直正常地使用,由于“先天性的缺陷”或者过度地执行擦除操作的原因,这些区块不能继续使用。为了解决这个问题,可以设置一个字段来作为坏块的标识,如果标记为真,则该块不能继续使用。可以在使用初期开设一些区块,一旦发现有不能使用的块,这些块就会被事先开设的区块所替代；或者改变映射表,隔离这些

坏块。

(2) 平衡磨损(Wear-Leveling)：区块的擦除操作执行总次数达到一定大的值时就不能继续使用，有些区块的数据由于经常变动而先达到擦除操作次数上限，从而对 NAND Flash 造成重大影响。可以设计专门的平衡磨损算法，使系统的写操作平均分布到各个区块，让擦除操作执行次数尽量接近，从而使 NAND Flash 得以长久使用。

(3) 垃圾回收(Garbage Collection)：对 NAND Flash 的写入操作只能在已经执行了擦除操作的区块中进行，写入操作完成之后原来结点所在的页面会被转变成为脏页。随着时间的推移，写操作的不断发生导致了系统中脏页增多，空闲页减少。当空闲数量减少到一定程度时，系统就要启动垃圾回收，对这些脏页所在的区块进行擦除操作，从而得到空闲空间以供系统使用。

如 3.1 节所述，嵌入式数据采用文件的方式在外存中持久存储。鉴于上述 Flash 存储器的特点及嵌入式系统特性，设计一个针对 Flash 的稳定可靠的文件系统必须考虑以下问题：

(1) 写 Flash 前需要擦除，且不能擦除比区块更小的单位，因此对文件的修改不能简单地直接覆盖。

(2) 在对文件操作的整个过程中都有可能发生意外(例如嵌入式系统常见的断电)，文件系统需要有从意外中恢复的能力，并具有一定的日志特性以确保数据完整性。

(3) 由于 Flash 区块可重复擦除次数有限，不适合采用通常文件系统中的集中式 FAT 结构，否则存放 FAT 的区块将会比其他数据区块先行损坏，而一旦 FAT 损坏则整个文件系统都将崩溃。

(4) 基于同样的原因，需要进行磨损控制，尽量限制并平衡各区块的擦除次数，避免在文件系统使用量较小的情况下只利用较前端的区块并反复擦写。

(5) 对于已经损坏的块，文件系统必须有能力检测并标记。

(6) 需要对处理时间与占用内存空间这对矛盾进行权衡。

目前，针对 Flash 存储设备的嵌入式 Linux 存储系统的解决方案主要有两种：第一种是在驱动与文件系统之间添加 FTL(Flash Translation Layer)层，为传统的磁盘文件系统提供模拟和透明的 Flash 存储器块设备操作接口，Flash 存储器的磨损平衡和垃圾回收管理由 FTL 层实现；第二种是设计基于 Flash 存储器的专用文件系统，让文件系统在硬件驱动程序的基础上直接管理 Flash 上的数据。现在比较常用的有 JFFS/JFFS2(The Journalling Flash File System，日志闪存文件系统)、YAFFS/YAFFS2(Yet Another Flash File System，另一个闪存文件系统)、UBIFS(Unsorted Block Image File System，无排序区块图像文件系统)等。它们都是对 Flash 设备进行直接管理的文件系统，其管理的方法和日志结构文件系统(Log Structured File System)非常相似，都采用日志记录的方法来管理 Flash 设备，这样可以避免 Flash 设备不能“就地更新”的问题。其主要实现思想是：当向 Flash 中写入新数据时，文件系统并不把原来的数据块从 Flash 中立即擦除，而是在 Flash 的空闲区中写入新的数据块，并使原有的数据块变成过时的和无效的。由于保留了文件的历史数据，这样就为恢复文件的原有数据提供了可能，提高了 Flash 文件系统的健壮性。

下面介绍几种常用的 Flash 文件系统。

(1) JFFS/JFFS2

JFFS是一种应用于Flash存储器上的日志文件系统，由瑞典Axis Communications AB公司开发。2001年3月17日，RedHat公司在此基础上发布了第二代闪存日志文件系统(JFFS2)。Linux实现中，JFFS必须建立在MTD(Memory Technology Devices，内存技术设备)驱动程序的上层(如图3-4所示)。这里MTD的作用就是为JFFS提供操作NAND或者NOR芯片的接口。

JFFS是针对以闪存为存储介质的嵌入式系统，所以充分考虑了闪存的物理局限性，使用了尽可能高效的日志系统。和前面介绍的TrueFFS以及其他中间层驱动相比，JFFS是专门针对闪存的文件系统，除了具有日志功能，还包含负载平衡、垃圾收集等功能。另外，这个文件系统是源代码公开的，方便学习和使用。

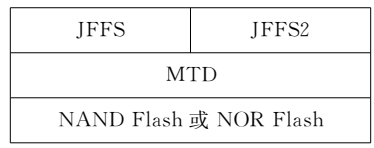

图3-4　应用JFFS/JFFS2的块设备层次结构图

(2) YAFFS/YAFFS2

YAFFS文件系统是专门针对NAND闪存设计的嵌入式文件系统，目前有YAFFS和YAFFS2两个版本，两个版本的主要区别在于YAFFS2能够更好地支持大容量的NAND Flash芯片。YAFFS文件系统有些类似于JFFS/JFFS2文件系统，与之不同的是JFFS1/2文件系统最初是针对NOR Flash的应用场合设计的，而NOR Flash和NAND Flash本质上有较大的区别，所以尽管JFFS1/2文件系统也能应用于NAND Flash，但由于它在内存占用和启动时间方面针对NOR的特性做了一些取舍，所以对NAND来说通常并不是最优的方案。

(3) UBIFS

UBIFS是一种由Nokia工程师和赛格德大学(University of Szeged)联合开发的新型Flash文件系统。UBIFS一般被认为是JFFS2后的下一代Flash文件系统。

JFFS2工作在MTD层上，而UBIFS工作在UBI层上，不能简单应用在MTD层上。换句话说，UBIFS由三部分组成：MTD层、UBI层和UBIFS层。与JFFS2不同，UBIFS很多针对Flash的设计都放在了UBI层中实现，比如负载均衡、块管理、坏块管理等。与JFFS2相比，UBIFS有很多优点，比如挂载速度、读写速度更快等，是一种JFFS2的替代方案。

3.3　磁盘型数据库

磁盘型数据库又称为基于磁盘的数据库，其数据主要存放在磁盘中，以物理文件的形式(数据文件)存储。当读取数据时，数据块需要从磁盘中加载(也可以说是复制)到内存中。更改数据时，首先更改数据库缓存(内存)中的数据块副本(内存中被更改的数据块称为脏内存)，然后再以同步或异步的方式将内存中更改的数据写入到磁盘。所以，在磁盘型数据库中，数据的读取和修改都存在着大量的I/O，包括随机的I/O(数据的写入)和顺序的I/O(用作日志的写入)。由于数据主要是存放于磁盘中，所以磁盘数据库可以存储海量数据，几百

TB甚至上千TB,这取决于磁盘阵列的容量。磁盘型数据库的基本架构如图3-5所示。在图3-5中,应用程序访问数据的时候,通过数据库提供的查询接口访问由一定方式(如Hash)组织的数据目录结构,如果待读取的数据已经在内存缓冲区中存在,则读取缓冲区中数据并返回给应用程序,否则,从磁盘中读取相应的数据放入内存缓冲区中,然后从缓冲区中读取数据返回应用程序。磁盘型数据库采用的数据组织方式包括散列方式与索引方式。

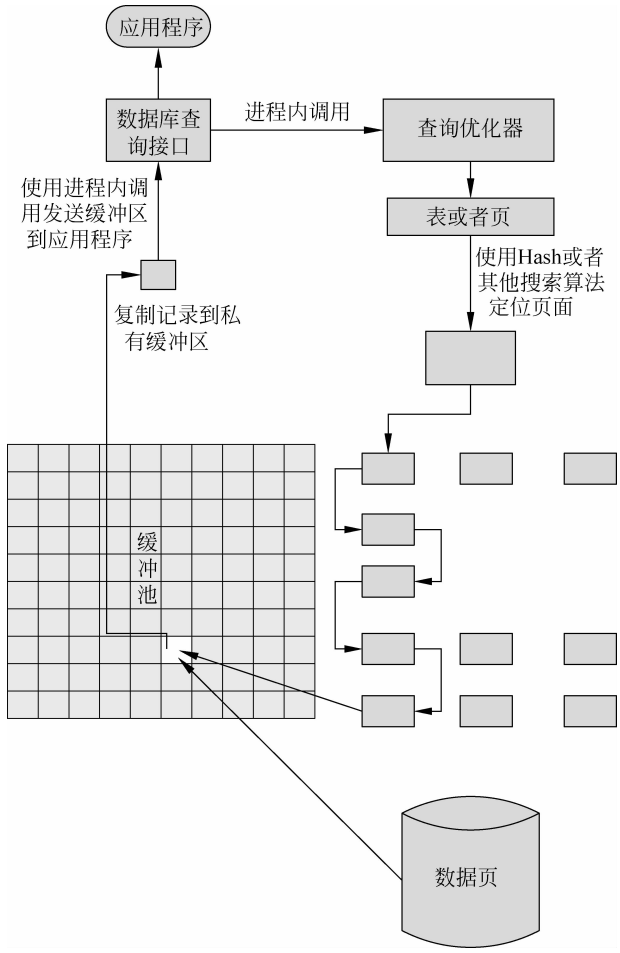

图3-5 磁盘型数据库的基本架构

3.3.1 磁盘型数据库的散列技术

1. 散列概念

散列索引又称哈希(Hash)索引,是以记录的某个属性值为参数,通过特定散列函数求得有限范围内的一个值作为记录的存储地址,是一种支持快速存取的索引方式。在散列结

构中，需要选择一个带有随机特性的函数（称为散列函数），它以一个（或一组）指定的查找键值为参数（通常称为散列域），计算出一个称为散列值的函数值。查询时以该函数的值找到记录所在的磁盘块，读入主存缓冲区，然后在主存缓冲区中找到记录。存储时也是根据散列函数计算存储块号，然后存入相应单元。

在数据库技术中，一般使用“桶”（block）作为基本的存储单位。一个桶可以存放多个记录。每个桶对应一个磁盘块，有唯一的编号。

散列技术中涉及散列函数 H，函数 H 是从 K（所有查找键值的集合）到 B（所有桶地址的集合）的一个函数，它把每个查找键值映像到地址集合中的地址。

要插入查找键值为 K_i 的记录，首先应计算 $H(K_i)$，以此作为记录存储的桶地址，然后把记录插入到桶内的空闲空间。

在文件中检索查找键值为 K_i 的记录，首先也是计算 $H(K_i)$，求出该记录的桶地址，然后在桶内查找。在散列方法中，由于不同查找键值的记录可能对应于同一个桶号，因此一个桶内记录的查找键值可能是不相同的。因此，在桶内查找记录时必须检查查找键值是否为所需的值。

在散列文件中进行删除操作时，一般先用前述方法找到欲删除记录，然后直接从桶内删去即可。

2. 散列函数

使用散列方法最坏的情况是可能把所有的查找键值映射到同一个桶中，致使所有的记录存放在同一个桶中，查找一个记录就必须检查所有记录。理想情况是散列方法把储存键值均匀分布到所有桶中，这首先需要有一个好的散列函数。

好的散列函数在把查找键值转换成存储地址（桶号）时，一般满足下面两点：第一，地址的分布是均匀的，即产生的桶号尽量不能聚堆；第二，地址的分布是随机的，即所有散列函数值不受查找键值各种顺序的影响。

在日常应用中，最常使用的散列函数是“质数求余法”。其基本思想是：首先确定所需存储单元数 M，给出一个接近 M 的质数 P；再根据转换的键号 K，代入公式 $H(K)=K-\mathrm{INT}(K/P)\times P$ 中，以求得数据作为存储地址，一般 $0\leqslant H(K)\leqslant P-1$。

采用散列方法时，总希望能通过计算将记录均匀分配到存储单元中去。实际上，无论采用哪一种方法，都不可避免地会产生碰撞现象，即两个或多个键值经过计算所得到的结果相同而发生冲突。

例如，设 $M=10$，$P=7$，则：

$H(1)=1$

$H(2)=2$

$H(3)=3$

$H(4)=4$

$H(5)=5$

$H(6)=6$

$H(7)=0$

$H(8)=1$

$H(9)=2$

$H(10)=3$

可见 1 和 8,2 和 9,3 和 10 发生冲突。

散列函数应仔细设计。设计得不好,会造成各个桶内的查找时间有长有短;设计得好,则各个桶内的查找时间相差无几,并且查找的平均时间是最小的。

3. 散列碰撞

从上述例子可见,不同的查找键值对应散列函数的值相同是一个普遍现象。在散列组织中,每个桶的空间是固定的,如果某个桶内已装满记录,还有新的记录要插入到该桶,那么称这种现象为桶溢出(也称为散列碰撞)。产生桶溢出的原因主要有两个:其一,初始设计时桶数偏少;其二,散列函数的"均匀分布性"差,造成某些桶存满了记录,而某些桶有较多空闲空间。

在设计散列函数时,桶数应放宽一些,一般桶数比正常需求大 20%,以减少桶溢出的机会。

桶溢出现象在所难免,一旦发生桶溢出时常采用如下方法进行处理:如果某个桶(称为主桶)已装满记录,还有新的记录等待插入该桶,那么可以由系统提供一个溢出桶,用指针链接在该桶的后面。如果溢出桶也装满了,那么用类似的方法在其后面再链接一个溢出桶。这种方法称为溢出链方法,也称为封闭散列法。例如,一个散列文件中共有 16 个记录,其关键字依次为 23、05、26、01、18、02、27、12、07、09、04、19、06、16、33、24。桶的容量 $m=3$,桶数 $b=7$,用除模取余法,令模数为 7,$H(K)=K$ MOD 7。当发生碰撞时采用链接溢出桶,图 3-6 是散列结构的溢出链示意图。记录的查找不仅要在主桶中查找,也可能要到后面链接的溢出桶中去找。

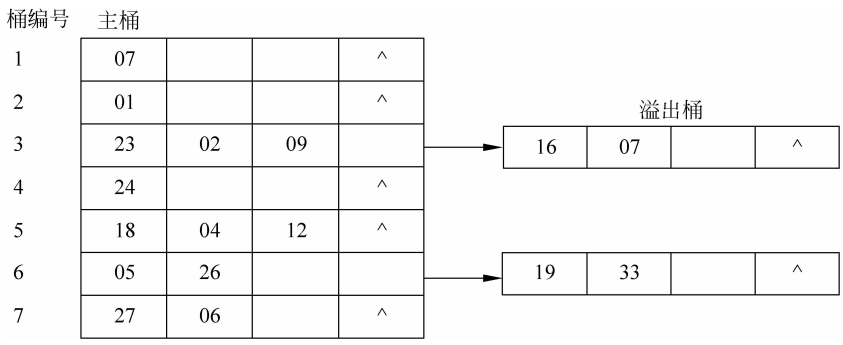

图 3-6　散列结构的溢出链示意图

4. 散列方法

下面介绍 3 种常用的散列方法。

(1) 静态散列方法

静态散列方法采用固定个数的散列桶,即把文件划分为 N 个散列桶,每个散列桶对应

一个磁盘块，每个散列桶有一个编号。为了实现散列桶编号到磁盘块地址的映射，每个散列文件具有一个散列桶目录。第 i 号散列桶的目录项存储该散列桶对应的磁盘块地址。图 3-7 给出了散列桶目录示例。

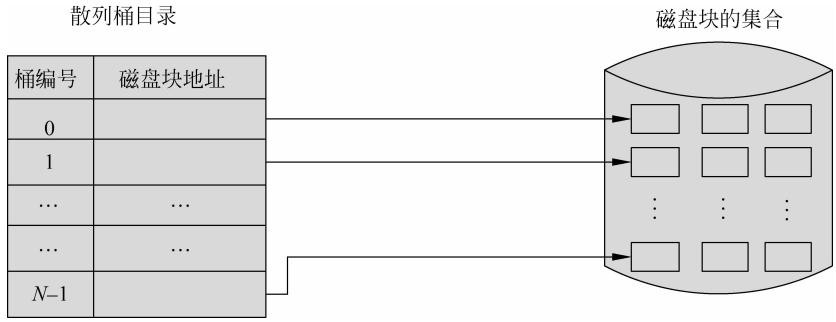

图 3-7　散列桶目录示例

显然，每个散列桶对应的磁盘块存储具有相同散列函数值的记录。如果文件的数据在散列属性上分布不均匀，可能产生桶溢出问题，其处理过程采用上述方法完成。查找记录时分两种情况处理：如果在散列文件上进行形如“$A=a$”的查询，其中，A 为散列域，a 为常数，则先计算 $H(A)$，得到桶号 i，查阅桶目录找到该桶的磁盘块链中第一个磁盘块的地址，并顺着链扫描检查每一块，直到找到满足条件的记录或证明没有满足条件的记录；若上述 A 不是散列域，则需要按无序文件的查找方法完成检索操作。

插入记录可以按如下方法处理：使用散列函数计算插入记录的桶号 i，在 i 桶的磁盘块链上寻找空闲空间，将记录存入。若桶上所有块均无空闲空间，此时向系统申请一个磁盘块，将新记录插入，并将新块链入 i 桶的磁盘块链。

散列文件上的删除操作也分两种情况。如果已知欲删记录的散列域，则运行散列函数计算欲删记录的桶号 i，在 i 桶的磁盘块链上找到欲删除记录，将其删除；如果不知道欲删除记录的散列域，则需要使用无序文件记录删除的方法处理记录。删除记录后，如果当前磁盘块为空，则释放该磁盘块。

修改记录时，若欲修改的是散列域，则通过先查询，再删除，后插入来完成；如果修改非散列域，则先进行查找，将记录读入内存缓冲区，在缓冲区中完成修改，并写回磁盘。

静态散列方法也有不足。第一，只能有效地支持散列域上具有相等比较的数据操作。如果数据操作的条件不是建立在散列域上或不是相等比较，则数据操作的处理时间与无序文件相同。第二，大多数数据库都会随时间而变大。由于散列桶的数量一成不变，当文件记录较少时，将浪费大量存储空间；当文件记录超过一定数量以后，磁盘块链将会很长，影响记录的存取效率。

（2）动态散列方法

动态散列方法在桶容量的自适应性和查找效率方面具有更好的效果。动态散列方法可以通过桶的分裂或合并来适应数据库的大小变化，此时散列桶的数量不是固定的，而是随文件记录的变化而增加或减少的。初始，散列文件只有一个散列桶，当记录增加，这个散列桶溢出时，它被划分为两个散列桶，原散列桶中的记录也被分为两部分。散列值的第一位为

1 的记录被分配到一个散列桶，散列值的第一位为 0 的记录被分配到另一个散列桶，当散列桶再次溢出时，每个散列桶又按上述规则划分为两个散列桶。动态散列方法需要一个用二叉树表示的目录，图 3-8 就是动态散列方法的结构。

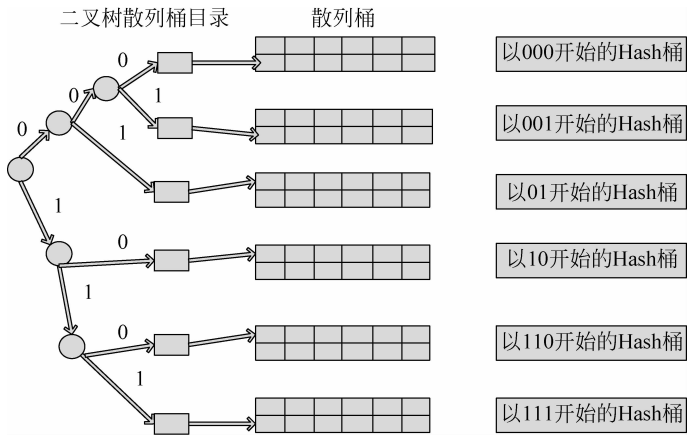

图 3-8　动态散列方法的结构

使用图 3-8 所示的散列桶，当插入一个具有 01 开始的散列值记录时，这个记录便被插入第三个散列桶，产生溢出，这时，第三个散列桶被划分为两个散列桶，散列值为 010 开始的记录被存储到划分出来的第一个散列桶，散列值为 011 开始的记录被存储到划分出来的第二个散列桶。当两个相邻散列桶中记录的总数不超过一个磁盘块容量时，可以将这两个散列桶合并为一个散列桶。

二叉树目录的级数随散列桶的分裂与合并而增加或减少。如果选定的散列函数能够把记录均匀地分布到各个散列桶，二叉树目录将是一个平衡的二叉树。

(3) 可扩展散列方法

可扩展散列方法的散列桶目录是一个包含 2^d 个磁盘块地址的一维数组，其中，d 称为散列桶目录的全局深度。设 $H(r)$ 是记录 r 的散列函数值，$H(r)$ 的前 d 位确定了 r 所在的散列桶编号。每个散列桶对应的磁盘块都有局部深度 d'，d' 是确定散列桶依赖的散列函数值的位数。图 3-9 给出了可扩展散列方法的结构。

下面用例子来说明散列桶的分裂过程。一个记录插入到第 01 号散列桶对应的第三个磁盘块，这时该磁盘块溢出，划分为两块，对应的散列桶也划分为两个散列桶。散列值的前三位为 010 开始的记录被存储到划分出来的第一个散列桶对应的磁盘块，散列值的前三位为 011 开始的记录被存储到划分出来的第二个散列桶对应的磁盘块。现在，散列桶 010 和 011 对应的磁盘块不再相同。这两个散列桶的局部深度 d' 由 2 变为 3。

如果一个局部深度与全局深度相同的散列桶溢出，则散列桶目录的大小需要增加两倍，因为需要增加一位数值才能识别散列桶。例如，当散列值的前三位为 111 的散列桶溢出时，必然裂变出编号为 1110 和 1111 的两个新散列桶，于是散列桶目录的全局深度必须改为 4，即散列桶目录的大小增加两倍。

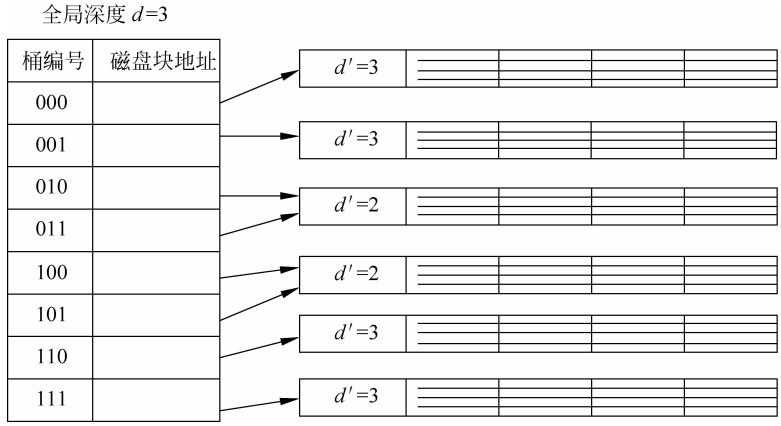

图 3-9　可扩展的散列方法的结构

3.3.2　磁盘型数据库的索引技术

1. B 树

在稀疏索引(Spare Index)方式中，当索引项很多时，可以将索引分块，建立高一级的索引；进一步还可以建立更高一级的索引……，直至最高一级的索引只占一个块为止。这种多级索引如图 3-10 所示，是一棵多级索引树。其中假设每个块可以存放 3 个索引项。

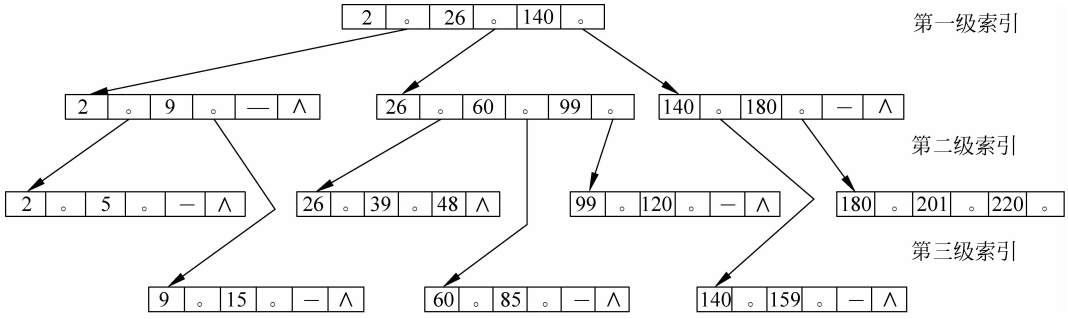

图 3-10　多级索引

当在多级索引上进行插入，使得第一级索引增长到一块容纳不下时，就可以再加一级索引，新加的一级索引是原第一级索引的索引。反之，在多级索引上进行删除操作会减少索引的级数，于是就产生了 B 树(平衡树)的概念。

B 树是 Bayer 和 McCreight 两人 1972 年在 *Organization and Maintenance of Large Ordered Indices* 一文中提出的，在数据库系统的存储组织中得到了广泛应用。B 树和下面将要介绍的 B^+ 树是树状数据结构的两个特例。为了更清楚地理解 B 树的概念，首先介绍将要用到的有关术语。

(1) 结点：包括一个数据元素及若干个指向其他子树的分支；例如 A、B、C、D 等。在 B 树中，将根结点、叶结点和内结点(B 树中除根结点和叶结点以外的结点)统称为结点。根结

点和内结点是存放索引项的存储块，简称为索引存储块或索引块。叶结点是存放记录索引项的存储块，简称为记录索引块或叶块，每个记录索引项包含关系中一个记录的关键值和地址指针。

(2) 树：树由结点组成。

(3) 子树：结点中每个地址指针指向一棵子树，即结点中的每个分支称为一棵子树。

(4) B 树的深度：每棵 B 树所包含的层数，包括叶结点，称为 B 树的深度。

(5) B 树的阶数：B 树结点中最大的指针数称为 B 树的阶数。

在上述术语的基础上，B 树定义如下。

满足如下条件的 B 树称为一棵 m 阶 B 树：

(1) 根结点或者至少有两个子树，或者本身为叶结点。

(2) 每个结点最多有 m 棵子树。

(3) 除根结点和叶子结点外，其他每个结点至少有 $m/2$ 棵子树。

(4) 从根结点到叶结点的每一条路径长度相等，也即树中所有叶结点处于同一层次上。

在上述定义基础上同时约定：

(1) 除叶结点之外的所有其他结点的索引块最多可存放 $m-1$ 个关键值和 m 个地址指针，其格式为：

P_0	K_1	P_1	K_2	P_2	…	K_{m-1}	P_{m-1}

其中，$K_i(1\leqslant i\leqslant m-1)$为关键值，$P_i(1\leqslant i\leqslant m-1)$为指向第 i 个子树的地址指针。为了节省空间，每个索引块的第一个索引项不包含关键值，但它包含比第二个索引项的关键值小的所有可能的数据记录。

(2) 叶结点上不包含数据记录本身，而是由记录索引项组成的记录索引块，每个记录索引项包含关键值和地址指针。每个叶结点中的记录索引项按其关键值大小从左到右顺序排列。每个叶结点最多可存放 m 个记录索引项，其格式为：

K_1	P_1	K_2	P_2	…	K_n	P_n

叶结点到数据记录之间的索引可以是稠密索引：每个记录索引项的地址指针指向一个数据记录，这时，$K_i(1\leqslant i\leqslant n)$为第 i 数据记录的关键值，P_i为指向第 i 个数据记录的地址指针；也可以是稀疏索引：每个记录索引项的地址指针指向包含该记录索引项的关键值所在块的起始地址，这时，$K_i(1\leqslant i\leqslant n)$为第 i 个记录块的最大关键值，P_i为指向第 i 个记录块的起始地址指针。

通常，为了表述方便，许多文献将叶结点的格式定义为：

K_1	K_2	K_3	…	K_n

其中省略了记录索引项的地址指针。但应注意，这仅仅是为了便于描述，在记录索引项中必须有地址指针。本书在 B 树和 B^+ 树的图示中，仍采用了这种方法。

(3) 假设每一个索引块能容纳的索引项数是个奇数，且 $m=2d-1\geqslant 3$；每一个记录索引块能容纳的记录索引项也是个奇数，且 $n=2e-1\geqslant 3$。

图 3-11 是图 3-10 中多级索引结构的 B 树表示方法，该 B 树是一个 3 阶 B 树。

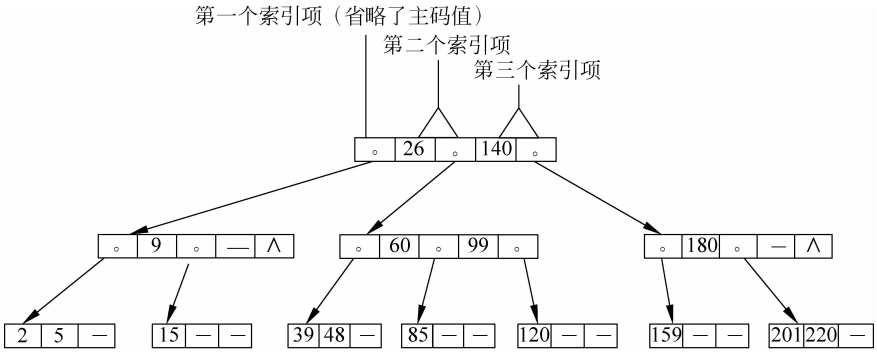

图 3-11　多级索引的 B 树

由图 3-11 可知，B 树中的关键值分布在各个索引层上。根结点和内结点中的索引项有两个作用：一是标识搜索的路径，起路标作用；二是标识关键值所属数据记录的位置，由其关键值即可指出该关键值所属记录的位置，这在 B 树中没有标出。

2. B^+ 树

(1) 树的概念

为了提高索引的查询效率，人们希望在保留 B 树基本特性的基础上，增加查询的灵活性，于是就提出了一种基于 B 树结构，又可同时实现随机查询和顺序查询两种检索方式的 B^+ 树模型。B^+ 树索引结构使用很广泛，在插入和删除数据时仍能保持其执行效率。

比较图 3-10 的多级索引结构和图 3-11 的 B 树可知，出现在 B 树中除叶结点以外的结点上的关键值不再出现在叶结点中，这样显然无法实现顺序查询。B^+ 树对此进行了改进，让树中所有索引项按其关键值的递增顺序从左到右都出现在叶结点上，并用指针链把所有叶结点链接起来。这样就实现了通过索引树的随机检索和通过叶结点链的顺序检索。图 3-12 给出了 B^+ 树的模型表示形式，它的上面是一棵 B 树，由存放各级索引的索引块组成；下面是所有叶结点组成的一个顺序集，由存放记录索引项的记录索引块组成。

在 B^+ 树中，由于出现在 B 树索引中的关键值均要出现在叶结点中，所以 B 树索引中的索引项就仅起路标作用。也就是说，由于 B^+ 树中关键值所属的数据记录的位置直到叶结点才给出来，所以在查询时，即使在非叶结点上找到了与给定值相等的关键值，也必须继续向下直到叶结点为止。

B^+ 树基本上遵从 B 树的定义和约定。而一棵 m 阶的 B^+ 树与一棵 m 阶的 B 树的区别在于：

① 在 B^+ 树的叶结点中包含了 B^+ 树中的全部关键值，且其中的所有索引项按其关键值的递增顺序从左到右顺序链接；而在 B 树中，由于关键值分布在各个索引层上，所以叶结点中没有包含 B 树中的全部关键值，且各叶结点间的关键值没有顺序链接。

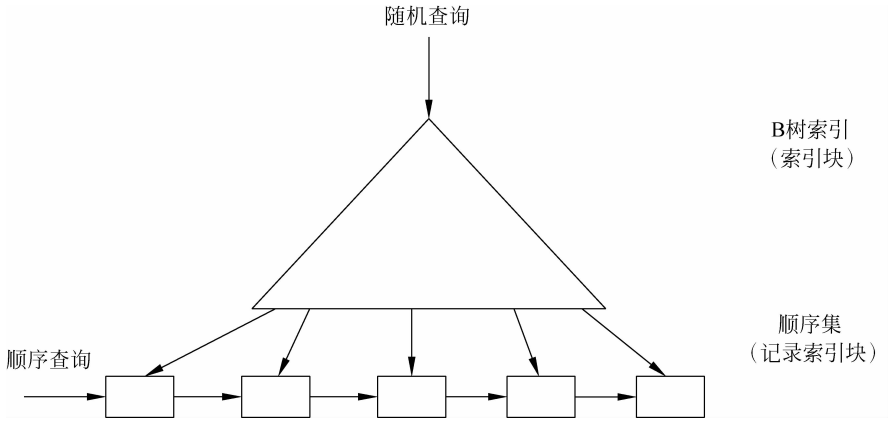

图 3-12　B^+ 树的模型表示

② 在 B^+ 树中，所有非叶结点包含了其子树中的最大(或最小)关键值；而在 B 树中，非叶结点中的关键值不再出现在其子树中。

③ 在 B^+ 树中，查询任何数据记录所经历的路径是等长的；而在 B 树中，不同数据记录的查询路径是不等长的。

④ 在 B^+ 树中可以采用两种方式进行查询，当随机查询时，通过 B 树索引找到要查找的数据记录，从根部开始找；当顺序查询时，通过顺序集找到要查找的数据记录，从顺序集的链头或通过 B 树索引得到某一顺序结点并开始查找。而在 B 树中，只有从根部随机查找的一种方式。图 3-13 是图 3-11 中 B 树的 B^+ 树表示。

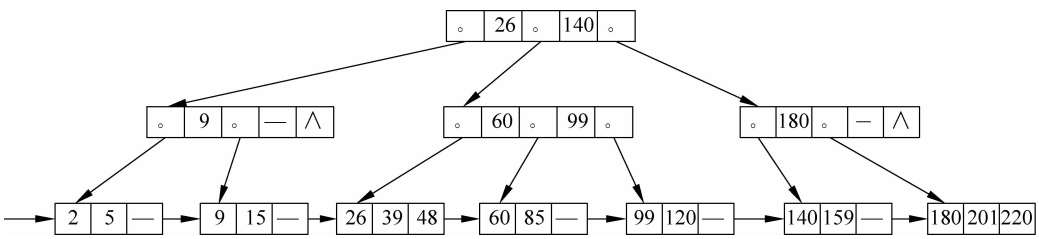

图 3-13　图 3-11 中 B 树的 B^+ 树

(2) B^+ 树的操作

B^+ 树的操作包括查找、修改、插入和删除。为了描述方便，下面的介绍中假设数据记录具有关键值 K。索引块的格式为：

P_0	K_1	P_1	K_2	P_2	…	K_{m-1}	P_{m-1}

其中，$K_i(1 \leqslant i \leqslant m-1)$为关键值，$P_i(1 \leqslant i \leqslant m-1)$为指向第 i 个子树的地址指针。

叶结点的格式为：

K_1	P_1	K_2	P_2	…	K_n	P_n

其中，K_i（$1 \leqslant i \leqslant n$）为第 i 个数据记录的关键值，P_i 为指向第 i 个数据记录的地址指针。

① 查找

通常在 B^+ 树上有两个头指针（root，seq），前者指向根结点，后者指向具有最小关键值记录索引项的叶结点的第一个记录索引项。

以随机查找方式查找具有关键值 K 的数据记录，就是要找一条从根结点到叶结点的路径。当从根结点开始查找已到达某个非叶结点时，需要将关键值 K 与该结点中的 K_1、K_2、…、K_{m-1} 进行比较：

- $K \leqslant K_1$ 时，进入由指针 P_0 指向的子树继续进行查找。
- $K > K_{m-1}$ 时，进入由指针 P_{m-1} 指向的子树继续进行查找。
- $K_i < K \leqslant K_{i+1}$，（$i=1,2,\cdots,m-2$）时，进入由指针 P_i 指向的子树继续进行查找。

当到达叶结点时，就可以在该叶结点中顺序查找要找的关键值。当找到某个 K_i 且有 $K=K_i$ 时，说明已经在叶结点中找到了具有关键值 K 的记录索引项。至于找数据记录的具体方法因数据组织方式不同（稠密索引或稀疏索引）而异。当在该叶结点中没有找到与 K 相等的关键值，也即在 B^+ 树索引中没有找到关键值 K 时，若叶结点到数据记录之间的索引采用的是稠密索引，则不存在关键值为 K 的数据记录；若叶结点到数据记录之间的索引采用的是稀疏索引，则不能立即确定是否存在关键值为 K 的数据记录，还必须在该数据记录块中继续查找后才能确定。

在按顺序查找方式查找具有关键值 K 的数据记录时，如果要查找全部叶结点的记录索引项，则可以从顺序集的链头开始顺序查找；如果是从要求的某个记录索引项开始查找，则可以从树根开始，以随机查找的方法找到所要求的记录索引项后，再从该记录索引项开始顺序查找。

② 修改

当要修改某具有关键值 K 的数据记录时，首先找到待修改的数据记录在叶结点中的记录索引项，然后按具体的存储组织方式修改数据记录。若修改的内容中包括要修改的数据记录的关键值，由于数据记录的关键值不能修改，所以这种修改实质上是一个删除和插入过程，即先从数据记录块中删除该数据记录，然后再通过重新插入（输入）达到修改目的。若修改的内容中不包括要修改的数据记录的关键值，则只需在修改该数据记录的非关键值字段的有关内容后进行该数据记录的重写。

③ 插入

为了插入具有关键值 K 的数据记录，首先要找到关键值 K 应当插入的叶结点 B。如果 B 中的已有记录索引项数小于 $n=2e-1$，则将 K 插入 B 中，并保持该叶结点中关键值的顺序排序。其中，与关键值对应的地址指针是按记录数据块的存储组织方式由插入该数据记录的位置决定。

如果 B 中已有 $n=2e-1$ 个记录索引项，则把 B 中记录索引项的关键值与新插数据记录的关键值 K 共 $2e-1+1=2e$ 个按递增顺序排序，并分成两组，每组 e 个，并新建一个记录索引块 B_1，把前面 e 个记录索引项放到块 B 中，后面 e 个记录索引项放到块 B_1 中（称为分裂）。同时，要把 B_1 的索引项插入到块 B 的父索引块中位于指向块 B 的索引项的右边。值得注意的是，如果从 B 的父结点开始向上的许多祖先结点都已装满 $m=2d-1$ 个索引项，则

在 B 中插入一个记录索引项后，会引起它的许多祖先结点分裂，这种过程有可能一直进行到根结点，这时，B^+ 树就增高了一层。

如果在 B 中发现有与 K 相等的关键值，则提示该数据记录已经存在。

如果给图 3-13 的 B^+ 树插入关键值为 41 的数据记录，则可得到如图 3-14 所示的 B^+ 树。其中，由于将关键值为 41 的记录索引项插入图 3-14 的第三个叶结点后，引起该叶结点的分裂，即增加了一个叶结点，由此引起了其所有祖先索引结点的分裂，使得 B^+ 树增高了一层。为了简化描述，图中省略了顺序集中各叶结点之间的横向顺序链。

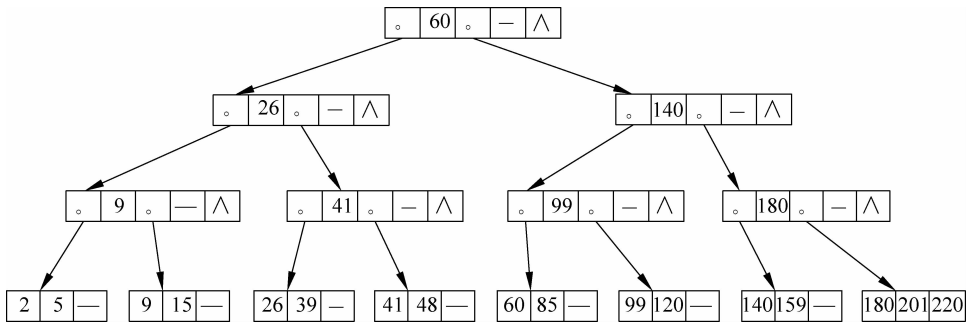

图 3-14　在图 3-13 中插入值为 41 数据记录后的 B^+ 树

④ 删除

为了删除具有关键值 K 的数据记录，首先要找到关键值 K 所在的叶结点 B。

第一种情况，如果 B 中的记录索引项数多于 e，则删除关键值为 K 的记录索引项后，B 中剩余的记录索引项个数仍不少于 e，可以进行删除操作：

若关键值为 K 的记录索引项不是 B 中的第一个记录索引项（例如，若删除的是图 3-13 中的第三个记录索引项中的 39，或第七个记录索引项中的 220），则删除该记录索引项后，操作结束。

若关键值为 K 的记录索引项是 B 中的第一个记录索引项，则要判断 B 是否是其父结点的最左一个孩子：若不是，则在删除该记录索引项后，将 B 的父结点中原指向 B 的那个索引项的关键值改为 B 中原第二个记录索引项的关键值（例如，若删除的是图 3-13 中的第七个记录索块的第一个记录索引项 180，则在删除该记录索引项后，将该记录索引块的父结点中的 180 改为 201），操作结束。

若关键值为 K 的记录索引项是 B 中的第一个记录索引项，且 B 是其父结点的最左一个孩子，则在删除该记录索引项后，要由 B 的父结点是否是 B 的父结点的父结点的最左一个孩子决定：若不是，则按前一种类似情况修改 B 的父结点的父结点中原指向 B 的父结点的那个索引项的关键值；若是，再向更高一层递归……直至根结点为止。

第二种情况，如果 B 中的记录索引项数等于 e，则删除关键值为 K 的记录索引项后，B 中剩余的记录索引项个数只有 $e-1$。此时，B 中的记录索引项数不到一半，根据 B^+ 树的定义，这时的 B 不能再作为树中的结点存在。于是就要通过 B 的父结点找到与 B 相邻的左孪生结点或右孪生结点 B_1，将 B 中剩余结点合并到 B_1 结点，或与 B_1 合并后再分裂成两个新结点，并修改相应的索引项。如前所述，也可能涉及其祖先。

如果从图 3-13 的 B⁺树删除关键值为 26 的数据记录，则可得到如图 3-15 所示的 B⁺树。其中，由于要删除树中第三个叶结点的最左边的关键值为 26 的记录索引项，所以引起对其父结点的父结点（图中为根结点）的相应索引项关键值的修改。

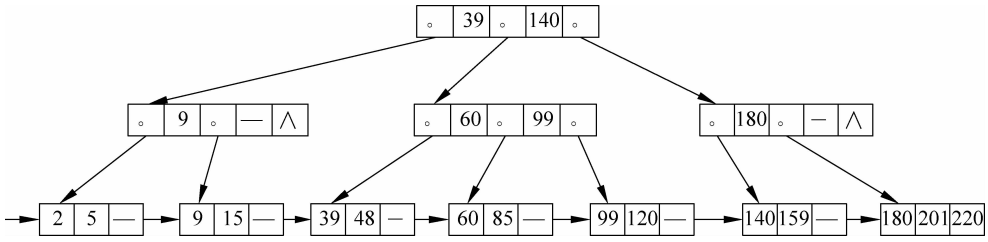

图 3-15　在图 3-13 中删除关键值为 26 的数据记录后的 B⁺树

（3）B⁺树的性能分析

设有 N 个数据记录的关系被组织成具有参数 $m=2d-1$ 和 $n=2e-1$ 的 B+树，其中，m 是非叶结点中索引项的个数，n 是叶结点中记录索引项的个数。显然，树中的叶结点不会超过 N/n，叶结点的父结点不会超过 $N/(m \cdot n)$，叶结点的父结点的父结点不会超过 $N/(m^2 \cdot n)$……可这样一直推算到树根。各层次与树中结点数关系如表 3-3 所示。

表 3-3　B⁺树的层次与树中结点数的关系

层　　号	非叶结点数	叶结点数	数据记录（关键值）数
1	0	1	n
2	1	m	$m\,n$
3	$m+1$	m^2	$m^2\,n$
4	m^2+m+1	m^3	$m^3\,n$
⋮	⋮	⋮	⋮
i	$m^{i-2}+m^{i-1}+\cdots+m+1$	m^{i-1}	$m^{i-1}n$

显然有：

$$N = m^{i-1}n$$

$$i = \log_m\left(\frac{N}{n}\right)+1$$

当每个结点均装满时：

$$i = \log_{2d-1}\left(\frac{N}{2e-1}\right)+1$$

当每个结点均只装到其下限时：

$$i = \log_d\left(\frac{N}{e}\right)+1$$

所以 B⁺树的层次取值范围为：

$$\log_{2d-1}\left(\frac{N}{2e-1}\right)+1 \leqslant i \leqslant \log_d\left(\frac{N}{e}\right)+1$$

例如，若 $N=20\,000$，取 $d=e=100$，则有 $2<i<3$，也即此种情况下 B⁺树的高度最大为 3，搜索代价较小，所以 B⁺树在数据库系统中得到了广泛应用。

B^+树除了搜索代价较小外，其突出的优越性是它较好地解决了数据记录在插入、删除和未用回收等存储组织问题，B^+树在操作中可动态地进行维护，可通过压缩索引项的办法来降低树的高度，减少读块次数，还可独立于具体的存储设备，并充分利用操作系统的分页技术。

3.4 内存型数据库

近年来，数据库系统在各种领域中扮演了关键角色，但传统的基于磁盘的关系型数据库系统却不能满足上述应用高性能、硬实时/软实时数据访问的要求，内存型数据库系统则可以很好地满足各种应用系统的实时数据管理需求，常见的内存型数据库基本架构如图 3-16 所示。在图 3-16 中，应用程序访问数据的时候，通过数据库提供的查询接口，查询一定的数据组织结构(如 B 树)，访问已经在内存中以一定方式(如表)存放的数据。在数据库运行的时候，一次性读取数据到内存中，或者按照一定的策略读取数据到内存中供应用程序使用。本节主要介绍了内存型数据库的基本概念，并将它和传统的磁盘型数据库进行了比较，然后分析了它在内存中的数据组织方式、数据库记录和内存之间的映射关系，以及内存受限时的数据库装入策略，最后介绍了典型的内存型数据库及其应用。

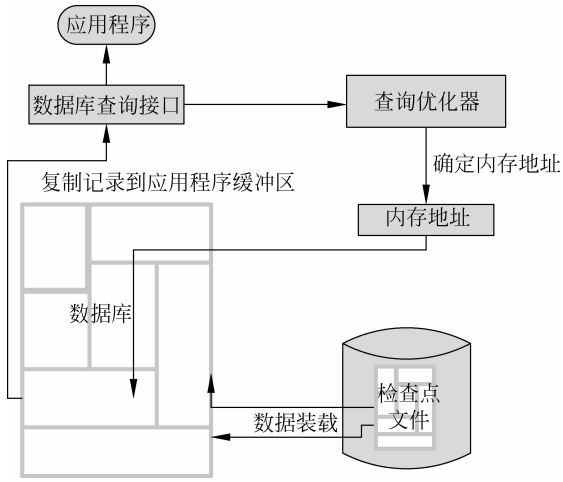

图 3-16　内存型数据库的基本架构

3.4.1 内存型数据库概述

随着电子技术的快速发展，内存已越来越便宜，这使得计算机上配置的内存容量变得越来越大。现在一些商用的系统已配置几 GB 甚至更多的内存。另外，随着计算机及操作系统从 32 位向 64 位的发展，使理论上计算机可配置的内存总数达 TB(Terabyte，万亿字节)级。从前，利用虚拟内存或内存交换技术使大于地址空间或大于物理内存的程序可以运行，然而，在嵌入式系统中的主要目的是如何充分利用大内存，使程序运行更快。

随着计算机应用领域不断扩大和应用程度不断加深，人们对数据库技术提出了新的更高的要求。内存数据库技术随着存储技术的发展和现代应用的高性能需求产生和发展起来。内存型数据库管理系统把数据全部或部分驻留在主存中，消除了传统的磁盘型数据库系统中事务运行的I/O瓶颈，从而获得直接访问数据的极高存取速度，大大提高了系统的性能，为需要快速响应和高吞吐量的应用提供了强有力的支持。相对于磁盘，内存的数据读写速度要高出几个数量级，将数据保存在内存中相比从磁盘上访问能够显著提高应用的运行速度。同时，内存数据库抛弃了磁盘数据管理的传统方式，基于全部数据都在内存中重新设计了体系结构，并且在数据缓存、快速算法、并行操作方面也进行了相应的改进，所以数据处理速度比传统数据库的数据处理速度要快得多。

从工作原理上看，内存型数据库与磁盘型数据库之间的主要区别在于：内存型数据库的主数据库常驻内存，体系结构设计的优化目标是提高内存和CPU使用效率。由于事务处理无须进行磁盘访问，使用内存数据库的应用系统性能得到极大提高。

在20世纪60年代末到80年代初，出现了内存数据库的雏形。1969年IBM公司International Business Machines Corporation，国际商业机器公司研制了世界上最早的数据库管理系统——基于层次模型的数据库管理系统(Information Management System，IMS)，并作为商品化软件投入市场。在设计IMS时，IBM考虑到基于内存的数据管理方法，相应推出了IMS/VS Fast Path。Fast Path是一个支持内存驻留数据的商业化数据库，但同时可以很好地支持磁盘驻留数据。在这个产品中体现了内存数据库的主要设计思想，也就是将需要频繁访问、要求高响应速度的数据直接存放在物理内存中访问和管理。在这个阶段中，包括网状数据库、关系数据库等其他各种数据库技术也都逐渐成型。

1984年，D. J. DeWitt等人发表了《内存数据库系统的实现技术》一文，第一次提出了内存型数据库的概念。该文章预言当时异常昂贵的计算机主存价格一定会下降，用户有可能将大容量的数据库全部保存在主存中，并提出了AVL树(Adel'son-Vel'skii和Landis树，即平衡二叉查找树)、哈希算法、内存数据库恢复机制等内存数据库技术的关键理论，为内存数据库的发展指出了明确的方向。

在内存数据库研究方面，1985年IBM推出了IBM 370上运行的OBE(Office By Example)内存数据库；1986年，R. B. Hagman提出了使用检查点技术实现内存数据库的恢复机制，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；1987年，ACM SIGMOD会议中提出了以堆文件(Heap File)作为内存数据库的数据存储结构，Southern Methodist大学设计出MARS(MAin memory Recoverabce database with Stable log)内存数据库模型；1988年普林斯顿大学设计出TPK内存数据库；1990年普林斯顿大学又设计出System M内存数据库。

随着互联网的发展，越来越多的网络应用系统需要能够支持大用户量并发访问、高响应速度的数据库系统，内存数据库市场成熟，半导体技术快速发展，半导体内存大规模生产，动态随机存取存储器(DRAM)的容量越来越大，而价格越来越低，这无疑为计算机内存的不断扩大提供了硬件基础，使得内存数据库的技术可行性逐步成熟。

1994年美国OSE公司推出了第一个商业化的、实际应用的内存数据库产品

Polyhedra；1998 年德国 SoftwareAG 公司推出了 Tamino DataBase；1999 年日本 UBIT 会社开发出 XDB 内存数据库产品，韩国 Altibase 公司推出 Altibase；2000 年奥地利 QuiLogic 公司推出了 SQL-IMDB；2001 年美国 McObject 公司推出 eXtremeDB，加拿大 Empress 公司推出 EmpressDB。

1. 内存型数据库的特点

内存系统和磁盘系统具有不同的特性，这是引起 MMDBS（Main Memory DataBase System，内存型数据库系统）和 DRDBS（Disk Resident DataBase System，磁盘型数据库系统）之间差别的根本原因，它们之间的区别主要表现在下列方面：

（1）速度不同。内存和磁盘在存取时间上有若干数量级的差别，内存数据库“工作版本”常驻内存，数据直接被访问，因而其系统“瓶颈”主要是内存空间和处理机的有效利用。据报道，Oracle 发布内存型数据库 TimesTen 的速度比传统产品快 10 倍。

（2）持久性不同。内存是易失性的，而磁盘是永久性的存储器，即当系统断电时，前者所存信息立即消失，通电后也不会恢复；而后者反之，断电时信息不会消失，再通电时即可继续使用。

（3）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。

（4）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多少差异，而对磁盘则几乎有数量级的差别。

（5）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导致的数据破坏。

这些差异影响到数据库管理的诸多方面，进而形成 MMDBS 自己的技术特征。

2. 内存型数据库的原理

传统的磁盘数据库都是基于磁盘的，即预先假定数据主要放在磁盘中，所以它的所有优化、查询算法都是以磁盘存储为主。举个简单的例子，比如说要查找一行记录，传统的数据库要先查找索引，通过索引查找该记录所在的页面，然后查找该页是已经在内存中，还是要从磁盘的数据文件中读取出来；而内存数据库是预先把所有的数据装入到内存中，所有的数据存储在内存里面，不会再通过其他的调用去决定数据在哪儿，这就少了很多环节，基本没有磁盘的 I/O，而且数据都在内存中，效率也就高了很多。内存数据库在物理数据组织上抛弃了磁盘数据管理的传统方式，基于全部数据都在内存中重新设计了体系结构，体系结构的设计优化目标是提高内存和 CPU 使用效率，所以处理数据速度要比传统的处理速度快得多。而从逻辑角度上，内存数据库相当于是磁盘数据库的映射，所以需要有一定的数据换入换出策略。

3.4.2 内存型数据库的数据组织方式

在传统的 DRDBS 中，数据以文件形式组织于磁盘上，其系统瓶颈是内外存的数据

I/O，因而其物理数据库组织应尽可能地减少对磁盘的存取次数。而在 MMDBS 中，“工作版本”常驻内存，数据直接被 CPU 访问，因而其系统“瓶颈”主要是内存空间和处理机的有效利用。因此，必须开发全新的适合内存特性的数据组织方法。由于内存中顺序存取和随机存取同样高效，数据不必簇聚存放，甚至可以将每个元组的各属性值分散存放在内存中，而在元组中仅保留指向各属性值的指针，从而大大提高了空间利用率。

1. 区段式

在内存型关系数据库中往往使用区段式的数据组织结构，区段式数据组织结构将共享内存划分为若干个“分区”，每个分区存储关系数据库的一个关系。区段式数据组织结构如图 3-17 所示。每个分区又是由若干固定长度的“段（也称做‘页’）”组成，一个段往往是共享内存动态分配的一个单位。而数据库中具体的数据记录则保存在段中分配的一个记录块中。在采用区段式的数据组织结构的数据库中，一个记录的地址信息由一个三元组＜P，S，L＞标志，其中，P 是分区号，一般对应于一个关系表名；S 是段号，标志组成这个分区的具体的段；L 是段内记录槽号，主要保存了记录在段内的偏移和长度，用来在段内进行寻址，从而通过这个三元组可以唯一地定位一个记录的具体位置。

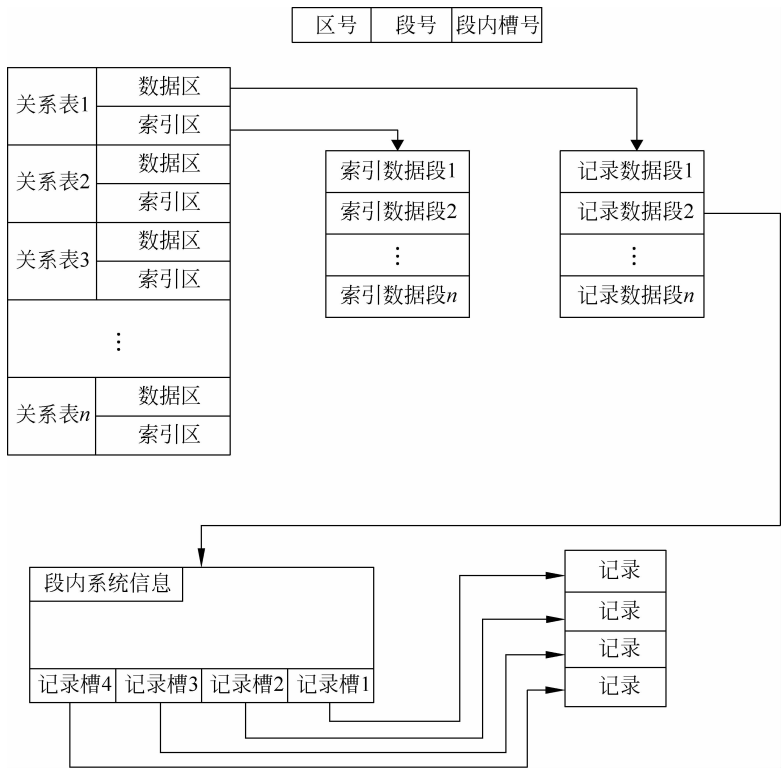

图 3-17　区段式数据组织结构示意图

2. 影子内存式

按影子内存式组织的内存数据库空间可以划分为两部分：一部分是 MMDB 的主数据

库(Primary DataBase,PDB)；另一部分是“影子内存”(Shadow Memory,SM)。影子内存式MMDB的体系结构如图3-18所示。

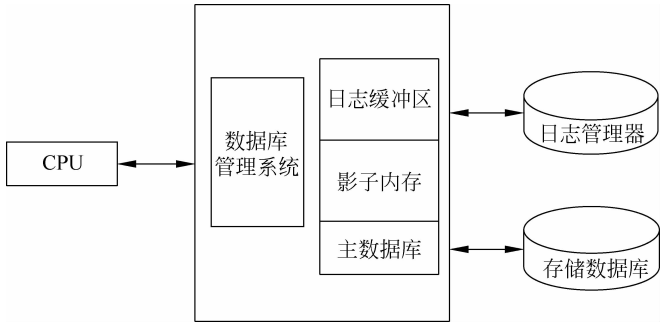

图3-18　影子内存式MMDB的体系结构

在事务的正常操作期间,每次查询都产生一个分别对应于影子内存SM和主数据库PDB的双地址,且总是先对SM查询,若不成功,再对PDB操作。所有的更新操作都在SM中进行,且都记录在活动日志中。每当一个事务提交时,由它所产生的SM中的“后映像”便复制到PDB中。

使用影子内存的优点是：

(1) 减少了日志缓冲区,因为其后映像区和用户区合二为一。

(2) 省去因事务失败或系统故障时的还原(Undo)操作,只清除相应的影子内存即可。

(3) 减少对MMDB(PDB)的存取,各事务可并行对各SM区操作。

(4) 缩短恢复过程,这是因为一方面如(2)所述,省去还原操作,只需做重做(Redo)型操作；另一方面还可以就当前事务对SM做“部分恢复”以后,先启动正常事务处理,然后按需要逐步恢复PDB。影子内存式和区-段式可以组合使用。

3. 哈希索引

哈希索引定义了一个哈希函数,通过将关系表的索引项传入到哈希函数可以计算出相应的哈希值,从而在索引项和哈希值之间建立起对应关系。同时用于保存不同的哈希值的索引信息首地址往往是线性结构,从而可以迅速地找到每个哈希值的首地址,使得通过哈希索引查找数据只需常数时间的复杂度。

哈希索引的示意图如图3-19所示。

由图3-19可知,索引项不同的具体数据使用哈希往往会得到相同的哈希值,所以一般为每个哈希值建立一个动态的冲突链表来保存同一哈希值的记录索引信息。当为一条记录建立索引时只需通过对索引项使用哈希函数得到其哈希值,通过计算得到的哈希值迅速找到保存此哈希值冲突链的首地址,并将这条记录的地址信息插入到冲突链表中。当需要通过这个索引项的一个特定值对记录查找时,只需对这个索引项的给定的值运用哈希函数求得哈希值,找到该哈希值冲突链的首地址,顺序遍历冲突链以找到待查找记录的地址信息。

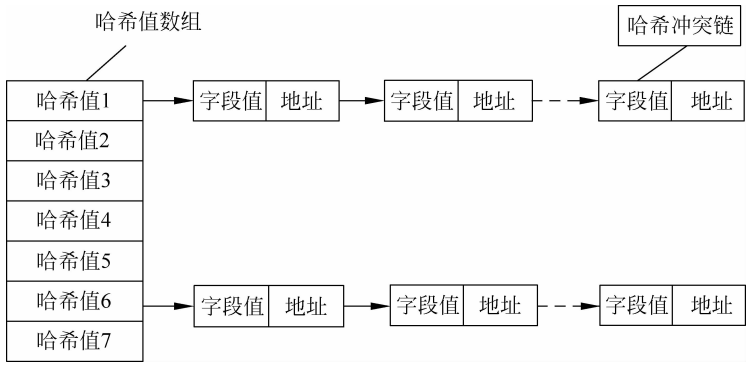

图 3-19　哈希索引示意图

4. T 树索引

索引用于在查询时提高效率之用。可以为数据表的某个字段定义索引来提高在该字段上的查询效率。由于数据库要处理的数据量非常大，而内存价格昂贵、容量有限，且必须满足一定的实时性，因而对其中的数据存储及索引方式进行研究，找出有效的数据组织方式是非常有必要的。磁盘数据库系统的典型索引技术是 B 树索引。B 树结构的主要目的是减少完成数据文件的索引查找所需要的磁盘 I/O 的数量。B 树通过控制结点内部的索引值达到这个目的，在结点中包含尽可能多的索引条目（增加一次磁盘 I/O 可以访问的索引条目）。B 树是比较适合于磁盘数据组织的数据结构，由于它是一个宽而浅的树，查找一个数需要访问很少的结点。大部分数据库系统用一个 B 树的变种 B^+ 树，将所有的数据保存在树的叶结点上。然而，对于内存数据库，B 树比 B^+ 树更合适，因为在内存中将所有的数据保存在叶子上太浪费空间。B 树的内存利用率是比较好的，所以用于内存数据库比较合适；搜索速度比较快（用二分查找时，只访问很少一部分结点）；而且更新速度也比较快（数据移动通常只涉及一个结点）。另一方面，T 树是针对主存访问优化的索引技术。T 树是一种一个结点中包含多个索引条目的平衡二叉树，T 树的索引项无论是从大小还是算法上都比 B 树精简得多。T 树的搜索算法无论搜索的值在当前的结点还是在内存中的其他位置，每访问到一个新的索引结点，索引的范围减少一半。

T 树索引用来实现关键字的范围查询。T 树是一棵特殊平衡的二叉树（AVL），它的每个结点存储了按键值排序的一组关键字。T 树除了较高的结点空间占有率，遍历一棵树的查找算法在复杂程度和执行时间上也占有优势。现在 T 树已经成为内存数据库中最主要的一种索引方式。T 树索引的示意图如图 3-20 所示。

T 树具有以下特点：

（1）左子树与右子树高度之差不超过 1。

（2）在一个存储结点可以保存多个键值。它的最左与最右键值分别为这个结点的最小与最大键值，它的左子树仅仅包含那些键值小于或等于最小键值的记录，同理，右子树只包含那些键值大于或等于最大键值的记录。

（3）为了保持空间的利用率，每一个内部结点都需要包含一个最小数目的键值。由此

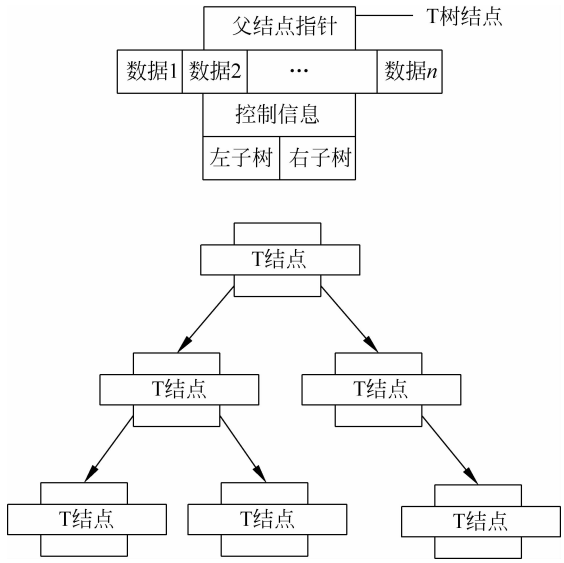

图 3-20　T 树索引示意图

可知 T 树是一个每个结点含有多个关键字的平衡二叉树，每个结点内的关键字有序排列，左子树的关键字比根结点的关键字小，右子树的关键字比根结点的关键字大。

T 树结点(T node)结构中包含如下信息：

(1) balance(平衡因子)，其绝对值不大于 1，balance ＝右子树高度－左子树高度。

(2) Left_child_ptr 和 Right_child_ptr，分别表示当前结点的左子树和右子树指针。

(3) Max_Item，表示结点中所能容纳的键值的最大数。

(4) Key[0]至 Key[Max_Item－1]，为结点内存放的关键字。

(5) nItem，是当前结点实际存储的关键字个数。

T 树与 AVL 树相比具有以下差异：

(1) 与 AVL 树相似，T 树中任何结点的左右子树的高度之差最大为 1。

(2) 与 AVL 树不同，T 树的结点中可存储多个键值，并且这些键值排列有序。

(3) T 树结点的左子树中容纳的键值不大于该结点中的最左键值；右子树中容纳的键值不小于该结点中的最右键值。

(4) 为了保证每个结点具有较高的空间占用率，每个内部结点所包含的键值数目必须不小于某个指定的值，通常为(Max_Item－2)。

用 T 树作为索引方式主要完成 3 个工作：查找、插入和删除。其中，插入和删除都是以查找为基础。下面分别介绍 3 种操作的流程。

(1) T 树的查找类似于二叉树，不同之处主要在于每一结点上的比较不是针对结点中的各个元素值，而是首先检查所要查找的目标键值是否包含在当前结点的最左键值和最右键值所确定的范围内。如果是的话，则在当前结点的键值列表中使用二分法进行查找；如果目标键值小于当前结点的最左键值，则类似地搜索当前结点的左孩子结点；如果目标键值大于当前结点的最右键值，则类似地搜索当前结点的右孩子结点。

(2) T 树的插入是以查找为基础,应用查找操作定位目标键值插入位置,并记下查找过程所遇到的最后结点。如果查找成功,判断此结点中是否有足够的存储空间。如果有,则将目标键值插入结点中;否则将目标键值插入此结点,然后将结点中的最左键值插入到它的左子树中(此时是递归插入操作)。如果查找失败,则分配新结点,并插入目标键值;然后根据目标键值与结点的最大最小键值之间的关系,将新分配的结点链接为结点的左孩子或右孩子;对树进行检查,判断 T 树的平衡因子是否满足条件,如果平衡因子不满足则执行旋转操作。

(3) T 树的删除操作是以查找为基础,应用查找操作定位目标键值。如果查找失败,则结束;否则令 N 为目标键值所在的结点,并从结点 N 中删除目标键值;删除结点后,如果结点 N 为空,则删除结点 N,并对树的平衡因子进行检查,判断是否需要执行旋转操作;如果结点 N 中的键值数量少于最小值,则根据 N 的平衡因子决定从结点 N 的左子树中移出最大的键值或者从右子树中移出最小值来填充。

T 树索引需要实现 T 树的查找、插入和删除。其中又以查找为基础,对 T 树的维护也以 T 树的旋转为关键。当由于插入或删除键值导致树的失衡时,则要进行 T 树的旋转,使之重新达到平衡。

在插入情况下,需要依次对所有沿着从新创建结点到根结点路径中的结点进行检查,直到出现如下两种情况之一时终止:某个被检查结点的两个子树高度相等,此时不需要执行旋转操作;某个被检查结点的两个子树的高度之差大于 1,此时对该结点仅需执行一次旋转操作。

在删除情况下,类似地需要依次对所有沿着从待删除结点的父结点到根结点路径中的结点进行检查,在检查过程中当发现某个结点的左右子树高度之差越界时,需要执行一次旋转操作。与插入操作不同的是,执行完旋转操作之后,检查过程不能中止,而是必须一直执行到检查完根结点。

由此可以看出,对于插入操作,最多只需要一次旋转操作即可使 T 树恢复到平衡状态;而对于删除操作,则可能会引起向上的连锁反应,使高层结点发生旋转,因而可能需要进行多次旋转操作。

为了对 T 树进行平衡,需要进行旋转操作,旋转是 T 树中最关键也是最困难的操作。旋转可分为 4 种情况:由左孩子的左子树的插入(或者删除)引起的旋转记为 LL 旋转,类似地有 LR、RR 及 RL 旋转。插入时的情况与删除类似。

5. N-Array 存储模型

所谓 N-Array 存储模型,就是将数据库关系表的记录在数据页面顺序存放,即将一条记录的所有字段的数据顺序存放在连续的空间。N-Array 存储模型示意图如图 3-21 所示。

N-Array 模型将整条记录顺序存放在数据页面中,通过一个包含记录页内偏移和记录长度的结构体在数据页面中定位记录。通过这种方式,N-Array 模型能够有效地使用数据页面的空余存储空间。但是使用 N-Array 存储模型时,进行数据查找每次找到的都是一整条记录,因而当只需查找关系表的某几个字段或对几个关系表进行联合查找时则会浪费过多的缓存空间,从而造成较大的开销,由此可见 N-Array 存储模型的缺点在于不能很好地

支持对缓存效率的要求。

当前主流内存数据库如 SQLite、fastDB、XtremDB 等都将记录以 N-Array 模型进行存储。

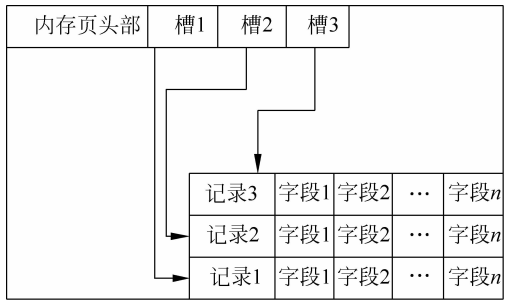

图 3-21　N-Array 存储模型示意图

3.4.3　数据库记录与内存的映射关系

因为内存数据库自己维持着大块 RAM 空间作为磁盘中数据的 Cache，所以其中就会涉及内存单元和磁盘数据的映射。假定磁盘 DBMS 管理的所有数据主要存在于磁盘中，记录的访问是通过 RID(Record IDentifier)实现的。在区段式组织方式中，每个段中的一个数据记录就是一个关系元组，每个记录都有一个唯一的标识 RID。RID 由三部分组成：分区号、段号以及段内记录槽号。记录槽包含了对应记录的长度和记录的首地址。这样通过 RID 找到相对应的记录槽，按槽中的地址和长度便可直接存取所要的记录。因此要访问一个记录，需要地址映射将 RID 转换为内存的物理地址。其实，对于内存数据库，存取方法返回的不必是所需的记录数据的副本，而只需是 RID。

3.4.4　内存受限时内存数据库装入策略

从某种角度上来看，MMDB 也是一种 Cache 机制，是磁盘数据库的“Cache”，通过物理内存中的数据存储区的直接操作，减少了与磁盘间的 I/O 交互。因为内存容量限制，无法将数据库所有的数据装入，所以在初始化数据装入时必须有一定的规则来保证装入的数据是最重要的。下面首先介绍数据的初始化策略。

内存数据库在初始化数据装入时，首先考虑的是事务的优先级，优先级高的事务先装入内存；其次是数据的流行性，流行数据对应的事务往往也是高优先事务；再次就是活跃性，存取频率高的数据一般是先要被存取的数据；紧密相关的数据则随时要考虑被使用。

由于内存数据库并不能容纳全部的外存数据库，因此内存数据库初始化的时候，需要选择装入最需要被装入的数据。但是在内存数据库初始化的时候，内存能容纳事务的数据是未知的，事务类型、开始时间、结束时间等标准的相互冲突，都给事务的选择带来

了难度。

首先需要把全部分析好的事务排成一个队列。内存数据库初始化的时候从队列的第一个事务开始装入，无论内存有多大，装入的数据都是最应该装入的数据。队列的形成策略顺序如下：

(1) 随机发生的硬实时事务。所谓硬实时事务，是指事务必须在一定的时间内，或者某个时间点前完成。由于硬实时事务如果超过了截止期限会给系统带来毁灭性的后果，因此随机发生的硬实时事务的数据应该最先被调入内存。

(2) 周期事务的数据。由于实时数据库的周期事务频繁地执行，如果不能保证数据在内存，就会发生“抖动”，严重影响系统的效率，因此周期事务的数据也应该被调入内存。

(3) 装入时间和开始时间已知的硬实时事务。设一个硬实时事务的开始时间 T_s 与当前时间 T_n 的差为 N，事务数据装入时间的最坏静态估计为 M(最坏情况下所有涉及的相关外存块数乘以读取一块的平均时间)，如果 N 远远大于 M，则该事务的数据现在可以不在内存中，相反，则硬实时事务的数据应该装入内存。

(4) 软实时事务的数据。软实时事务虽然具有截止期限的要求，但超过截止期限完成仍然是有意义的。如果在以上两类事务的装入过程中内存耗尽，则内存数据库虽然初始化成功，但是性能堪忧。以上的装入顺序保证无论内存有多大，装入的数据从事务类型来看都是最需要装入的数据，而且也为后来的数据换出及如何方便地换出最应该更换的数据打下了基础。

初始化完后，和 Cache 一样，还需要有数据的替换策略。内存数据库运行时的数据装入(数据的替换策略)策略如下：

(1) 内存数据库初始化成功以后，系统开始运行。如果接纳一个事务后发现其数据不在内存中，仍然要进行数据装入。数据交换策略必须考虑以下因素。

① 高易变的实时数据必须常驻内存中且不能被交换出去。

② 活跃或高频数据应留驻内存中，一般不应交换出去。

③ 立即运行的数据在第一个处理请求以前不能被交换出去。

④ 高优先级事务的数据在事务的活动期不能被交换出去，尤其当事务是周期性事务时，其数据应尽可能常驻内存中。

⑤ 非永久数据和关键数据最好不要换出。非永久数据无须换出，关键数据至关重要，要保证对它存取的及时性和有效性。

(2) 进行交换的数据单位通常是分区中的段。

(3) 为了让系统正常运行，尽量减少因数据装入而造成超过截止期限的事务数目增多。设定内存数据库运行时的数据装入时机有以下几种。

① 新接纳的事务数据不在内存中，且新接纳的事务比正在运行的事务有更高的优先级，则挂起正在运行的事务，启动数据换入。

② 原运行事务结束重新调度事务时，发现接纳队列的首元素比就绪队列的首元素优先级高，则启动数据换入。这要求接纳队列和就绪队列按照优先级排序。

3.4.5 典型的内存型数据库

典型的内存型数据库包括以下几种。

1. SQLite

SQLite 是一款轻型的数据库，是遵守 ACID 的关联式数据库管理系统，它的设计目标是嵌入式的，而且目前已经在很多嵌入式产品中使用。它占用资源非常低，在嵌入式设备中，只需要几百 KB 的内存就可以使用。它能够支持 Windows、Linux、UNIX 等主流的操作系统，同时能够和很多程序设计语言相结合，例如 TCL、C＃、PHP、Java 等，还有 ODBC 接口，比 MySQL、PostgreSQL 这两款世界著名的开源数据库管理系统处理速度都快。SQLite 第一个 Alpha 版本诞生于 2000 年 5 月，至 2012 年 10 月，SQLite 已经推出了 3.7.14 版。

2. eXtremeDB

eXtremeDB 实时数据库是 McObject 公司的一款专门为实时与嵌入式系统数据管理而设计的数据库，只有 50KB 到 130KB 的开销，速度达到微秒级。eXtremeDB 完全驻留在内存中，不使用文件系统(包括内存盘)。eXtremeDB 采用了新的磁盘融合技术，将内存拓展到磁盘，将磁盘当做虚拟内存来用，实时性能保持微秒级的同时，数据管理量在 32 位下能达到 20GB。

3. Oracle TimesTen

Oracle TimesTen 是 Oracle 从 TimesTen 公司收购的一个内存优化的关系数据库，它为应用程序提供了实时企业和行业(例如电信、资本市场和国防)所需的即时响应性和非常高的吞吐量。Oracle TimesTen 可作为高速缓存或嵌入式数据库被部署在应用程序层中，利用标准的 SQL 接口对完全位于物理内存中的数据存储区进行操作。

4. SolidDB

SolidDB 由 Solid Information Technology 开发，该公司成立于 1992 年，总部位于加州 Cupertino，Solid 数据管理平台将基于内存和磁盘的全事务处理数据库引擎、载体级高可用性及强大的数据复制功能紧密地融为一体。

5. Altibase

Altibase 由 ALTIBASE 公司开发。它适用于通信、网上银行、证券交易、实时应用和嵌入式系统领域。目前占据 80％以上内存数据库市场，是当今嵌入式数据库软件技术的领导者。目前，Altibase 在国内已经得到比较多的应用，尤其是在电信行业应用比较广泛。

3.4.6　内存型数据库的应用

现代通信对计算机技术依赖性越来越强，在电信运营商的不断发展中建立起各类信息系统，但是这些数据资料过于庞大，记录条数动辄以亿计。以中国移动为例，其全国移动电话用户总数已经超过三亿，许多省公司的用户规模都超过千万，对于如此庞大的数据量，传统的基于磁盘的数据库管理系统越来越难以应付，于是内存数据库在电信领域出现了。

目前在国内，中国移动、中国联通、中国电信、中国网通等主要运营商的不同业务系统，以计费为重点，包括 NGN、IN 系统等，已经广泛采用内存数据库技术。全球主要电信运营商的数据库管理系统正在经历由“磁盘数据库为主”向“内存数据库为主”的转变。

在提升数据库工作效率的同时，使用内存数据库可以直接节省硬件投资和系统维护成本。实际部署经验表明，为达到特定重载系统的实时性要求，使用内存数据库比不使用内存数据库所需的 CPU 数量减少 2/3 以上，而只需把内存数量少量提高，在大大节约经费资金投入的情况下，系统稳定性却得到提高。

内存数据库由于大量数据在内存中运行，没有过多的 I/O 操作，能较好地满足实时性、灵活性、精确性的要求，在电信领域得到大量使用。

由于电信行业处理的是巨量数据，而内存容量有限，故需磁盘来存放事务处理的数据和非实时性的数据，此过程涉及内存和磁盘之间接口 I/O 的数据交换。但实时数据不会涉及 I/O 交换，所以不会降低事务处理的效率，这也是内存数据库和传统数据库的主要区别。

用户通过前端页面直接查询内存数据库中的实时话费累计表。采用内存数据库后，大部分数据操作在内存中运行，极大地减少了 I/O 操作，减少了运行中因等待数据而消耗的时间，提高了计费系统的运行效率。同时在话费查询中，由于省去了以往内存中的数据和磁盘数据库数据同步的环节，用户可以通过营业部实时查询话费，比目前只能提供查询到前一天的实时话费在业务上有了质的飞跃。

本章小结

嵌入式数据库由于其存储的高效率、低空间和时间开销等方面的特殊要求，从而在存储介质和数据组织算法方面与传统的桌面数据库相比具有较大的差异。本章介绍了嵌入式数据库的物理存储层的结构，它通常具有 RAM、NVRAM、Flash 等存储介质，在逻辑组织方面，嵌入式数据库中的数据分为数据字典、索引和数据三类，在外存和内存中存放的时候，具有不同的数据结构。尽管目前嵌入式数据库中的存储介质多种多样，但 Flash 是目前最广泛使用的一种外存介质，它在读写方面具有与磁盘和 RAM 不同的特点，在使用过程中需要考虑磨损平衡、垃圾回收、坏块处理等问题，目前已经有 JFFS/JFFS2、YAFFS/YAFFS2、UBIFS 等文件系统支持 Flash 的管理。按照存储介质划分，嵌入式数据库分为磁盘型和内存型两种，磁盘型的数据库默认的数据访问位置在外存中，具有存储量大的优点，但访问速度稍慢，通常使用散列算法和索引算法如 B 树和 B^+ 树管理磁盘中存储的数据，目的是提高

访问的速度。内存型数据库默认的数据访问位置在内存中，具有访问速度快的优点，但要求内存容量比较大，通常使用区段式存储、T 树、N-Array 存储等方式，目的是提高内存利用率以及内存访问的速度。

习题 3

1. 嵌入式数据库的物理存储结构包括哪些层次？

2. NOR Flash 和 NAND Flash 在性能方面各具有什么特点？

3. 在散列文件组织中，是什么原因引起桶溢出的？有什么办法可以减少桶溢出？

4. 设查找键值集为{2,3,5,7,11,17,19,23,29,31}。假设初始时 B^+ 树为空，按升序次序插入键值。就下面 3 种情况建立 3 棵 B^+ 树：(1)二阶；(2)三阶；(3)四阶。

5. 内存型数据库和磁盘型数据库之间有什么主要区别？

6. 典型的内存型数据库包括哪些？简述其主要特点。