第4章 索引结构

前面介绍了表示记录的几种可选方式，现在我们需要考虑整个关系或类外延如何表示。仅仅把关系中的元组或类外延中的对象随机分散到各存储块中是不够的。为了说明这一点，我们来看一看怎样回答哪怕是像 SELECT * FROM这样最简单的查询。我们将不得不检索存储器中的每个存储块，并且还得依赖于：

1) 块首部中存在足够的信息来标明该块中记录从什么地方开始。

2) 记录首部中存在足够的信息来说明该记录属于哪个关系。

一个稍好的记录组织方式是为给定的关系预留一些块甚至几个完整的柱面。我们可以认为这些柱面上所有存储块中都存放表示该关系中元组的记录。现在，我们至少不需要扫描整个存储器就能找到该关系的所有元组。

然而，这种组织方式却无助于回答下面这个简单查询：“找出给定主键值的元组”。举例来说，*name*是图3-1所示关系MovieStar的主键。像

```
SELECT *
FROM MovieStar
WHERE name = 'Jim Carrey';
```

这样的查询需要扫描可能存放有 MovieStar元组的所有存储块。为了便于实现类似查询，通常给关系建立一个或多个索引。如图 4-1所示，索引是这样一种数据结构：它以记录的特征（通常是一个或多个字段的值）为输入，并能“快速地”找出具有该特征的记录。具体来说，索引使我们只需查看所有可能记录中的一小部分就能找到所需记录。建立索引的字段（组合）称为查找键，在索引不言而喻时也可称“键”。

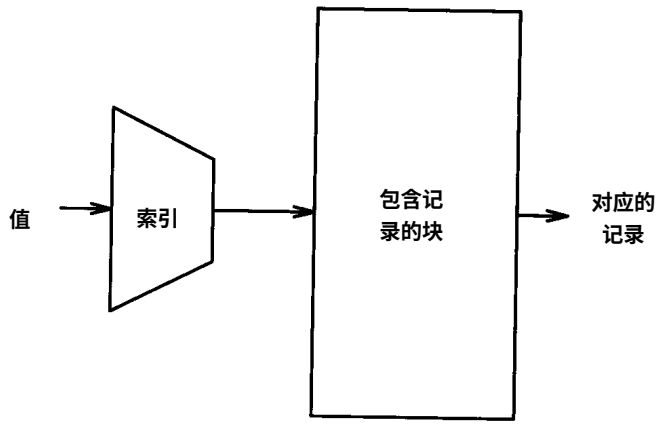

图4-1 索引以某个（或某些）字段值为输入并找出对应字段值符合要求的记录

目前已有多种不同数据结构可用做索引。在本章剩下的内容中，我们考虑下面几种设计和实现索引的方法：

1) 排序文件上的简单索引。
2) 非排序文件上的辅助索引。
3) B树，一种可在任何文件上建立索引的常用方法。
4) 散列表，另一种有用而重要的索引结构。

4.1 顺序文件上的索引

研究索引结构，我们首先来考虑最简单的一种：由一个称为数据文件的排序文件得到另一个称为索引文件的文件，而这个索引文件由键-指针对组成。在索引文件中查找键 K通过指针指向数据文件中查找键为 K的记录。索引可以是“稠密的”，即数据文件中每个记录在索引文件中都设有一个索引项；索引也可以是“稀疏的”，即数据文件中只有某些记录在索引文件中表示出来，通常为每个数据块在索引文件中设一个索引项。

4.1.1 顺序文件

一种最简单的索引结构要求文件按索引的属性排序，这样的文件称为顺序文件。当查找键是关系的主键时这种结构特别有用，当然对关系的其他属性也可使用这种结构。图 4-2所示为一个用顺序文件表示的关系。

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

图4-2 顺序文件

键以及几种不同的键

术语“键”有几个涵义，本书在适当的上下文中使用了“键”的不同涵义。你肯定对“键”用来表示“关系的主键”十分熟悉。这样的键用 SQL定义。且需要关系满足在主键属性或属性组上不存在值相同的两个元组。

在2.3.4节中我们学过“排序键”，文件的记录据此排序。现在来看“查找键”已知在该属性(组)上的值，需要通过索引查找具有相应值的元组。当“键”的涵义不清楚时，尽量使用恰当的修饰词(“主”、“排序”或“查找”)。不过请注意，在4.1.2节和4.1.3节中，很多时候这三种键是同一涵义。

在这个文件中，元组按主键排序。这里假定主健是整数，我们只列出了主键字段。同时，我们还做了一个不代表典型情况的假定，即每个存储块中只可存放两个记录。例如，文件的第一个块中存放键值为 10和20的两条记录。在这里和其他许多例子中，我们使用 10的连续倍数来做键值，虽然实际中不会要求键值都是 10的倍数或10的所有倍数都必须出现。

4.1.2 稠密索引

既然把记录排好了序，我们就可以在记录上建立稠密索引。它是这样的一系列存储块：块中只存放记录的键以及指向记录本身的指针，指针就是如 3.3节讨论的那样的地址。我们说这里的索引是“稠密的”，因为数据文件中每个键在索引中都被表示出来。与此相比，将在 4.3节中讨论的“稀疏”索引通常在索引中只为每个数据块存放一个键。

稠密索引文件中的索引块保持键的顺序与文件中的排序顺序一致。既然假定查找键和指针所占存储空间远小于记录本身，我们就可以认为存储索引文件比存储数据文件所需存储块要少得多。当内存容纳不下数据文件但能容纳下索引文件时，索引的优势尤为明显。这时，通过使用索引文件，我们每次查询只用一次I/O操作就能找到给定键值的记录。

例4.1　图4-3所示为一个建立在顺序文件上的稠密索引，该顺序文件的起始部分如图 4-2所示。为了方便起见，还是假定文件中记录的键值是 10的倍数，虽然实际中我们不能指望找到这样一个有规律的键值模式。我们还假定每个索引存储块只可存放四个键 -指针对。在实际中我们会发现每个存储块能存放更多的键-指针对，甚至是好几百个。

第一个索引块存放指向前四个记录的指针，第二个索引块存放指向接下来的四个记录的指针，依此类推。出于将在4.1.6节讨论的原因，实际中我们可能不希望装满所有的索引块。　□

稠密索引支持按给定键值查找相应记录的查询。给定一个键值 K，我们先在索引块中查找 K。当找到 K后，按照 K所对应的指针到数据文件中寻找相应的记录。似乎在找到 K之前我们需要检索索引文件的每个存储块，或平均一半的存储块。然而，由于有下面几个因素，基于索引的查找比它看起来更为有效：

1) 索引块数量通常比数据块数量少。
2) 由于键被排序，我们可以使用二分查找法来查找 K。若有 n个索引块，我们只需查找 $\log_2 n$ 个块。

3) 索引文件可能足够小，以致可以永久地存放在主存缓冲区中。要是这样的话，查找键 K时就只涉及主存访问而不需执行I/O操作。

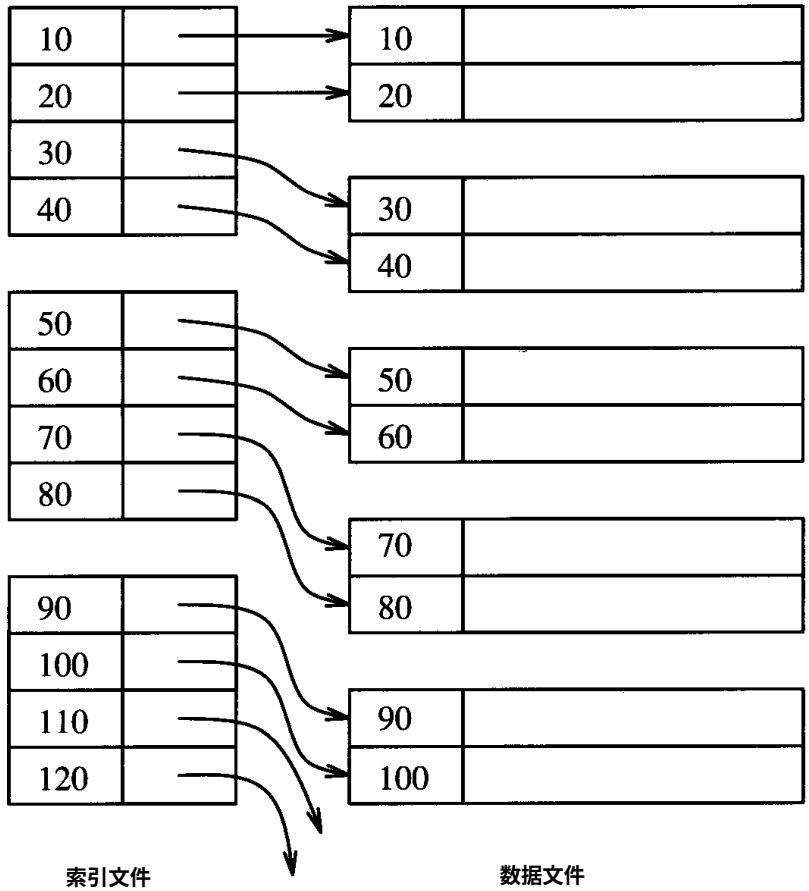

图4-3 顺序文件（右）上的稠密索引（左）

例4.2 假设一个关系有1 000 000个元组，大小为4096字节的存储块可存放10个这样的元组，那么这个关系所需的存储空间就要超过 400MB，因为太大而可能在主存中无法存放。然而，要是关系的键字段占30字节，指针占8字节，加上块头所需空间，那么我们可以在一个小大为 4096字节的存储块中存放100个键-指针对。

这样，一个稠密索引就需要10 000个存储块，即40 MB。我们就有可能为这样一个索引文件分配主存缓冲区，不过要看主存的使用情况和主存的大小。进一步讲， log2（10000）大约是13，因此采用二分查找法我们只需访问 13~14个存储块就可以查找到给定键值。且由于二分查找法只检索所有存储块中一小部分（是这样一些存储块，它们位于 1/2、1/4或3/4、1/8或3/8、5/8或7/8，等等），即使不能把整个索引文件存放到主存中，我们也可以把这些最重要的存储块放到主存中，这样，检索任何键值所需的I/O次数都远小于14。 □

索引块的定位

假设存在一些定位索引块的机制，根据索引可以查找到单个元组（若是稠密索引）或数据存储块(若是稀疏索引)。许多定位索引的机制可以使用。例如。假如索引较小，我们可以将它存放到主存或磁盘的预留存储区中。假如索引较大，则如我们在 4.1.4节讲述的那样，我样可以在索引上再建一级索引，并将新的索引本身存放到某个固定的地方。对这一观点进行推广。最后就产生了4.3节中的B树，在B树中我们只需要知道根索引块的位置。

4.1.3 稀疏索引

要是稠密索引太大，我们可以使用一种称为稀疏索引的类似结构。它节省了存储空间，但查找给定值的记录需更多的时间。如图 4-4所示，稀疏索引只为每个存储块设一个键 -指针对。键值是每个数据块中第一个记录的对应值。

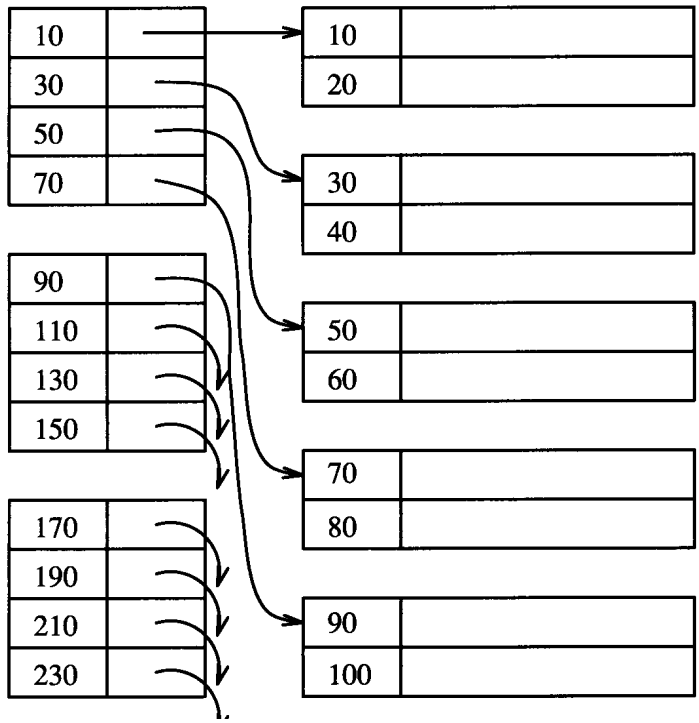

图4-4 顺序文件上的稀疏索引

例4.3 同例4.1一样，假定数据文件已排序，且其键值为连续的 10的倍数，直至某个较大的数。我们还继续假定每个存储块可存放四个键 -指针对。这样，第一个索引存储块中为前四个数据存储块的第一个键值的索引项，它们分别是 10、30、50和70。按照前面假定的键值模式，第二个索引存储块中为第 5~第8数据存储块的第一个键值的索引项，它们分别是 90、110、130和

150。图中还列出第三个索引存储块存放的键值，它们分别是假设的第九 ~第十二个数据存储块的第一个键值。 □

例4.4 稀疏索引所需的索引存储块要比稠密索引少得多，以例4.2中更接近实际的参数为例。由于有100 000个数据存储块，且每个索引存储块可存放100个键-指针对，那么要是使用稀疏索引，我们就只需要1000个索引存储块。现在索引的大小只剩下4MB，这样大小的文件完全可能存到主存中。

另一方面，稠密索引不用去检索包含记录的块，就可以回答下面形式的查询：“是否存在键值为K的记录？”。键K在索引中的存在足以保证数据文件中键值为K的记录的存在。当然如果使用稀疏索引，对于同样的查询，却需要执行I/O操作去检索可能存在键值为K的记录的块。 □

在已有稀疏索引的情况下，要找出键值为K的记录，我们得在索引中查找键值小于或等于K的最大键值。由于索引文件已按键排序，我们可以使用二分查找法来定位这个索引项，然后根据它的指针找到相应的数据块。现在我们必须搜索这个数据块以找到键值为K的记录。当然，数据块中必须有足够的格式化信息来标明其中的记录及记录内容。只要合适，可以采用3.2节和3.4节中的任何技术。

4.1.4 多级索引

索引文件本身可能占据多个存储块，正如我们在例4.2和例4.4中看到的那样。要是这些存储块不在我们知道能找到它们的某个地方，比如指定的磁盘柱面，那么，就可能需要另一种数据结构来找到这些索引存储块。即便能定位索引存储块，并且能使用二分查找法找到所需索引项，我们仍可能需要执行多次I/O操作才能得到我们所需的记录。

通过在索引上再建索引，我们能够使第一级索引更为有效。图4-5对图4-4进行了扩展，它是在图4-4的基础上增加二级索引得到的（和前面一样，我们假设使用10的连续倍数这一不常见的模式）。按照同样想法，我们可以在二级索引的基础上建立三级索引，等等。然而，这种做法有它的局限，与其建立多级索引，我们宁愿考虑使用在4.3节讲述的B树。

在这个例子中，一级索引是稀疏的，虽然我们也可以选择稠密索引来作为一级索引。但是，二级和更高级的索引必须是稀疏的，因为一个索引上的稠密索引将需要和其前一级索引同样多的键-指针对，因而也就需要同样的存储空间。因此，二级稠密索引只是增加额外的结构，而不会带来任何好处。

例4.5 继续以例4.4中假设的关系来分析，假定我们在一级稀疏索引上建立二级索引。由于一级稀疏索引占据1000个存储块，且我们可以在一个存储块中存放100个键-指针对，则只需要10个存储块来存放这个二级索引。

10个存储块完全可以一直缓存在主存中。若是这样，要查找给定键值K的记录，可以先查看二级索引以找到小于或等于键值K的最大键值。根据这个最大键值对应的指针找到一级索引的某个存储块B，通过这个存储块B，我们就肯定能找到所需的记录。假若存储块B不在内存中，我们就把存储块B读进内存，这是我们所需的第一次I/O操作。在块B中查找小于或等于键值K的最大键值，要是有键值为K的记录存在，则通过块B中这一键值对应的指针就可以找到包含键值为K的记录的数据块。这个数据块需要又一次的I/O操作。这样，我们只用两次I/O操作就完成了查询。

□

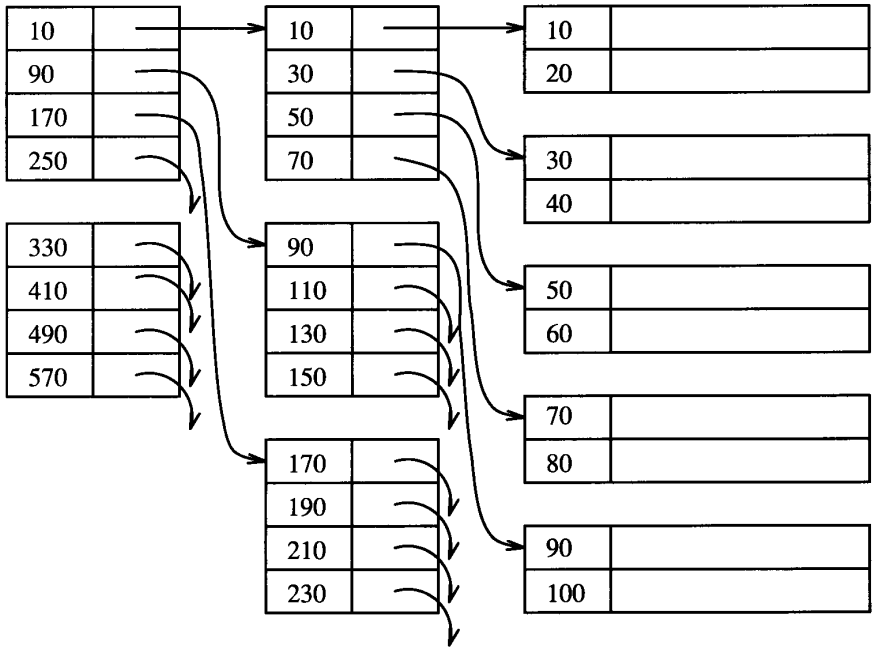

图4-5 增加一个二级稀疏索引

4.1.5 重复键的索引

到目前为止，我们都假定作为建立索引基础的查找键是关系的键，所以对任何一个键值，关系中最多有一个记录存在。然而，索引经常用于非键属性，因此有可能一个给定的键对应于多个记录。假如按查找键对记录进行排序，而不管相同键值记录之间的次序，那么，我们可以采用前面介绍的方法来处理不是关系的键的查找键。

对前面方法最简单的扩充是为数据文件建立稠密索引：每一个具有键值K的记录设一索引项。也就是说，我们允许索引文件中出现重复的查找键。找出所有给定索引键 K的记录因此就比较简单：在索引文件中找到具有键值K的第一个索引项，后面紧接着就是其他的具有键值 K的索引项，找出它们，然后依据这些索引的指针找出所有具有键值 K的记录。

更为有效的方法是为每个键值 K只设一个索引项。该索引项的指针指向键值为 K的第一个记录。为了找出其他键值为K的记录，需在数据文件中顺序向前查找；在排序的数据文件中，这些记录一定紧跟在所找到的记录后存放。图 4-6举例说明了这一想法。

例4.6 假如要找出图4-6中所有索引键值为20的记录。先在索引中找到键值为 20的索引项并顺着它的指针找到第一个键值为 20的记录。然后在数据文件中继续往前找，由于刚好处于数据文件第二个存储块的最后一条记录，我们就到第三个存储块中去查找[⊖]。我们发现第三块中第一

⊖ 为了能找到数据文件的下一个块，我们可以用链表的形式把所有块链接起来，换言之，给每个存储块一个指针指向下一个存储块。我们也可以返回索引，沿下一指针到达下一数据文件块。

个记录的键值为20，但第二个记录的键值为30。因此，不用再往前查找；我们已经找到键值为20的两条记录。 □

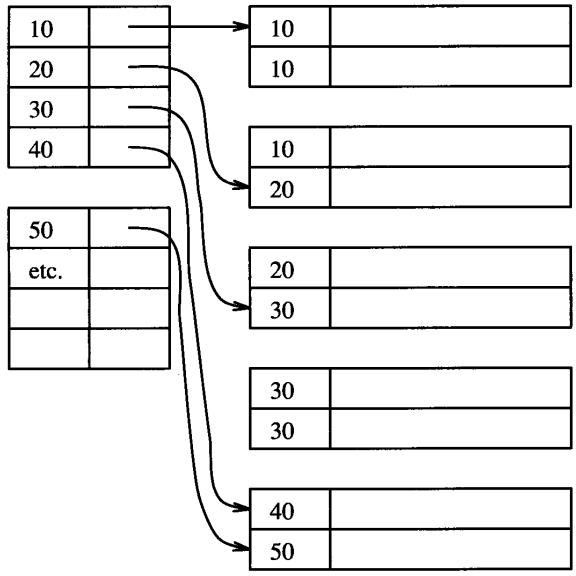

图4-6 允许重复键的稠密索引

图4-7为图4-6所示数据文件上的一个稀疏索引。这种稀疏索引是很常见的；它的键 -指针对对应数据文件中每个块的第一个查找键。

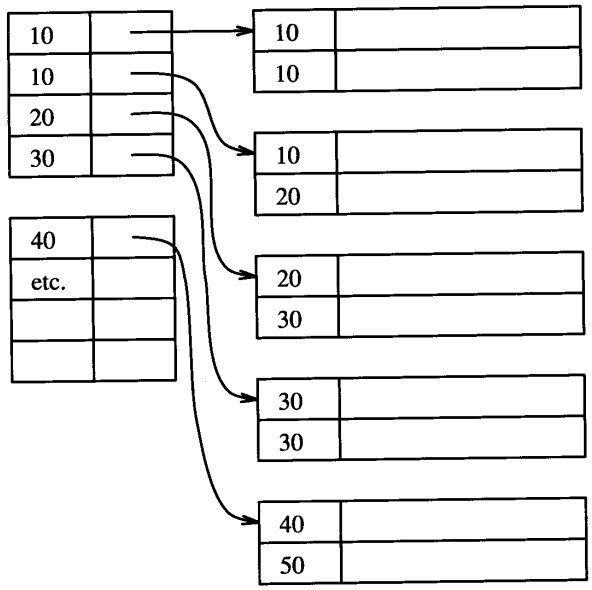

图4-7 指明了每个块中最小查找键的稀疏索引

为了在这种数据结构中找出所有索引键为 K的记录，我们先找到索引中键值小于或等于 K的最后一个索引项 E_1，然后往索引起始的方向找，直到碰到第一个索引项或碰到一个严格小于键值K的索引项E_2为止。从E_2到E_1的索引项（含 E_2和E_1）指向所有可能包含查找键为 K的记录的数据块。

例4.7　假定我们想在图 4-7中查找键值为 20的记录。第一个索引块的第三个索引项就是 E_1，它是符合键值小于等于 20的最后一个索引项。我们向后查找，立即找到了键值小于 20的索引项。因此，第一个索引块的第二个索引项就是 E_2。它们相应地指向第二、第三个数据块，正是在这两个数据块中我们找到查找键为 20的所有记录。

再举一例。若K等于10，则E_1就是第一个索引块的第二个索引项，而 E_2不存在，因为我们在索引中找不到更小的键值。因此，按照第一个索引项到第二个索引项的指针，我们找到前两个数据块。在这两个数据块中我们就能找到所有键值为 10的记录。　□

一种稍有不同的方式如图 4-8所示。图中为每个数据块中新的、即未在前一存储块中出现过的最小查找键设一个索引项。要是在存储块中没有新键值出现，那么就为该块中唯一的键值设一个索引块。在这种方式下，我们查找键值为 K的记录可以通过在索引中查找第一个键值满足如下条件之一的索引项。

a) 等于K；或者

b) 小于K，但下一个键值大于K。

我们按照这个索引项的指针找到相应的数据块。要是在这个数据块中至少找到一个键值为 K的记录，那么我们就要继续查找其他数据块，直到找出所有查找键为 K的记录。

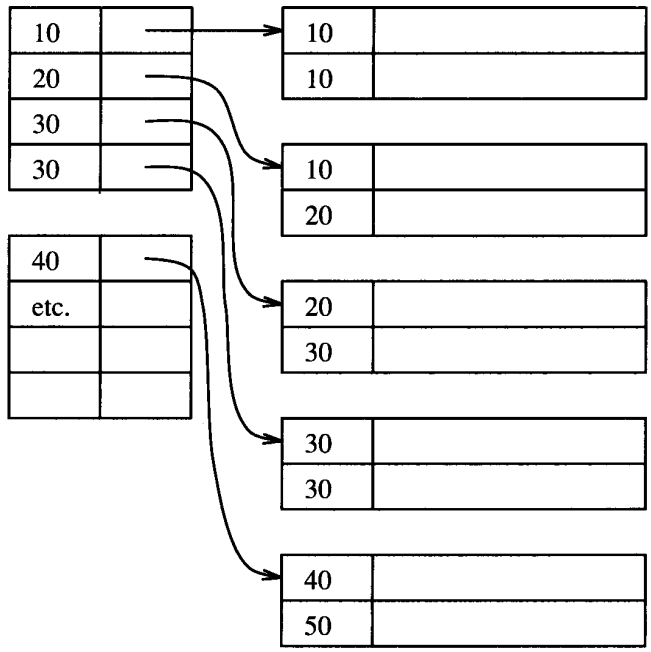

图4-8　指明每块中最小新查找键的稀疏索引

例4.8 假定要在图4-8所示的结构中查找K=20的记录，上述规则指示出第一个索引块的第二个索引项，我们沿着这个索引项的指针找到键值为 20的第一个存储块。由于下一个存储块也有键值为20的记录，我们必须继续向前查找。

假若K=30，上述规则指示出第三个索引项。我们沿着它的指针找到第三个数据块，键值为30的记录在这个数据块中开始存放。最后，假若 K=25，则上述选择规则的b）部分指出第二个索引项。我们因此来到第二个数据块。如果存在查找键为 25的记录，则至少有一个在该块上值为20的记录之后，因为我们知道第三个存储块中新出现的第一个键值是 30。既然没有键值为 25的记录，我们的查找就失败了。 □

4.1.6 数据修改期间的索引维护

到目前为止，我们都假定数据文件和索引文件由一些连续、装满某种类型的记录的存储块组成。由于随着时间的推移数据会发生变化，我们需对记录进行插入、删除和更新。这势必引起像顺序文件这样的文件组织发生变化，以至于曾经能容纳于块中的记录不再被容纳。我们可以使用3.5节讲述的技术来重新组织数据文件。我们来回忆一下那一部分的三个重要想法。

1) 当需要额外的存储空间时，创建溢出块；或当溢出块中记录被删除后不再需要该存储空间时删除溢出块。溢出块在稀疏索引中没有索引项而应该被看作是基本存储块的扩充。
2) 若不用溢出块，可以按序插入新的存储块。要是这样做，那么新的存储块就需要在稀疏索引中设索引项。在索引文件中索引项的变动会引起和数据文件的插入与删除同样的问题。要是我们创建新索引块，那么这些索引块必须能以某种方法定位，例如像 4.1.4节中那样使用另一级索引。
3) 当块中没有空间可以插入元组时，有时可移动一些元组到相邻块；相反，当相邻块元组太少时，我们可以合并它们。

然而，当数据文件发生变化后，我们通常必须对索引进行调整以适应数据文件的变化。具体的方法要依赖于索引是稠密还是稀疏以及在上述三种方法中选择哪一个。不过，我们应记住一个一般的原则：

- 索引文件是顺序文件的一个例子，键 -指针对可以看作是按查找键排序的记录。因此，数据文件修改过程中用来维护数据文件的那些策略同样适用于索引文件。

在图4-9中，我们总结了针对数据文件七种不同行为而对稠密索引或稀疏索引所需采取的措施。这七种行为包括：创建或删除空溢出块、创建或删除顺序文件的空块、插入、删除和移动记录。注意，假定空的存储块才能被创建或删除，具体来说，要是我们想删除一个有记录的块，需要先删除其中的记录或把记录移到其他的块。

在这个表中，我们注意到：

- 创建或删除一个空溢出块对两种索引均无影响。对稠密索引不产生影响是因为索引是针对记录；对稀疏索引不产生影响是因为索引是针对基本存储块而非溢出块。
- 创建或删除顺序文件的块对稠密索引无影响，这仍是因为索引是针对记录而非存储块；但它对稀疏索引会有影响，因为我们必须为创建或删除的块分别创建或删除一个索引项。
- 插入或删除记录导致稠密索引上的同一动作，因为记录的相应键-指针对要被插入或删除。

行　为	稠密索引	稀疏索引
创建空溢出块	无	无
删除空溢出块	无	无
创建空顺序块	无	插入
删除空顺序块	无	删除
插入记录	插入	更新（?）
删除记录	删除	更新（?）
移动记录	更新	更新（?）

图4-9　顺序文件上的行为对索引文件的影响

然而，这对稀疏索引通常没有影响。例外的情况是当记录是存储块中第一个记录时，稀疏索引中对应块的键值必须被更新。因此，我们在图 4-9中相应的更新操作后加注了一个问号，表示这个更新是可能的，但不确定。

- 类似地，移动一个记录，不论是在块内还是在块间都会引发稠密索引中相应索引项的更新；对稀疏索引而言，则仅当被移动记录是或变成了该块中的第一个记录时才会引发更新操作。

为数据变迁所做的准备

由于随着时间的推移，关系或类外延通常会增长。因而较为明智的做法是：不论数据块还是索引块，都为其保留一定的空闲空间。比如说，一开始在每块中只使用75%的空间。这样，在创建溢出块或在块之间移动记录之前，我们能运行一段时间。无溢出块或仅有少量的溢出块的优势在于访问每个记录平均I/O次数仅为1。溢出块数越多，则查找给定记录所需访问的平均存储块数也多。

我们将通过一系列的例子来阐明上述几点所隐含的一组算法。这些例子既包括稀疏索引又包括稠密索引，既包括记录移动方法又包括溢出块方法。

例4.9　首先，让我们来考虑顺序文件的记录删除操作，该顺序文件上建有稠密索引。我们从图4-3所示的文件和索引开始。假设键值为 30的记录被删除。图4-10所示为记录删除后的结果。

首先，键值为 30的记录从顺序文件中删除。我们假定块外的指针可以指向块内的记录，因而我们不打算将剩余记录 40在块中向前移动，而选择在记录 30处留下一个删除标记。

在索引中，我们删去键值为 30的键-指针对。假定不允许块外的指针指向索引记录，因而没有必要为该键-指针对留下删除标记。所以，我们采取合并索引块的方法，并把后面的索引记录前移。　□

例4.10　现在，我们来考虑顺序文件上的两个删除操作，该顺序文件上建有稀疏索引。从图4-4所示的文件和索引开始，同样假设键值为 30的记录被删除。我们还假定块中记录可前后移动：要么块外没有指针指向记录，要么我们使用图 3-17所示的偏移量表来支持这样的移动。

删除记录30后的情形如图4-11所示。记录30已被删除，后面的记录40向前移以使块的前部紧

凑。由于40现在是第二个数据块的第一个键值，我们需要更新该块的索引记录。从图 4-11中我们看到，与指向第二数据块指针相对应的键值已从 30改为40。

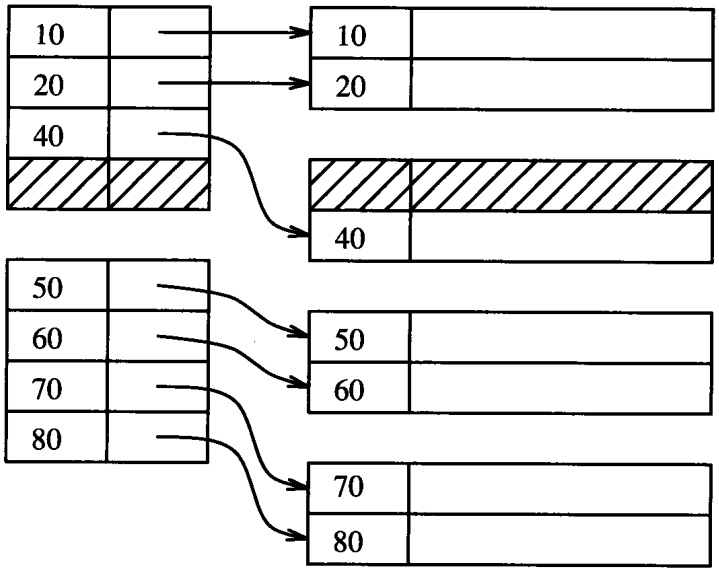

图4-10 稠密索引中删除查找键为30的记录

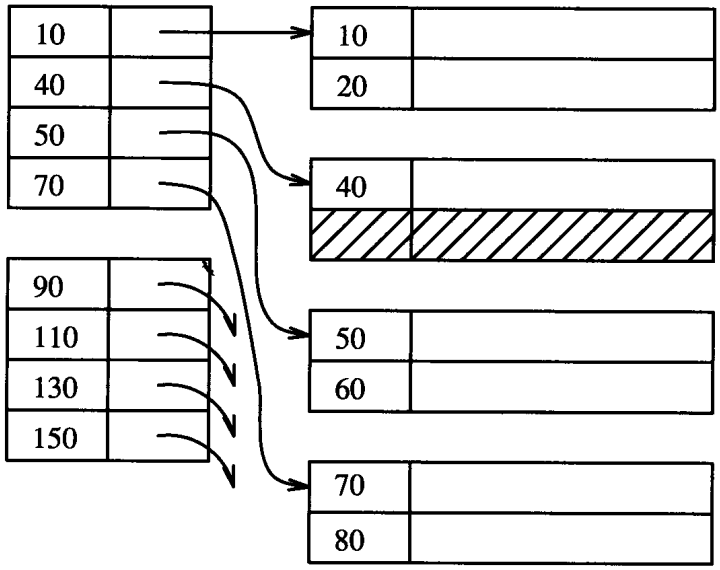

图4-11 稀疏索引中删除查找键为30的记录

现在，假定记录40也被删除，删除后的情形如图 4-12所示。第二个数据块已经没有记录。如果顺序文件是存放在任意的存储块上（例如，不是柱面上连续的存储块），那么我们可以把该块链接到可用空间链表中。

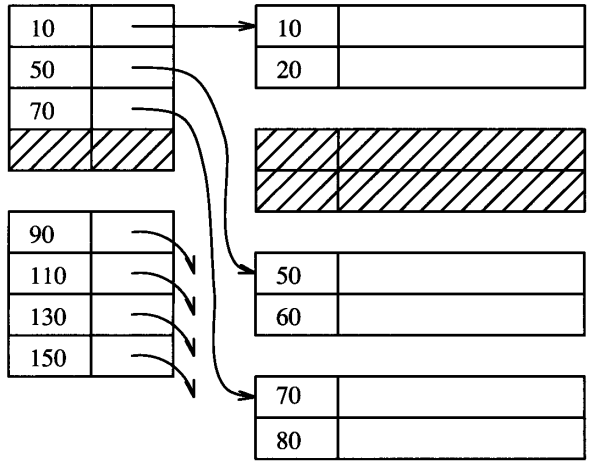

图4-12 稀疏索引中删除查找键为40的记录

要完成记录40的删除，我们还要调整索引。既然第二个数据块不再存在，我们就从索引中删除其索引项。在图4-12中通过把后面的索引项前移，使第一个索引块紧凑。这一步是可选的。□

例4.11 现在，我们来考虑插入的影响，从图4-11开始，该图中我们刚刚从一个有稀疏索引的文件中删除了记录30，但记录40还保留着。我们现在插入一个键值为15的记录。通过查看稀疏索引，我们发现该记录属于第一个数据块。但是该数据块已满；它存放着记录10和记录20。

我们能做的一件事是在附近找有空闲空间的块，这样就找到第二个数据块。因此，我们就在文件中往后移动记录，以腾出空间来存储记录15。结果如图4-13所示。记录20从第一个数据块移到了第二个数据块，并且记录15存放到记录20的位置上。为了在第二个数据块中存放记录20和保持记录的顺序，将第二个数据块的记录40往后移并把记录20放在它的前面。

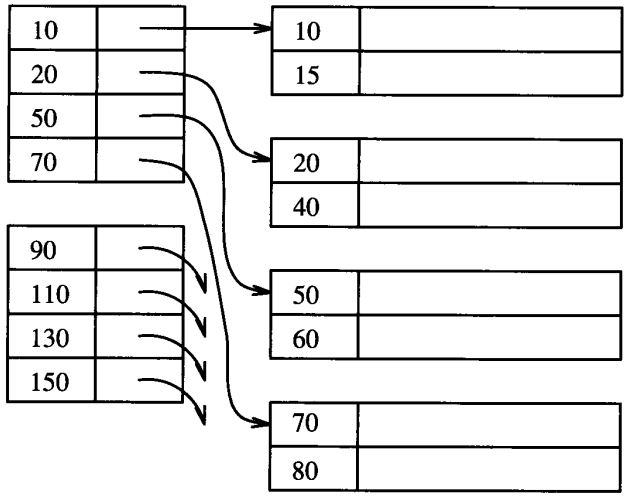

图4-13 在有稀疏索引的文件中插入，使用立即重组方法

最后一步是修改变动的数据块的索引项。我们可能需要修改第一个数据块的键-指针对的键值。不过这里不用，因为插入的记录不是该数据块的第一个记录。然而，我们需要修改第二个数据块的索引项的键值，因为该块的第一个记录的键值原来是40，现在变成了20。 □

例4.12 例4.11所示策略的问题在于，我们恰好在相邻存储块中找到了空闲空间。要是前面没有删除键值为30的记录，那么我们查找空闲空间的工作只是徒劳。原则上讲，我们不得不把从记录20到数据文件最后的所有记录全部后移，直到我们到达文件末尾并能创建额外的块。

因为有这样的风险，用溢出块为有太多记录的基本块补充空间的做法通常更为明智。图1-14所示为在图4-11所示结构中插入键值为15的记录后的情形。和例4.11一样，第一个数据块有太多记录。我们不是将记录移动到第二个数据块，而是为第一个数据块创建一个溢出块。可以看到图4-14中每个数据存储都有一个小凸起，它表示块头中存放指向溢出块的指针的空间。任意多个溢出块之间可通过这些指针空间链接起来。

在我们这个例子中，记录15被插入到位于记录10之后的正确位置；记录20为腾出空间被移到溢出块中。因为数据块1的第一个记录没有改变，所以索引项也就不必改变。注意，索引中不存在该溢出块的索引项，因为该溢出块被看成是数据块1的扩充，它不是顺序文件本身的存储块。□

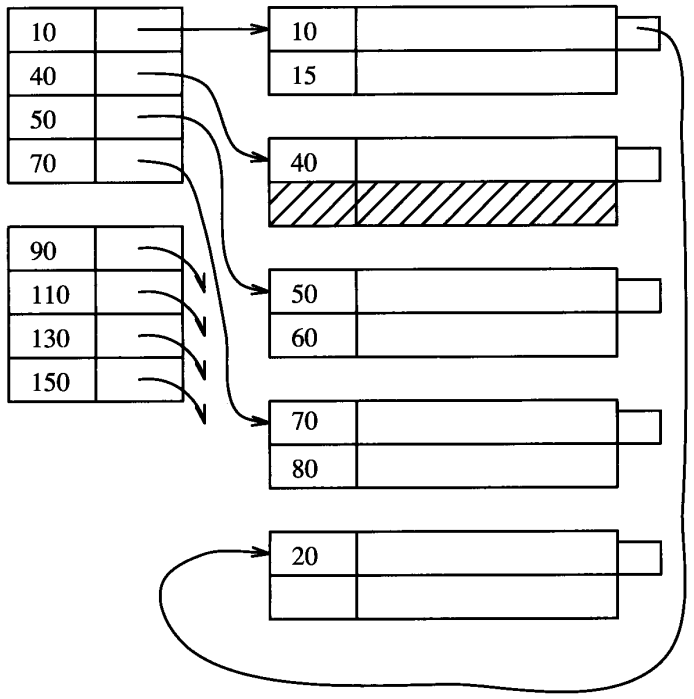

图4-14 使用溢出块技术在有稀疏索引的文件中插入

习题

*习题4.11 假定每个存储块可存入3个记录或10个键-指针对。设记录数为n，保存一个数据

文件和

a) 一个稠密索引

b) 一个稀疏索引

各需要多少块（表示为n的函数）？

习题4.1.2　假定每个存储块可存放30个记录或200个键-指针对，但数据块和索引块充满度均不许超过80%。重做习题4.1.1。

!习题4.1.3　假定我们使用多级索引，级数恰好使最后一级索引只有一个存储块。重做习题4.1.1。

*!!习题4.1.4　假定每个存储块可存放3个记录或10个键-指针对，如习题4.1.1，但是可能出现重复键。说得具体点，数据库中1/3的键有一个记录，1/3的键有两个记录，1/3的键恰好有三个记录。假设我们有一个稠密索引，但它只为每个查找键值设一个索引项，指针指向该键值的第一个记录。最初没有存储块在内存中，试计算找到给定键值K的所有记录所需的平均磁盘I/O数。你可以认为包含键K的索引块地址已知，尽管它在磁盘上。

!习题4.1.5　分别按下述情况重做习题4.1.4。

a) 稠密索引：每个记录设一个键-指针对，包括有重复键的记录。

b) 稀疏索引：如图4-7所示，索引中指明每个数据块的最小键值。

c) 稀疏索引：如图4-8所示，索引中指明每个数据块新出现的最小键值。

!习题4.1.6　假如在一个关系的主键属性上建立稠密索引，那么，我们就可以使指向元组（或表示这些元组的记录）的指针指向索引项而非记录本身。这两种方式各自的优点是什么？

习题4.1.7　在图4-13基础上继续做改变，假设接下来删除键值为60、70和80的记录，然后插入键值为21、22等直到29的所有记录。假定额外的空间可通过下述方式得到：

*a) 为数据文件或索引文件增加溢出块。

b) 记录尽量后移：必要时在数据文件和/或索引文件末尾增加存储块。

c) 可根据需要在这些文件中间插入新的数据块或索引块。

*!习题4.1.8　假定在处理一个具有n个记录的数据文件中的插入操作时，我们按照需要创建溢出块。同时还假定数据块当前平均只充满一半。假如随机地插入记录，试问得插入多少记录才能使我们查找到给定键值记录所需访问存储块（包括溢出块）的平均数达到2？假定在查找中，我们先查看索引指向的块，然后只按顺序查看溢出块，直到找到所需记录，该记录必然在链上的某一块中。

4.2 辅助索引

4.1节中所描述的数据结构为主索引，因为它们决定了被索引记录的位置。在4.1节中数据文件是按查找键的值排序，由此能决定记录的位置。在4.4节中将讨论另一种常用的主索引：散列表，其中查找键决定记录所属的“桶”。

然而，为了给各种各样的查询提供便利，经常需要在一个关系上建多个索引。例如，再考虑图3-1中定义的关系`MovieStar`。由于将`name`声明为该关系的主键，我们可以期望DBMS创

建主索引结构来支持指定明星名字的查询。但是我们现在想在数据库中按出生日期查找指定明星的信息，因而执行如下查询：

```
SELECT name, address
FROM MovieStar
WHERE birthdate = DATE '1950-01-01';
```

我们需要在属性birthdate上建立辅助索引来帮助执行这个查询。在SQL系统中，可以通过如下显示的命令来建立这样一个索引：

```
CREATE INDEX BDIndex ON MovieStar(birthdate);
```

辅助索引可用于任何索引目的：这种数据结构有助于查找给定一个或多个字段值的记录。但是，辅助索引与主索引最大的差别在于辅助索引不决定数据文件中记录的存放位置。而仅能告诉我们记录的当前存放位置，这一位置可能是由建立在其他某个字段上的主索引确定的。辅助索引和主索引的这一差别有一个有趣的推论：

- 谈论一个稀疏的辅助索引是毫无意义的。因为辅助索引不影响记录的存储位置，我们也就不能根据它来预测键值不在索引中显式指明的任何记录的位置。
- 因此，辅助索引总是稠密索引。

4.2.1 辅助索引的设计

辅助索引是稠密索引，且通常有重复值。如前所述，索引项由键-指针对组成，这里的“键”是查找键且不要求唯一。为了便于找到给定键值的所有索引项，索引文件中索引项按键值排序。要是我们在这种结构上建立二级索引，那么这个二级索引将是稀疏的，其原因在4.1节中已讨论过。

例4.13 图4-15所示为一个典型的辅助索引。与我们前面图示的准则一样：数据文件中每块存放两个记录。记录只显示了各自的查找键；其值为整型，而且像前面一样我们给它们取值为10的倍数。要注意，与4.1.5节数据文件不同的是，这里的数据没有按查找键排序。

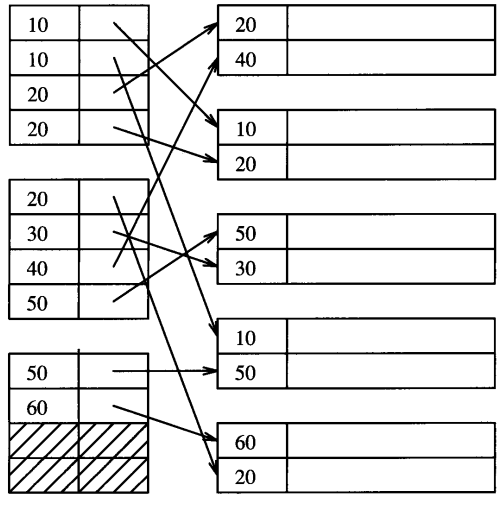

图4-15 辅助索引

然而，索引文件中的键是排序的。这样就造成索引块中的指针并不是指向一个或少数几个连续存储块，而是指向许多不同的数据块。例如，为了检索键值为 20的所有记录，不仅要查找两个索引块，而且还得访问指针指向的三个不同数据块。因此，查找同样数量的记录，使用辅助索引比使用主索引可能需要多得多的磁盘 I/O。但是这个问题是无法解决的，我们无法控制数据块中的元组顺序，因为这些元组可能已按其他属性排序。

我们可以在图 4-15所示的辅助索引上建立二级索引。这一级索引将是稀疏的，如 4.1.4节讨论的那样，对应于每个索引块的第一个键值或第一个新出现的键值有一个键 -指针对。 □

4.2.2 辅助索引的应用

除了能在被组织成顺序文件的关系（或类外延）上建立附加索引外，辅助索引甚至还用做某些数据结构的主键索引。这些结构之一就是“堆”结构。在这种结构中，关系的记录之间没有特定的顺序。

第二种需要辅助索引的常见数据结构是聚簇的文件。在这种结构中，两个或多个关系的元组被混在一起。下面的一个例子说明了这种组织结构在特定情况下存在的合理性。

例4.14 假设有两个关系，其数据模式可简要定义如下：

```
Movie(title, year, length, studioName)
Studio(name, address, president)
```

属性title和year一起组成关系Movie的键，而属性name是关系Studio的键。Movie中的属性studioName是参照Studio中的name的外键。进一步假定查询的常见形式如下：

```
SELECT title, year
FROM Movie
WHERE studioName = 'zzz';
```

这里,*zzz*表示某一特定制片厂的名称，如，‘Disney’。

要是我们确信上面这类查询是典型的查询，那么就可不按主键 title和year排序，而是按studioName来给元组排序。如 4.1.5节讨论的那样，我们就在这个顺序文件上建立一个带重复键的主索引。这样做的好处在于当我们按给定的制片厂名称来查询电影时，可以在少数的几个，或者就比查找的记录所占最小存储块数多一个的存储块中找到所需记录。这使得该查询的磁盘 I/O次数最少，使回答这种查询的效率极高。

但是，假如需要把电影的信息与制片厂的信息关联起来，仅按 Movie的主键外的属性排序是没有帮助的。例如：

```
SELECT president
FROM Movie, Studio
WHERE title = 'Star Wars' AND
      Movie.studioName = Studio.name
```

即：找出制作电影“Star Wars”的制片厂的经理。又如：

```
SELECT title, year
FROM Movie, Studio
WHERE address LIKE '%Hollywood%' AND
```

```
Movie.studioName = Studio.name
```

即：找出所有在Hollywood制作的电影。为了实现这些查询，我们需要把关系 Movie和Studio进行连接。

如果我们确信关系Moive和Studio之间的基于制片厂名称的连接很常见，就可以采用一种聚簇的文件结构来使这些连接效率更高。在这种结构中，关系 Movie的元组和关系Studio的元组存放在相同的一系列块中。说得具体些，我们在每个 Studio的元组后面存放关系Movie中该制片厂的所有电影元组。其结构如图 4-16所示。

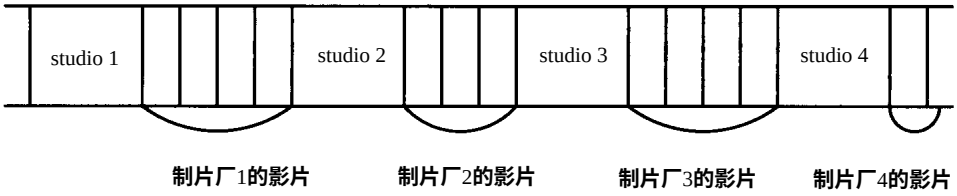

图4-16 将制片厂及其制作的影片聚簇在一起的聚簇文件

现在，如果想找出制作某部电影的制片厂的经理，我们很有可能在同一存储块中找到该制片厂和它制作的电影的记录，这可以省去一个 I/O步骤。要是想找出某个制片厂制作的全部电影，我们也会在相应电影制片厂的存储块中很容易地找出这些的电影，这也能节省 I/O。

然而，为了高效地执行这些查询，我们需要有效地找到给定名称的电影或制片厂。因此，需要在Movie.title上建立辅助索引以查找给定电影名的一部电影（或多部电影，因为有可能存在多部同名的电影），而不管它们位于存放 Moive和Studio元组的哪些块中。同样，为了查找给定制片厂名的Studio元组，我们也需要在Studio.name上建立辅助索引。 □

4.2.3 辅助索引中的间接

图4-15所示结构存在空间浪费，有时浪费很大。假如某个索引键值在数据文件中出现 n次，那么这个键值在索引文件中就要写 n次，如果我们只为指向该键值的所有指针存储一次键值，这样就会比较好。

避免键值重复的一种简便方法是使用一个称为桶的间接层，它介于辅助索引文件和数据文件之间。如图4-17所示，每个查找键K有一个键-指针对，指针指向一个桶文件，该文件中存放 K的桶。从这个位置开始，直到索引指向的下一个位置，其间指针指向索引键值为 K的所有记录。

例4.15 例如在图4-17的索引文件中，沿索引键为50的索引项指针找到中间“桶”文件。这一指针刚好将我们带到桶文件中第一个块的最后一个指针。继续向前查找，找到下一块的第一个指针。因为索引文件中键值为 60的索引项指针刚好指向桶文件的第二个块的第二个指针，所以我们停止查找。 □

在图4-17所示的方式中，只要查找键值的存储空间比指针大就可以节省空间。不过，即使在键值和指针大小相当的情况下，在辅助索引上使用间接也有一个重要的好处：我们通常可以在不访问数据文件记录的前提下利用桶的指针来帮助回答一些查询。特别是，当查询有多个条件，

而每个条件都有一个可用的辅助索引时，我们可以通过在主存中将指针集合求交来找到满足所有条件的指针，然后只需要检索交集中指针指向的记录。这样，我们就节省了检索满足部分条件而非所有条件的记录所需的I/O开销[⊖]。

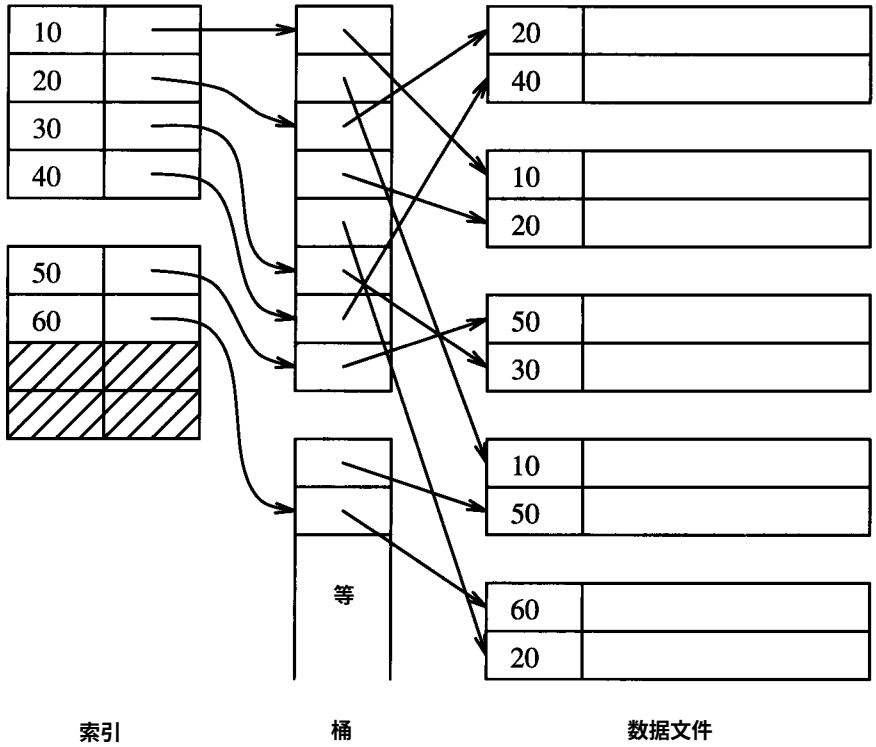

图4-17 在辅助索引中使用间接节省空间情况

例4.16 考虑例4.14中的关系

```
Movie(title, year, length, stadioName)
```

假定我们在studioName和year上都建立了有间接桶的辅助索引，而且我们要执行如下查询：

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND
      year = 1995;
```

即：找出Disney在1995年制作的所有电影。

图4-18说明了如何使用索引来回答这个查询。通过 studioName上的索引，我们找出了所有指向Disney制作的电影的指针。但是，我们并不把这些记录从磁盘上取到主存中，而是通过

⊖ 假若我们直接从索引而非桶中取得指针，也可以使用这一指针求交技巧。不过，由于指针比键-指针对使用更少存储空间，因此使用桶通常能节省磁盘I/O。

year上的索引，再找出所有指向1995年制作的电影的指针。然后我们求两个指针集的交集，正好得到1995年Disney制作的所有电影。现在我们到磁盘上去检索所有包含一部或几部这样的电影的块，这样只需检索尽可能少的数据块。

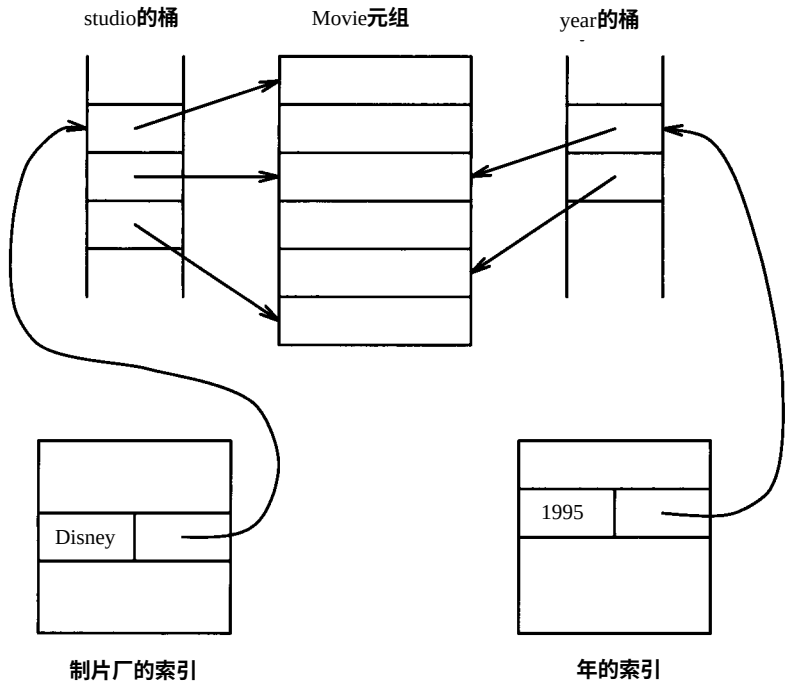

图4-18 在主存中求桶的交集

4.2.4 文档检索和倒排索引

多年来，信息检索界都在处理文档的存储和按关键字集高效检索文档的问题。随着WWW的出现以及在线保存所有文档成为可能，基于关键字的文档检索已成为数据库最大的难题之一。尽管可用来找出相关的文档的查询有许多种，但最简单、最常见的形式可用关系的术语描述为：

- 一个文档可被看成是关系Doc的元组。这个关系有很多属性，每个属性对应于文档可能出现的一个词。每个属性都是布尔型——表明该词在该文档出现还是没有出现。因此，这一关系模式可以被看作：

  Doc(hascat, hasDog,…)

  其中hascat取值为真当且仅当该文档中至少出现一次“cat”这个词。
- 关系Doc的每个属性上都建有辅助索引。不过，我们不必费心为属性值为FALSE的元组建索引项；相反，索引只会将我们带到出现该词的那些文档，即，索引中只有查找键值为TRUE的索引项。
- 我们不是给每个属性（即每个词）建立一个单独的索引，而是把所有的索引合成一个，称为倒排索引。这个索引使用间接桶来提高空间利用率，正如4.2.3节中讨论的那样。

例4.17 图4-19所示为一个倒排索引。取代记录数据文件的是一个文档集合，每个文档可以被存放在一个或多个磁盘块上。倒排索引本身由一系列词-指针对组成；词实际上是索引的查找键。正如到目前为止讨论的任何一种索引那样，倒排索引被存储在连续的块中。不过，在一些文档检索应用中，数据的变化不像典型数据库中那样频繁，因而通常也就不用提供应付块溢出或索引变化的措施。

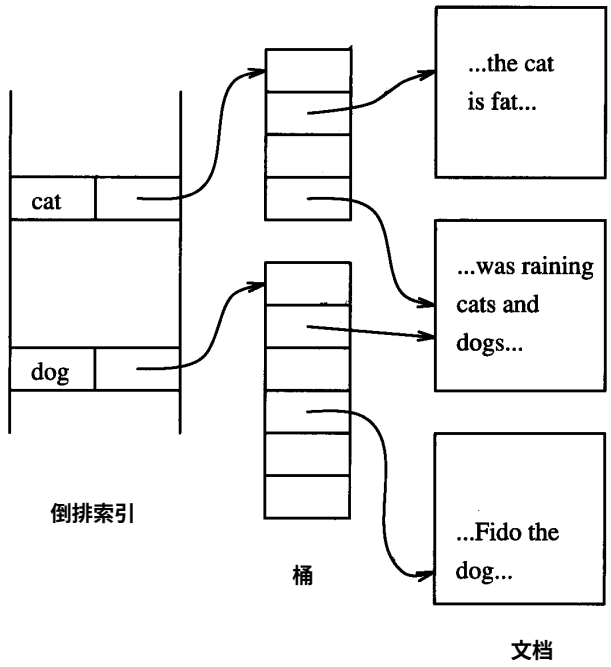

图4-19 文档的倒排索引

指针指向“桶”文件中的位置。例如，在图 4-19中，“cat”一词有一个指针指向桶文件。该指针在桶文件中指明的位置之后是所有包含“ cat”的文档的指针。图中给出了一些这样的指针。类似地，图中“ dog”一词的指针指向一个指针列表，该列表中的指针指向包含“ dog”的所有文档。 □

桶文件中的指针可以是：

1) 指向文档本身的指针。

2) 指向词的某一次出现的指针。在这种情况下，指针可以是由文档的第一个块和一个表示该词在文档中出现次数的整数构成的对。

一旦有了使用这种由指向词的每次出现的指针桶的想法，我们可能就会想扩展这个想法，使桶数组包含更多有关词的出现的信息。这样，桶文件本身就成了有重要结构的记录集合。这种做法的早期应用是区分一个词出现在文档的题目、摘要，还是正文中。随着 Web上文档的增长，尤其是使用HTML、XML或其他标记语言的文档的增长，我们也可以指明与词关联的标记。例如，我们不仅可以区分出现在题头、表或锚中的词，而且可以区分以不同字体和字号出现的词。

对信息检索的进一步讨论

有很多技术可用于改进基于关键字的文档检索效率。完整介绍这些技术已超出本书的范围，这里介绍两种有用的技术：

1) 抽取词干。在将单词的出现放入索引中之前。我们删除词的后缀以找出它的“词干”。例如，复数名词可被当作其单数形式处理。因此，例4.17中的倒排索引显然使用了抽取词干技术，因为搜索“dog”一词时我们不仅得到“dog”的文档，而且得到一个有单词“dogs”的文档。
2) 无用词。像“the”、“and”之类最常用的词称为无用词，通常不包含在倒排索引中。原因在于好几百个常用词出现在太多的文档中，以至于它根本无益于检索特定主题。去除无用词还可以明显地缩小索引。

例4.18 图4-20所示为一个标明HTML文档中词的出现情况的桶文件。如果有出现类型即标记的，就在第一列指明。第二、第三列一起构成指针指向词的出现。第三列指明文档，而第二列给出了该文档中该词出现的次数。

可以用这种数据结构来回答关于文档的各种查询，而且不用仔细查看文档。例如，假设我们想找出有关狗的，并将狗与猫做了比较的文档；没有深刻理解文档的内容，我们就无法准确地回答这个查询。但是，要是查找符合以下条件的文档，我们可以获得一个很好的提示：

a) 在标题(title)中提到狗，且
b) 在某个锚中提到猫——该锚可能是连到一个关于猫的文档的链接。

桶中的插入与删除

在一些图如图4-19中，我们所给的桶是大小适中的紧凑数给。实际上，它们是单个字段(指针)的记录，且像其他任何记录集合一样存放在块中。因此在插入和删除指针时，我们可用到目前为止学过的任一种技术，例如为文件的扩充预留空闲空间、溢出块和可能的块内或块间记录移动。在后一种情况下，当我们移动例排索引的桶中指针指向的记录时，必须小心地改变从倒排索引到桶文件中的相应指针。

我们可用通过对指针求交来回答这个查询。也就是说，按对应于“cat”的指针找到这一单词的所有出现。我们从桶文件中选择有“cat”出现且类型为“锚”的文档指针。接着，我们找到“dog”的桶中项目，并从中选择类型为“标题”的文档指针。如果把这两个指针集相交，就得到符合在标题中提到“dog”且在某个锚中提到“cat”这一条件的文档。 □

习题

*习题4.2.1 随着数据文件的插入和删除，辅助索引也需要修改。请提出一些使辅助索引与数据文件的改变同步的方法。

!习题4.2.2 如习题4.1.1一样，假定我们的存储块能存放3个记录或10个键-指针对。假设用这样的块来存放一个数据文件和查找键K上的辅助索引。对于文件中出现的每个K-值v，可能有1

个、2个或3个记录的K字段值为v。正好1/3的键值出现一次，1/3的键值出现2次，1/3的键值出现3次。再假定索引块和数据块都在磁盘上，但有在一种结构能使我们接受任意值 v，并获知包含一个或多个查找键值为K的记录的所有索引块指针（或许在主存中有一个二级索引）。计算检索所有查找键值为v的记录的平均磁盘I/O数？

*!习题4.2.3 考虑如图4-16所示的聚集文件组织结构，且假定每个存储块可以放10个制片厂记录或电影记录。再假定每个制片厂制作的电影数都在1～m之间均匀分布。如果表示成m的函数，则检索某个制片厂和它所制作的电影所需的平均磁盘I/O数是多少？如果电影记录随机分布在大量块中，这个平均磁盘I/O数又是多少？

习题4.2.4 假定一个存储块可存放3个记录、10个键值-指针对或50个指针。如果我们使用图4-17的间接桶：

*a) 如果平均每个查找键值出现在10个记录中，存放3000个记录和它的辅助索引共需要多少块？如果不使用桶又需要多少块？

!b) 如果给定键值的记录数没有限制，所需的最大和最小存储块数各为多少？

!习题4.2.5 在习题4.2.4a）的假定下，在有桶结构和无桶结构时查找和检索具有给定键值的10个记录所需的平均磁盘I/O各为多少？假定开始时内存中没有任何存储块，但定位索引块或桶的块时，可以不引入额外的I/O，而只需要检索这些块并将其送入主存的I/O。

习题4.2.6 假定和习题4.2.4一样，一个存储块可存放3个记录，10个键-指针对或50个指针。和例4.16一样，设关系`movie`的属性`studioName`和属性`year`上建有辅助索引。假定有51部Disney制作的电影，101部1995年制作的电影，其中只有一部是Disney制作的。分别计算下列情况下回答例4.16的查询（找出1995年Disney制作的电影）所需的磁盘I/O数：

*a) 两个辅助索引都使用桶，从桶中检索指针，并在主存中求它们的交，然后只检索1995年Disney制作的那一部电影对应的记录。

b) 不使用桶，而使用`studioName`上的索引来取得指向Disney影片的指针，检索它们并选择1995年制作的那些电影记录。假定任意两个Disney影片的记录都不在同一个存储块中。

c) 同（b）一样，但从`year`上的索引开始。假定任意两个1995年制作的电影的记录都不在同一个存储块中。

习题4.2.7 假定我们有一个1000个文档的库，且希望建立一个1 0000个词的倒排索引。一个存储块能容纳10个词-指针对或50个指针，指针可以指向文档或文档的某个位置。词的分布为Zipfian分布（参见7.4.3节中的“Zipfian分布”框）；出现频率排名第i名的词出现的次数是$\lfloor 100000/\sqrt{i} \rfloor$，其中$i$=1，2，…，10000。

*a) 每个文档中平均有多少个词？

*b) 假定我们的倒排索引中只为每个词记录出现该词的所有文档。存放该倒排索引最多需要多少个存储块？

c) 假定我们的倒排索引保存指向每个词的每次出现的指针。存放该倒排索引需要多少存储块？

d) 如果常用的400个词（“无用”字）不包括在索引中，重做b）。

e) 如果常用的400个字不包括在索引中，重做c）。

习题4.2.8 如果我们使用一个扩充的倒排索引，如图 4-20所示，就能执行许多其他类型的查询。说明如何使用这种索引去找到：

*a) “ cat ”和“ dog ”彼此相距不超过五个位置并且出现在同一类元素（如：标题、文本或锚）中的文档。

b) “ cat ”后刚好隔一个位置就跟（有）“ dog ”的文档。

c) 题目中同时出现“ dog ”和“ cat ”的文档。

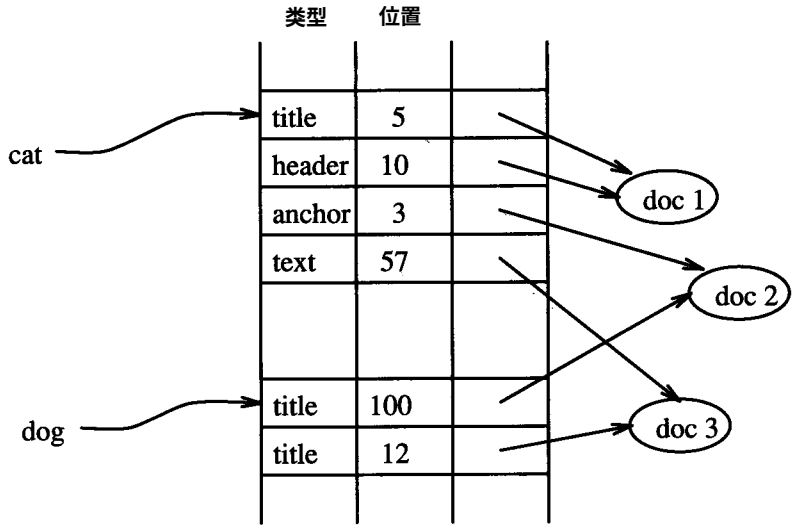

图4-20 在倒排索引中存放更多信息

4.3 B树

虽然一级或两级索引通常有助于加快查询，但在商用系统中常使用一种更通用的结构。这一通用的数据结构簇称为B树，而最常使用的变体称为B+树。实质上：

- B树能自动地保持与数据文件大小相适应的索引层次。
- 对所使用的存储块空间进行管理，使每个块的充满程度在半满与全满之间。这样的索引不再需要溢出块。

在接下来的内容中，我们将讨论“ B树 ”，但具体细节都针对 B+树这一变体。其他类型的 B树在习题中讨论。

4.3.1 B树的结构

正如其名称所暗示的那样， B树把它的存储块组织成一棵树。这棵树是平衡的，即从树根到树叶的所有路径都一样长。通常 B树有三层：根、中间层和叶，但也可以是任意多层。为了对 B树有一个直观的印象，可以先看一下图 4-21、图4-22和图4-23，其中前两个图所示的为B树结点，而后一个图所示的为一个小而完整的 B树。

对应于每个B树索引都有一个参数n，它决定了B树的所有存储块的布局。每个存储块存放n个查找键值和$n+1$个指针。在某种意义上讲，B树的存储块类似于4.1节讲述的索引块，只不过B树的块除了有n个键-指针对外，还有一个额外的指针。在存储块能容纳n个键和$n+1$个指针的前提下，我们把n取得尽可能大。

例4.19　假定我们的存储块大小为4096字节，且整数型键值占4字节，指针占8字节。要是不考虑存储块块头信息所占空间，那么我们希望找到满足$4n+8(n+1)\leq 4096$的最大整数值n。这个值是$n=340$。　□

下面几个重要的规则限制B树的存储块中能出现的东西：

- 根结点中至少有两个指针被使用[⊖]。所有指针指向位于B树下一层的存储块。
- 叶结点中，最后一个指针指向它右边的下一个叶结点存储块，即指向下一个键值大于它的块。在叶块的其他n个指针当中，至少有$\lfloor (n+1)/2 \rfloor$个指针被使用且指向数据记录；未使用的指针可看作空指针且不指向任何地方。如果第i个指数被使用，则指向具有第i个键值的记录。
- 在内层结点中，所有的$n+1$个指针都可以用来指向B树中下一层的块。其中至少$\lceil (n+1)/2 \rceil$个指针被实际使用（但如果是根结点，则不管n多大都只要求至少两个指针被使用）。如果j个指针被使用，那该块中将有$j-1$个键，设为K_1，K_2……，K_{j-1}。第一个指针指向B树的一部分，一些键值小于K_1的记录可在这一部分找到。第二个指针指向B树的另一部分，所有键值大小等于K_1且小于K_2的记录可在这一部分中。依此类推。最后，第j个指针指向B树的又一部分，一些键值大于等于K_{j-1}的记录可以在这一部分中找到。注意：某些键值远小于K_1或远大于K_{j-1}的记录可能根本无法通过该块到达，但可通过同一层的其他块到达。
- 假若我们以常规的画树方式来画B树，任一给定结点的子结点按从左（第一个子结点）到右（最后一个子结点）的顺序排列。那么，我们在任何一个层次上从左到右来看B树的结点，结点的键值将按非减的顺序出现。

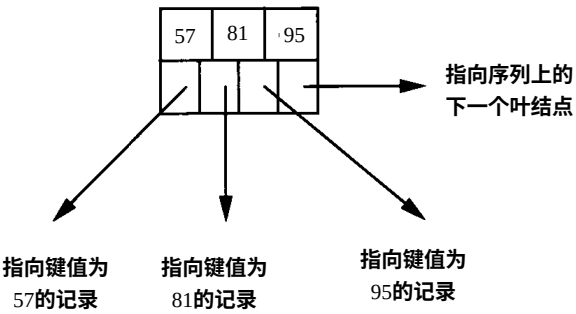

图4-21　典型的B树叶结点

⊖ 从技术上来讲，整个B树的块只有一个指针也是可能的，因为它可能是只有一个记录的数据文件的索引。在这种情况下，整个B树既是根块又是叶块，且这个块只有一个键值和一个指针。在下面的描述中我们忽略这种平凡的情况。

例4.20 在这个例子和其他B树实例中，我们设n=3。也就是说，块中可存放3个键值和4个指针，这是一个不代表通常情况的小数字。键值为整数。图 4-21所示为一个完全使用的叶结点。其中有三个键值57、81和95。前三个指针指向具有这些键值的记录。而最后一个指针，指向右边键值大于它的下一个叶结点，这正是叶结点中通常的情况。如果该叶结点是序列中的最后一个，则该指针为空。

叶结点不必全部充满，但在我们这个例子中，n=3，故叶结点至少要有两个键-指针对。也就是说，图4-21中的键值95和第三个指针可以没有，该指针标有“至键值为95的记录”。

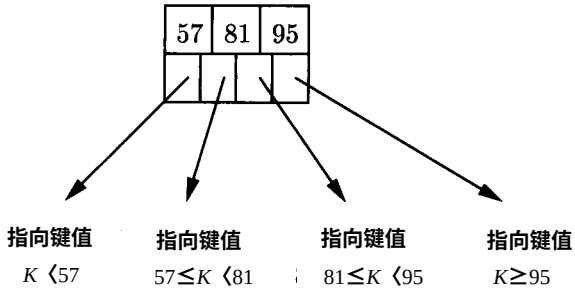

图4-22 典型的B树内部结点

图4-22所示为一个典型的内部结点。其中有三个键值，与我们在叶结点的例子中所选的一样：57、81和95[⊖]。该结点中还有四个指针。第一个指针指向 B树的一部分，通过它我们只能到达键值小于第一个键值即57的那些记录。第二个指针通向键值介于该 B树块第一个键值和第二个键值之间的那些记录，第三个指针对应键值介于该块第二个键值和第三个键值之间的那些记录，第四个指针将我们引向键值大于该块中第三个键值的那些记录。

同叶结点的例子一样，内部结点的键和指针槽也没有必要全部占用。不过，当 n=3时，一个内部结点至少要出现一个键和两个指针。元素缺失最极端的情形就是键值只有 57，而指针也仅使用前两个，在这种情况下，第一个指针对应于小于 57的键值，而第二个指针对应于大于等于57的键值。 □

例4.21 图4-23所示为一棵完整的三层B+树[⊜]；其中使用例4.20中所描述的结点。我们假定数据文件的记录的键是2~47之间的所有素数。注意，这些值在叶结点中按顺序出现一次。所有叶结点都有两个或3个键-指针对，还有一个指向序列中下一叶结点的指针。当我们从左到右去看叶结点时，所有键都是排好序的。

根结点仅有两个指针，恰好是允许的最小数目，尽管至多可有 4个指针。根结点中的某个键将通过第一个指针访问到的键值与通过第二个指针访问到的键值分隔开来。也就是说，不超过12的键值可通过根结点的第一个子树找到；大于等于 13的键值可通过第二个子树找到。

⊖ 虽然键值一样，但图4-21所示的叶结点与图4-22所示的内部结点之间并没有什么联系。事实上，它们不可能出现在同一棵B树中。

⊜ 记住本节讨论的所有B树都是B+树，但在以后提到时我们将省略“+”号。

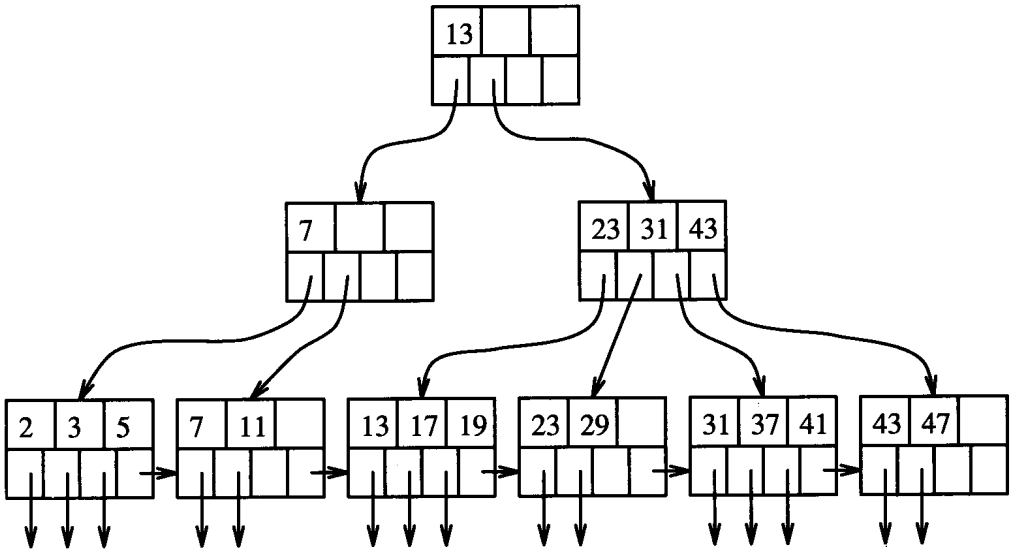

图4-23 B+树

如果我们看根结点的第一个具有键值7的子结点，会发现它有两个指针，一个通向小于7的键，而另一个通向大于等于7的键。注意，该结点的第二个指针只能使我们找到键7和11，而非所有大于7的键，比如键13（虽然我们可以通过叶结点中指向下一个块的指针找到那些更大的键）。

最后，根结点的第二个子结点的4个指针槽都被使用。第一个指针将我们引向一些键值小于23的键，即13、17和19。第二个指针将我们引向键值大于等于23而小于31的所有键；第三个指针将我们引向键值大于等于31而小于43的所有键；而第四个指针将我们引向一些键值大于等于43的键（在这个例子中，是所有的键）。 □

4.3.2 B树的应用

B树是用来建立索引的一种强有力的工具。它的叶结点上指向记录的一系列指针可以起到我们在4.1节和4.2节学过的任何一种索引文件中指针序列的作用。下面是一些实例：

1) B树的查找键是数据文件的主键，且索引是稠密的。也就是说，叶结点中为数据文件的每一个记录设有一个键-指针对。该数据文件可以按主键排序，也可以不按主键排序。
2) 数据文件按主键排序，且B+树是稀疏索引，在叶结点中为数据文件的每个块设有一个键—指针对。
3) 数据文件按非键属性排序，且该属性是B+树的查找键。叶结点中为数据文件里出现的每个属性值K设有一个键-指针对，其中指针指向排序键值为K的记录中的第一个。

B树变体的另一些应用允许叶在结点上查找键重复出现[⊖]。图4-24所示即为这样一棵B树。这一扩展类似于我们在4.1.5节讨论的带重复键的索引。

⊖ 记住，如果“键”必须唯一，那么从这个意义上来说“查找键”不一定是“键”。

如果我们确实允许查找键的重复出现，就需要稍微修改对内部结点中键的涵义的定义，我们曾在4.3.1节中讨论过这一定义。现在，假定一个内部结点的键为 $K_1, K_2\cdots, K_n$, 那么K_i将是从第i+1个指针所能访问的子树中出现的最小新键。这里的“新”，是指在树中第i+1个指针所指向的子树以左没有出现过K_i，但K_i在第i+1个指针指向的子树中至少出现一次。注意，在某些情况下可能不存在这样的键，这时K_i可以为空，但它对应的指针仍然需要，因为它指向树中碰巧只有一个键值的那个重要的部分。

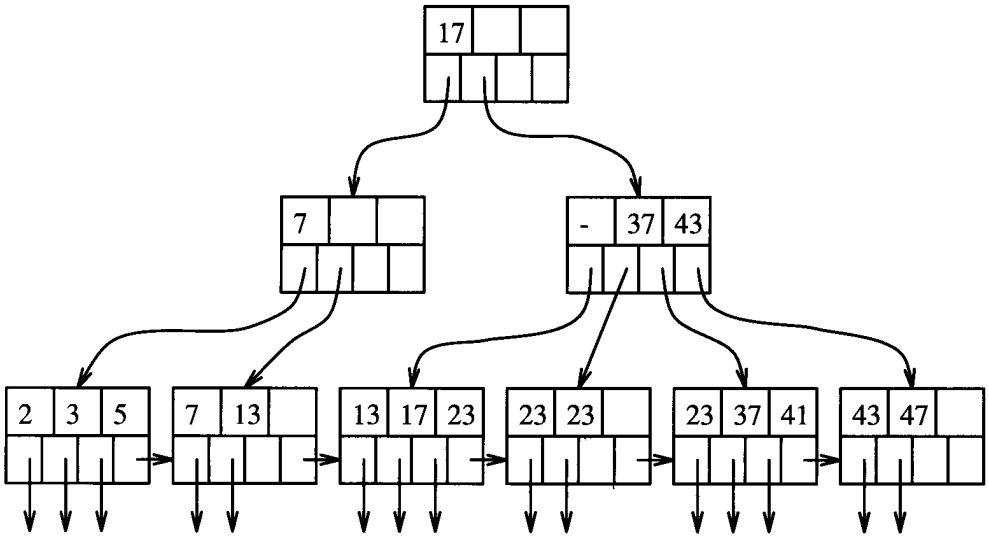

图4-24 一棵带重复键的B树

例4.22 图4-24所示的B树类似于图4-23，但有重复键值。具体来说：键 11已被键13替换；键19、29和31全部被键23替换。这样就造成根结点的键是 17而不是13。原因在于，虽然13是第二个子树根结点中最小的键，但它不是该子树的新键，因为它在根结点的第一个子树中出现过。

我们还需要对根结点的第二个子结点做些改变。第二个键改为 37，因为它是第三个子结点（从左数第五个叶结点）的第一个新键。最有趣的是，第一个键现在为空，因为第二个子结点（第四个叶结点）根本就没有新键。换言之，如果查找某个键且到达根结点的第二个子结点，我们不会从该子结点的第二个子结点起开始查找。若是查找 23或其他更小的值，我们应该从它的第一个子结点起开始查找，在那里我们将找到所需的记录（如果是 17），或找到所需的记录的第一个（如果是23）。

注意：

- 查找13时我们不会到达根结点的第二个子结点，而是直接到第一个子结点中去查找。
- 如果查找介于24~36之间的某个键，我们会直接到第三个叶结点中查找。但当我们连一个所需键值都找不到时，我们就知道不必继续往右查找。举例来说，如果叶结点中存在键 24，它要么在第四个叶结点上，这时根结点的第二个子结点中的空键将会被 24替代；要么在第五个叶结点上，这时根结点的第二个子结点的键 37将被24替代。□

4.3.3 B树中的查找

我们现在再回到最初的假定，即叶结点中没有重复键。这个假定可以简化对B树操作的讨论，但该假定对这些操作来说并非必不可少。假设我们有一个B树索引并且想找出查找键值为K的记录。我们从根到叶递归查找，查找过程为：

基础：若处于叶结点，我们就在其键值中查找。若第i个键是K，则第i个指针可让我们找到所需记录。

归纳：若处于某个内部结点，且它的键为$K_1, K_2, \cdots\cdots, K_n$，则依据在4.3.1节中给出的规则来决定下一步该对此结点的哪个子结点进行查找。也就是说，只有一个子结点可使我们找到具有键K的叶结点。如果$K<K_1$，则为第一个子结点；如果$K_1 \leq K < K_2$，则为第二个子结点，等等。在这一子结点上递归地运用查找过程。

例4.23　假定有一棵如图4-23所示的B树，且我们想找到查找键为40的记录。我们从根结点开始，其中有一个键13。因为$13 \leq 40$，我们就沿着它的第二个指针来到包含键为23、31和43的第二层结点。

在这个结点中，我们发现$31 \leq 40 < 43$，因而我们沿着第三个指针来到包含31、37和40的叶结点，如果数据文件中有键值为40的记录，我们就应该在这个叶结点中找到键40。既然我们没有发现键40，就可以断定在底层的数据块中没有键值为40的记录。

注意，要是查找键为37的记录，我们所做的决定都和上面一样，但当到达叶结点时，我们将找到键37。因为它是叶结点中的第二个键，因此我们沿着第二个指针可以找到键值为37的数据记录。□

4.3.4 范围查询

B树不仅对搜寻单个查找键的查询很有用，而且对查找键值在某个范围内的查询也很有用。一般来说，范围查询在Where子句中有一个项，该项将查找键与单个值或多个值相比较，可用除“=”和“<>”之外的其他比较操作符。使用查找键属性K的范围查询例子如下：

```
SELECT *
FROM R
WHERE R.k > 40;
```

或者

```
SELECT *
FROM R
WHERE R.k >= 10 AND R.k <= 25;
```

如果想在B树叶结点上找出在范围$[a,b]$之间的所有键值，我们通过一次查找来找出键a。不论它是否存在，我们都将到达可能出现a的叶结点，然后在该叶结点中查找键a或大于a的那些键。我们所找到的每个这样的键都有一个指针指向相应的记录，这些记录的键在所需的范围内。

如果我们没有发现大于b的键，我们就使用当前叶结点指向下一个叶结点的指针，并继续检查键和跟踪相应指针，直到我们

1) 找到一个大于b的键，这时我们停止查找；或者

2) 到了叶结点的末尾，在这种情况下，我们到达下一个叶结点且重复这个过程。

上面的查找算法当b为无穷时也有效；即项中只有一个下界而没有上界。在这种情况下，我们查找从键a可能出现的叶结点开始到最后一个叶结点的所有叶结点。如果 a为$-\infty$（即项中有一个上界而没下界），那么，在查找“负无穷”时，不论处于 B树的哪个结点，我们总被引向该结点的第一个子结点，即最终将找到第一个叶结点。然后按上述过程查找，仅在超过键 b时停止查找。

例4.24 假定我们有一棵如图4-23所示的B树，给定查找范围是（10，25）。我们查找键10，找到第二个叶结点，它的第一个键小于 10，但第二个键11大于10。我们沿着它的相应指针找到键为11的记录。

因为第二个叶结点中已没有其他的键，我们沿着链找到第三个叶结点，其键为 13，17和19。这些键都小于或等于 25，因此我们沿着它们的相应指针检索具有这些键的记录。最后，移到第四个叶结点，在那里我们找到键 23。而该叶结点的下一个键 29超过了25，因而已完成我们的查找。这样，我们就检索出了键11到键23的五个记录。 □

4.3.5 B树的插入

当我们考虑如何插入一个新键到 B树时，就会发现B树优于4.1.4节介绍的简单多级索引的一些地方。对应的记录可使用 4.1节中介绍的任何方法插入到建有 B树索引的数据文件中；这时，我们只考虑B树如何相应地修改。插入原则上是递归的：

- 设法在适当的叶结点中为新键找到空闲空间，如果有的话，就把键放在那里。
- 如果在适当的叶结点中没有空间，就把该叶结点分裂成两个，并且把其中的键分到这两个新结点中，使每个新结点有一半或刚好超过一半的键。
- 某一层的结点分裂在其上一层看来，相当于是要在这一较高的层次上插入一个新的键-指针对。因此，我们可以在这一较高层次上逆归地使用这个插入策略：如果有空间，则插入；如果没有，则分裂这个父结点且继续向树的高层推进。
- 例外的情况是，如果试图插入键到根结点中并且根结点没有空间，那么我们就分裂根结点成两个结点，且在更上一层创建一个新的根结点。这个新的根结点有两个刚分裂成的结点作为它的子结点。回想一下，不管 n（结点中键的槽数）多大，根结点总是允许只有一个键和两个子结点。

当我们分裂结点并在其父结点中插入时，需要小心地处理键。首先，假定 N是一个容量为n个键的叶结点，且我们正试图给它插入第（ $n+1$）个键和它相应的指针。创建一个新结点 M，该结点将成为N结点的兄弟，紧挨在 N的右边。按键排序顺序的前 $\lceil (n+1)/2 \rceil$个键-指针对保留在结点N中；而其他的键-指针对移到结点M中，注意，结点M和结点N中都有足够数量的键-指针对，即至少有 $\lfloor (n+1)/2 \rfloor$个这样的键—指针对。

现在，假定N是一个容量为n个键和$n+1$个指针的内部结点，并且由于下层结点的分裂， N正好又被分配给第（$n+2$）个指针。我们执行下列步骤：

1) 创建一个新结点M，它将是N结点的兄弟，且紧挨在N的右边。

2) 按排序顺序将前 $\lceil (n+1)/2 \rceil$ 个指针留在结点n中，而把剩下的 $\lceil (n+1)/2 \rceil$ 个指针移到结点M中。

3) 前 $\lceil n/2 \rceil$ 个键保留在结点N中，而后 $\lfloor n/2 \rfloor$ 个键移到结点M中。注意，在中间的那个键总是被留出来，它既不在结点N中也不在结点M中。这一留出的键K指明通过M的第一个子结点可访问到最小键。尽管K不出现在N中也不出现在M中，但它代表通过M能到达的最小键值，从这种意义上来说它与M相关联。因此，K将会被结点N和M的父结点用来划分在这两个结点之间的查找。

例4.25　让我们在图4-23所示的B树中插入键40。根据4.3.3节中的查找过程找到插入的叶结点。如例4.23一样，我们找到第五个叶结点来插入。由于n=3，而该叶结点现在有四个键-指针对：31、37、40和41，所以我们需要分裂这个叶结点。首先是创建一个新结点，并把两个最大的键40和41以及它们的指针移到新结点。图4-25表示了这个分裂。

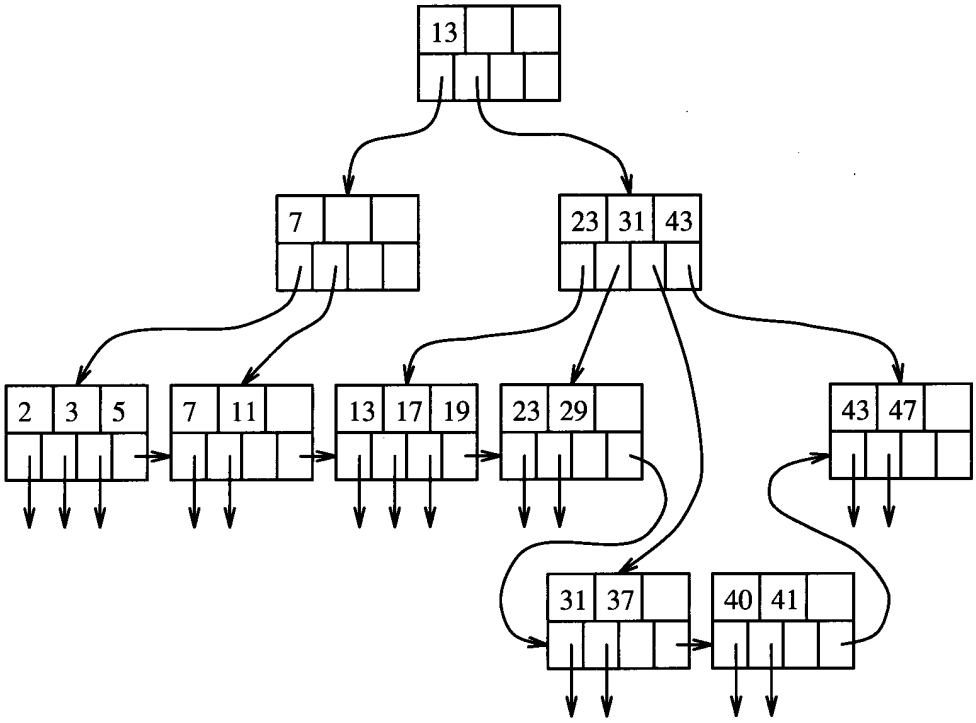

图4-25　键40插入之初

注意，虽然我们现在把这些结点显示在四排，但对树而言还是只有三层，而七个叶结点占据了图中的后两排。这些叶结点通过各自的最后一个指针链接起来，仍形成了一条从左到右的链。

我们现在必须插入一个指向新叶结点（具有键 40和41的那个结点）的指针到它上面的那个结点（具有键23、31和43的那个结点），还必须把该指针与键40关联起来，因为键40是通过新叶结点可访问到的最小键。很不巧，分裂结点的父结点已满，它没有空间来存放别的键或指针。因此，它也必须分裂。

我们开始先找到指向后五个叶结点的指针和表示这些叶结点中后四个的最小键的键列表。

也就是说，我们有指针P1、P2、P3、P4和P5指向这些叶结点，它们的最小键分别是13、23、31、40和43，并且我们用键序列23、31、40和43来分隔这些指针。前三个指针和前两个键保留在被分裂的内部结点中；而后两个指针和后一个键放到一个新结点中。剩下的键是 40，表示通过新结点可访问到最小键。

插入键40后的结果如图4-26所示。根结点现在有三个子结点，最后两个是分裂的内部结点。注意，键40标志着通过分裂结点的第二个子结点可访问到的最小键，它被安置在根结点中，用来区分根结点的第二个子结点和第三个子结点的键。 □

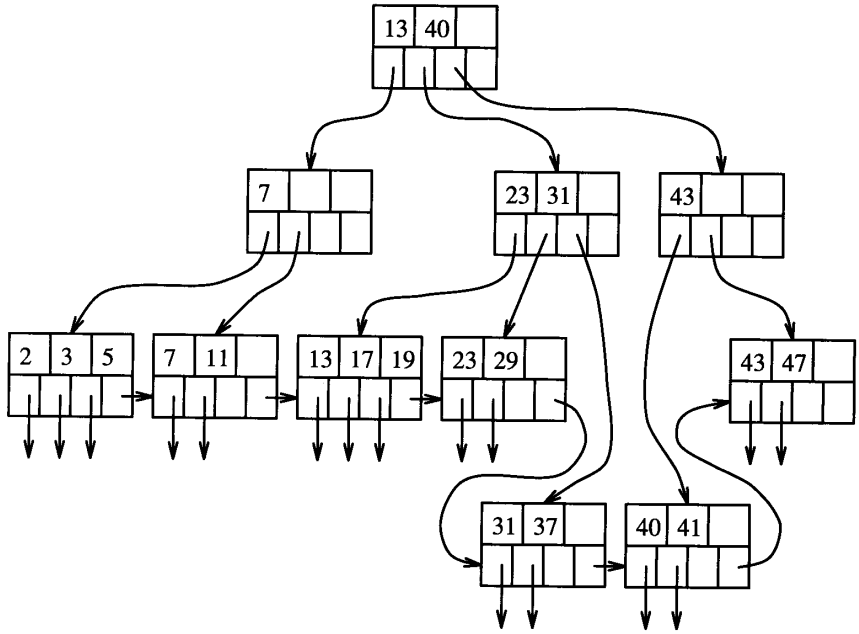

图4-26 键40插入完成后

4.3.6 B树的删除

如果我们要删除一个具有给定键 K 的记录，必须先定位该记录和它在B树叶结点中的键-指针对。正如4.3.3节所述，这部分删除过程主要是查找。然后我们删除记录本身并从 B树中删除它的键-指针对。

如果发生删除的B树结点在删除后至少还有最小数目的键和指针，那就不需要再做什么⊖。但是，结点有可能在删除之前正好具有最小的充满度，因此在删除后，对键数目的约束就被违背了。这时，我们需要为这个键的数目仅次于最小数目的结点 N 做下面两件事之一，其中有一种情况需要沿着树往上递归地删除。

⊖ 如果具有叶结点中最小键的数据记录被删除，那么我们可选择在该叶结点的某个祖先上将某个合适的键增大，但不一定非这样做不可；所有的查找仍能找到正确的叶结点。

1) 如果与结点*N*相邻的兄弟中有一个键和指针超过最小数目，那么它的一个键-指针对可以移到结点*N*中并保持键的顺序。结点*N*的父结点的键可能需要调整以反映这个新的情况。例如，如果结点*N*的右边兄弟*M*可提供一个键和指针，那么从结点*M*移到结点*N*的键一定是结点*M*的最小键。在结点*M*和结点*N*的父结点处有一个表示通过*M*可访问到的最小键，该键必须被提升。

2) 最困难的情况是当相邻的两个兄弟中没有一个能提供额外的键给结点*N*时。不过，在这种情况下，我们有结点*N*和它的一个兄弟结点*M*，其中一个的键数少于最小数，而另一个的键数刚好为最小数。因此，它们合在一起也没有超过单个结点所允许的键和指针数（这就是为什么B树结点最小允许的充满程度为一半的原因）。我们合并这两个结点，实际上就是删除它们中的一个。我们需要调整父结点的键，然后删除父结点的一个键和指针。如果父结点现在足够满，那我们就完成了删除操作，否则，我们需要在父结点上递归地运用这个删除算法。

例4.26　让我们从图4-23所示最初的B树开始，即在键40插入之前。假定我们删除键7。该键在第二个叶结点中被找到。我们删除该键、该键对应的指针以及指针指向的记录。

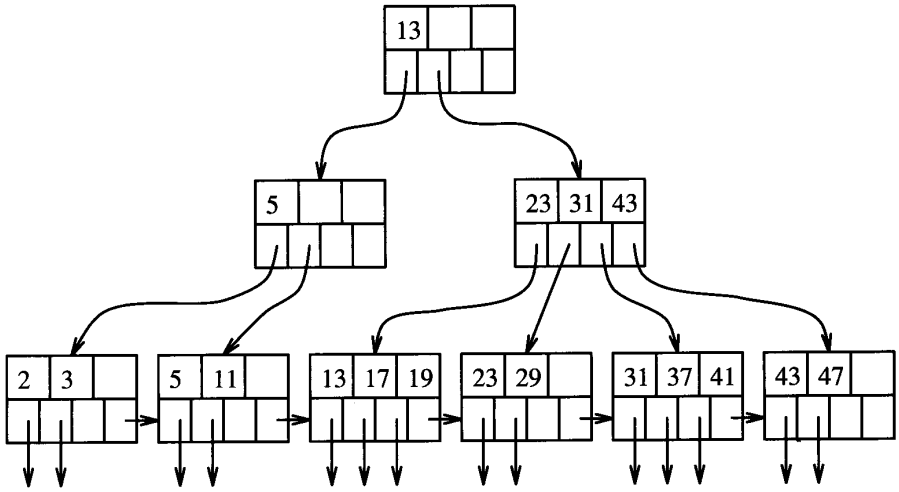

图4-27　键7的删除

不巧的是，第二个叶结点现在只剩下一个键，而我们需要每个叶结点至少有两个键。但该结点左边的兄弟，即第一个叶结点，有一个额外的键-指针对，这就帮了我们的大忙。我们因此可以将它的最大键以及它的相应指针移到第二个叶结点。产生的B树如图4-27所示。注意，因为第二个叶结点的最小键现在是5，所以前两个叶结点的父结点的键从7改为5。

下一步，假定我们删除键11。这个删除对第二个叶结点产生同样的影响；又一次把它的键数减少到低于最小数。不过，这次我们不能从第一个叶结点借键，因为后者的键数也到了最小数。另外，它的右边没有兄弟，也就无处可借⊖。这样，我们需要合并第二个叶结点和它的兄弟，

⊖ 注意：它右边的键为13、17和19的叶结点不是它的兄弟，因为它们有不同的父结点。不论怎样，我们还是可以从那个结点“借”键的，但那样的话，调整键的算法将涉及整个树，因而使算法变得更复杂。我们把这一改进留作习题。

即第一个叶结点。

前两个叶结点剩下的三个键-指针对可以放在一个叶结点中，因此，我们把键 5移到第一个叶结点并删除第二个叶结点。父结点中的键和指针需要进行调整，以反映它的子结点的新情况；具体地说，它的两个指针被换成一个指针（指向剩下的叶结点）且键 5不再有用，也被删除。现在的情况如图4-28所示。

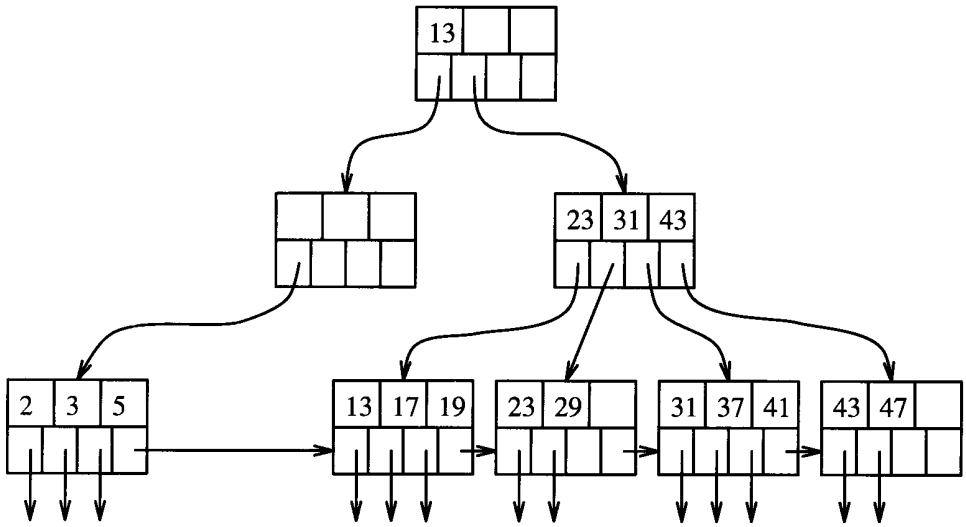

图4-28 键11删除之初

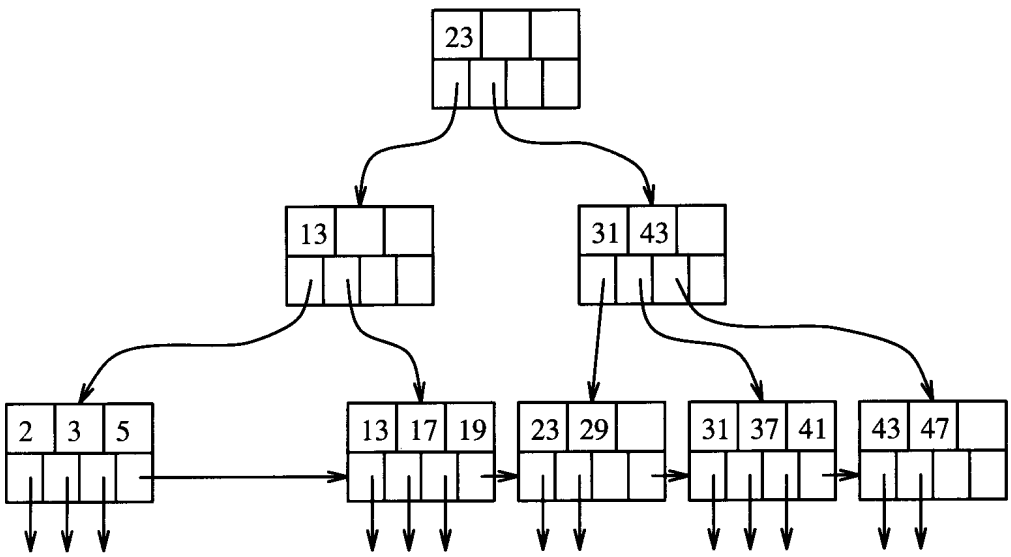

图4-29 键11删除完成后

不幸的是，叶结点的删除给它的父结点即根结点的左子结点带来了负面的影响。正如我们

在图4-28中所看到的那样，该结点现在没有键且只剩一个指针。因此，我们试图从与它相邻的兄弟那里获得一个额外的键和指针。这一次，我们碰到容易的情况，因为根结点的另一个子结点可以提供它的最小键和一个指针。

变化如图4-29所示。指向键为13、17和19的叶结点的指针从根结点的第二个结点移到了它的第一个子结点。我们还修改了内部结点的一些键。键 13原来位于根结点且表示通过那个被转移的指针可访问到的最小键，现在需要放到根结点的第一个子结点中。另一方面，键 23原来用来区分根结点第二个子结点的第一个和第二个子结点，现在表示通过根结点第二个子结点可访问到最小键，因此它被放到根结点中。 □

4.3.7 B树的效率

B树使我们能实现记录的查找、插入和删除，而每个文件操作只需很少的磁盘 I/O。首先我们注意到，如果每个块容纳的键数 *n*相当大，比如10或更大，那么，分裂或合并块的情况将会很少。此外，这种操作必需时，绝大多数时候都被局限在叶结点，因此只有两个叶结点和它们的父结点受到影响。所以，我们基本上可以忽略 B树重组的I/O开销。

然而，每次按给定查找键值查找记录都需要我们从根结点一直访问到叶结点以找到指向记录的指针。因为我们只读B树的块，所以磁盘I/O数将是B树的层数加上一次（对查找而言）或两次（对插入或删除而言）处理记录本身的磁盘 I/O。我们肯定会这样问： B树到底有多少层？对于典型的键、指针和块大小来说，三层就足够了，除非数据库极大。因此，我们一般取 3作为B树的层数。下面的例子说明了其原因。

例4.27 回忆一下我们在例4.19中的分析，我们当时确定每块可容纳示例数据的 340个键-指针对。假若一般的块充满度介于最大和最小中间，即一般的块有 255个指针。一个根结点，有255个子结点，有255^2=65025个叶结点；在这些叶结点中，我们可以有 255^3，即约1.66×10^7个指向记录的指针。也就是说，记录数小于等于 1.66×10^7的文件都可以被3层的B树容纳。 □

不过，对于每次查找，我们甚至可以通过 B树用比3次还少的磁盘I/O来实现。B树根结点块是永久地缓存在主存中的绝佳选择。如果这样，那么每次查找 3层的B树只需两次磁盘读操作。实际上，在某些情况下，把 B树的第二层结点块保存在缓冲区中也是合理的。这样， B树的查找就减少到一次磁盘I/O再加上处理数据文件本身所需的磁盘 I/O。

我们是否该从B树中删除？

有一些B树的实现根本不对删除做修复。如果叶点键和指针太少，这种情况也允许保留。其基本理由在于，大多数文件的发展比较平衡，尽管有时可能出现使键数刚好少于最小数删除操作。但该叶结点可能很快增长并且再次达到键 -指针对的最小数。

此外，如果记录有来自 B树索引外的指针，那么，需要用“ 删除标记 ”来替换记录，并且我们不想用任何方式删除 B树中的指针。在某些情况下，如果可以保证所有对删除记录的访问都将通过 B树，我们甚至可以在 B树叶结点中指向记录的指针处留下删除标记。这样，该记录的空间就可以重新使用。

习题

习题4.3.1 假定存储块能放10个记录或者99个键和100个指针，再假定B树结点的平均充满程度为70%；即有69个键和70个指针。我们可以用B树作为几种不同结构的一部分。对下面描述的每种结构，确定：(1) 1 000 000个记录的文件所需的总块数；(2) 检索一个给定键值的记录所需的平均磁盘I/O数。可以假定最初在主存中不存在任何东西，并且查找键是记录的主键。

*a) 数据文件是按查找键排序的顺序文件，每块存放10个记录。B树为稠密索引。

b) 同a）一样，但组成数据文件的记录没有特定顺序；每块存放10个记录。

c) 同a）一样，但B树为稀疏索引。

!d) B树的叶结点中不放指向数据记录的指针，而是保存记录本身。每块可存放10个记录，但平均每个叶结点的充满度为70%，即每个叶结点存入7个记录。

*e) 数据文件是顺序文件，且B树是稀疏索引，但数据文件的每个基本块有一个溢出块。平均来讲，基本块是满的，而溢出块只半满。不过，记录在基本块和溢出块中没有特定的顺序。

习题4.3.2 假设查询是范围查询且匹配的记录有1000个，在这种情况下，重做习题4.3.1。

习题4.3.3 假定指针占4个字节，而键占12个字节，大小为16 384字节的块可存放多少个键和指针？

习题4.3.4 B树中（1）内结点和（2）叶结点的键和指针的最小数目在下列情况下分别是多少？

*a) n=10；即每块可存放10个键和11个指针。

b) n=11；即每块可存放11个键和12个指针。

习题4.3.5 在图4-23中执行下操作，描述那些引起树改变的操作所带来的变化。

a) 查找键值为41的记录。

b) 查找键值为40的记录。

c) 查找键值在20~30之间的所有记录。

d) 查找键值小于30的所有记录。

e) 查找键值大于30的所有记录。

f) 插入键值为1的记录。

g) 插入键值为14~16的所有记录。

h) 删除键值为23的记录。

i) 删除键值大于等于23的所有记录。

!习题4.3.6 我们提到图4-21所示的叶结点和图4-22所示的内部结点不可能出现在同一棵B树中。解释其原因。

习题4.3.7 当在B树中允许重复键值时，我们在这一节描述的查找、插入和删除的算法都要作一些必要的修改。给出下列情况所需进行的修改：

*a) 查找。

b) 插入。

c) 删除。

!习题4.3.8 在例4.26中我们提出，如果使用更复杂的维护内部结点键的算法，那么可以从右（或左）边的非兄弟结点中借键。描述一个合适的算法，它可以通过从同层相邻结点中借键来重新达到平衡，而不管这些相邻结点是否是键 -指针对太多或太少的结点的兄弟结点。

习题4.3.9 如果我们使用这一节例子中的3个键和4个指针的结点，当数据文件中记录数如下时分别有多少不同的B树：

*!a) 6个记录。

!!b) 10个记录。

!!c) 15个记录。

*!习题4.3.10 假定我们的B树结点可存放3个键和4个指针，如同本节的例子一样。再假定当我们分裂叶结点时，把指针分成 2和2，而当分裂内部结点时，前三个指针到第一个（左）结点，后两个指针到第二个（右）结点。我们从指向键分别为 1、2和3的记录的指针所在叶结点开始，按序加入键值为4,5,6,…等的记录。在插入哪个键时B树将第一次达到四层？

!!习题4.3.11 考虑按B+树组织的一个索引。叶结点一共包含指向 N个记录的指针，且构成索引的每个块有 m个指针。我们希望选择一个 m值，使在具有下列特征的特定磁盘上查找时间最短：

1) 读一给定块到内存的时间大致为 70+0.05m毫秒，其中70毫秒表示读操作的寻道和等待的时间，而0.05m毫秒是传输时间。也就是说，随着 m的增大，块也将变大，因而把块读进内存也就需要更多的时间。
2) 一旦块在内存中，可用二分查找法来找到正确的指针，这样，在内存中处理一个块的时间为$a+b\log_2 m$毫秒，其中a、b为常量。
3) 主存时间常量a比磁盘的寻道和等待时间70毫秒小得多。
4) 索引是满的，因而每次查找需要检查块数为 $\log_m N$。

回答下列问题：

a) 什么样的m值能最小化查找一个给定记录的时间。

b) 随着寻道和等待时间常量（70毫秒）的减少，会发生什么？例如，如果这个常量到一半，最优值m会怎样变化？

4.4 散列表

有许多涉及散列表的数据结构可用做索引。我们假定读者知道用作主存数据结构的散列表。在这种结构中有一个散列函数，它以查找键（我们可称之为散列键）为参数并计算出一个介于 0到$B-1$的整数，其中B是桶的数目。桶数组，即一个序号从 0~$B-1$的数组中包含B个链表的头，每一个对应于数组中的一个桶。如果记录的查找键为 K，那么通过将该记录链接到桶号为 h（K）的桶列表中来存储它，其中h是散列函数。

4.4.1 辅存散列表

有的散列表包含大量记录，记录如此之多，以至于它们主要存放在辅助存储器上，这样的散列表在一些细小而重要的方面与主存中的散列表存在区别。首先，桶数组由存储块组成而不

是由链表头的指针组成。通过散列函数 h散列到某个桶中的记录被放到该桶的存储块中。如果桶溢出，即它容纳不下所有属于它的记录，那么可以给该桶加一个溢出块链以存放更多的记录。

我们将假定，只要给一个 i，桶i的第一个存储块的位置就可以找到。例如，主存中可以有一个指向存储块的指针数组，数组项以桶号为序号。另一种可能是把每个桶的第一个存储块存放到磁盘上某固定的、连续的位置，这样我们就可以根据整数 i计算出桶i的位置。

例4.28 图4-30所示为一个散列表。为了使我们的图例易于处理，假定每个存储块只能存放两个记录，且 $B=4$，即散列函数 h的返回值介于0～3之间。我们列出了一些位于散列表中的记录。在图4-30中，键值为字母a～f。我们假定 $h(d)=0$，$h(c)=h(e)=1$，$h(b)=2$且h$h(a)=h(f)=3$。因此，这六个记录在块中的分布如图所示。

□

注意，图4-30中所示每个存储块的右端都有一个小凸块，这个小凸块表示存储块块头中附加的信息。我们将用它来链接溢出块，并且从4.4.5节开始，我们将用它来保留存储块的其他重要信息。

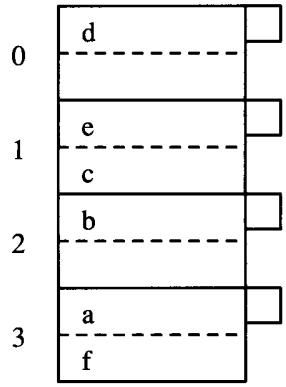

图4-30 散列表

散列函数的选择

散列函数对键的“散列”应使得到的整数类似键的一个随机函数、因此。桶常常能分到相同数量的记录，正如我们将在 4.4.4节中讨论的那样，这能改进访问一个记录的平均时间。另外，散列函数应该容易计算，因为我们要多次计算它。

- 当键为整数时，散列函数的一种常见选择是计算 K/B的余数，其中K是键值，B是桶的数目。通常，B选为一个素数，尽管正如我们将从 4.4.5节开始讨论的那样，将 B选为2的幂也有其理由。
- 当键为字符串时，我们可以把每个字符看作一个整数来处理，把它们累加起来，并将总和除以B，然后取其余数。

4.4.2 散列表的插入

当一个查找键为 K的新记录需要被插入时，我们计算 $h(K)$。如果桶号为 $h(K)$的桶还有空间，我们就把该记录存放到此桶的存储块中或在其存储块没有空间时存储到块链上的某个溢出块中。如果桶的所有存储块都没有空间，我们就增加一个新的溢出块到该桶的链上，并把新记录存入该块。

例4.29 假若我们给图4-30的散列表增加一个键值为 g的记录，并且 $h(g)=1$。那么，我们必须把记录加到桶号为1的桶中，也就是从上面数起的第二个桶。可是，该桶的块中已经有两个记录。因此，我们增加一个新块，并把它链到桶 1的第一块上。键值为 g的记录插入到这一块中，如图4-31所示。 □

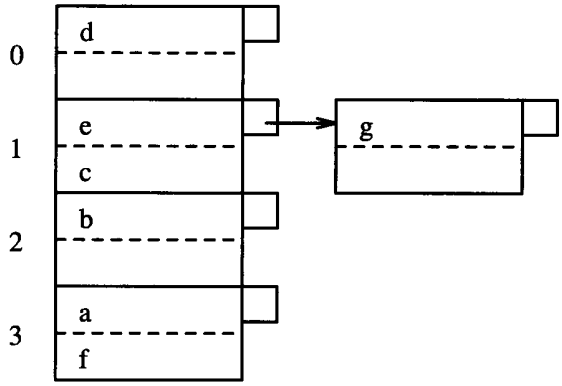

图4-31 为散列表的桶增加另外的一块

4.4.3 散列表的删除

删除查找键值为K的记录方式相同。我们找到桶号为$h(K)$的桶且从中搜索查找键为K的记录，继而将找到的记录删除。如果我们可以将记录在块中移动，那么删除记录后，我们可选择合并同一链上的存储块[⊖]。

例4.30 图4-32所示为从图4-31的散列表中删除键值为c的记录后的结果。由前面可知，$h(c)=1$，因而我们到桶号为1的桶（即第二个桶）中去查看它的所有块，以找出键值为c的一条记录（或所有记录，当查找键不是主键时）。我们在桶1的链表的第一个存储块中找到了该记录。既然现在有可用空间，我们可以把键值为g的记录从链表的第二个存储块移到第一个存储块，并删除第二个存储块。

我们也做了删除键值为a的记录。对于这一键值，我们找到桶3，删除该记录，并把剩下的记录移到块的前部以使之紧凑。 □

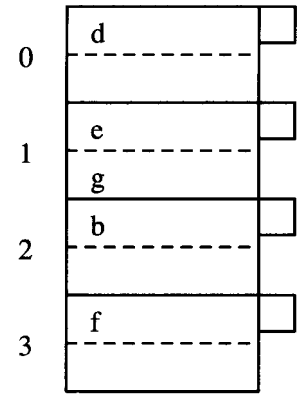

图4-32 散列表的删除结果

4.4.4 散列表索引的效率

理想情况是存储器中有足够的桶，使绝大多数桶都只由单个块组成。如果这样，那么一般的查询只需一次磁盘I/O，且文件的插入和删除也只需两次磁盘I/O。这样的结果比直接用稀疏索引、稠密索引或B树好得多（尽管散列表不能像B树那样支持范围查询；参见4.3.4节）。

但是，如果文件不断增长，那么最终就会出现多数桶的链表中都有许多块的情况。如果这样，我们就需要在块的长链表中查找，每个块至少需要一次磁盘I/O。因此，我们就必须设法减少每个桶的块数。

⊖ 合并一条链上的块随时都要冒风险，这是指当摇摆发生时，即当我们往一个桶里交替地插入或删除记录时，可能每一步都会导致块的创建或删除。

到目前为止，我们学过的散列表都称为静态散列表，因为桶的数目B从不改变。但是，散列表中还有几种动态散列表，它们允许B改变，使B近似于记录总数除以块中能容纳的记录数所得到的商；也就是说，每个桶大约有一个存储块。我们将讨论两种这样的方法：

1) 4.4.5节的可扩展散列；和

2) 4.4.7节的线性散列。

第一种方法在认为B太小时即将其加倍，而第二种方法每当文件的统计数字表明B需要增加时即给B加1。

4.4.5 可扩展散列表

我们的第一种动态散列方法称为可扩展散列表。它在简单的静态散列表结构上主要增加了：

1) 为桶引入了一个间接层，即用一个指向块的指针数组来表示桶，而不是用数据块本身组成的数组来表示桶。
2) 指针数组能增长，它的长度总是2的幂，因而数组每增长一次，桶的数目就翻倍。
3) 不过，并非每个桶都有一个数据块；如果某些桶中的所有记录都可以放在一个块中，那么，这些桶可能共享一个块。
4) 散列函数h为每个键计算出一个K位二进制序列，该K值足够大，比如32。但是，无论何时桶的数目都使用从序列第一位开始的若干位，此位数小于K，比如说是i位。也就是说，当i是使用的位数时，桶数组将有2^i个项。

例4.31 图4-33所示为一个小的可扩展散列表。为简单起见，我们假定K=4，即散列函数h只产生四位二进制序列。当前使用的只有其中一位，正如桶数组上方的框中i=1所标明的那样。因此，桶数组只有两个项，一个对应0，另一个对应1。

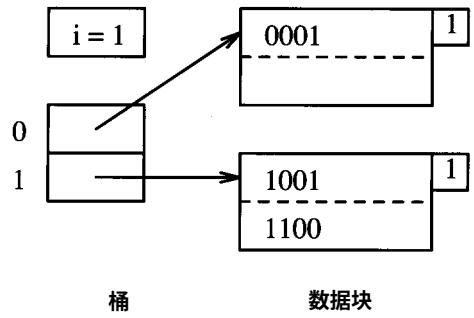

图4-33 可扩展散列表

桶数组项指向两个块。第一块存放当前所有查找键被散列成以0开头的二进制序列的记录；第二个块存放所有查找键被散列成以1开头的二进制序列的记录。为方便起见，我们显示的记录键是散列函数将这些键转换成的二进制位序列。因此，第一块有一个键被散列为0001的记录；而第二个块存放着键分别散列为1001和1100的记录。 □

我们应该注意到，图4-33中每个存储块的“小凸块”中都出现了数字1。这个数字其实出现在每个存储块的块头中，表明由散列函数得到的位序列中有多少位用于确定记录在该块中的成员资格。在例4.31的情况下，只用一个二进制位来确定所有的块和记录，但正如我们将要看到的那样，随着散列表的增长，不同块中需要考虑的位数可能不同。也就是说，桶数组的大小由我们当前正在使用的最大二进制位数来决定，但有些块可能使用较少的位数。

4.4.6 可扩展散列表的插入

可扩展散列表的插入开始时类似静态散列表的插入。为了插入键值为K的记录，我们计算出

$h(K)$，取出这一二进制位序列的前 i 位，并找到桶数组中序号为这个 i 位的项。注意，因为 i 作为散列数据结构的一部分保存，我们能确定 i。

根据数据组中该项的指针找到某个存储块 B。如果 B 中还有存放新记录的空间，我们就把新记录存入，而插入也就完成了。如果 B 中没有空间，那么视数字 i 的不同有两种可能，数字 i 表明散列值中有多少位于确定存储块 B 的成员资格（回忆一下，j 的值可在图中每个存储块的“小凸块“中找到）。

1) 如果 $j<i$，那么不必对桶数组做什么变化。我们：

(a) 将块 B 分裂成两个存储块。

(b) 根据记录散列值的第（$j+1$）位，将 B 中的记录分配到这两个存储块中，该位为 0 的记录保留在 B 中，而该位为 1 的记录则放入到新块中。

(c) 把（$j+1$）存入这两个存储块的小凸块中，以标明用于确定成员资格的二进制位数。

(d) 调整桶数组中的指针，使原来指向块 B 的项指向块 B 或新块，这由项的第（$j+1$）位决定。

注意，分裂块 B 可能解决不了问题，因为有可能块 B 中所有记录将分配到由 B 分裂成的两个存储块的其中一个中去。如果这样，我们需要对仍然太满的块用下一个更大的 j 值重复上述过程。

2) 如果 $j=i$，那么我们必须先将 i 加 1。我们使桶数组长度翻了一倍，因此数组中现在有 2^{i+1}。假定 w 是以前的桶数组中作为某项序号的 i 位二进制位序列。在新桶数组中，序号为 $w0$ 和 $w1$（即分别用 0 和 1 扩展 w 所得到的数）的项都指向原 w 项指向的块。也就是说，这两个新项共享同一个存储块，而存储块本身没有变化。该块的成员资格仍然按原先的位数确定。最后，我们继续像第一种情况中那样分裂 B。由于 i 现在大于 j，所以满足第一种情况。

例 4.32　假如在图 4-33 的表中插入一个键值散列为 1010 序列的记录。因为第一位是 1，所以该记录属于第二个块。然而，该块已满，因此需要分裂。这时我们发现 $j=i=1$，因此首先需要将桶数组加位，如图 4-34 所示。图中我们已将 i 设为 2。

注意，以 0 开头的两个项都指向存放键值散列序列以 0 开头的记录的那个存储块，且该存储块的“小凸块”中数字仍然为 1，这表明该块的成员资格只由位序列的第一位确定。但是，位序列以 1 开头的记录存储块需要分裂，因此我们把这一块中的记录分到以 10 开头和 11 开头的两个存储块中。在这两个存储块中的小凸中有一个 2，表示成员资格用位序列的前两位来确定。幸好，分裂是成功的；既然两个新块都至少有一个记录，我们就不用进行递归分裂。

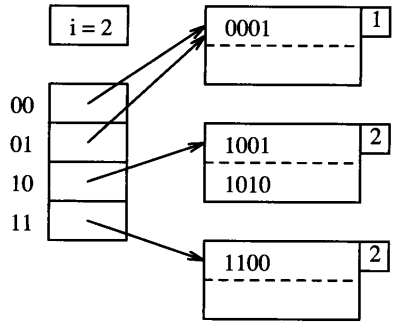

图 4-34　使用两位散列函数值的散列表

现在，假定我们插入键值分别为0000和0111的记录。这两个记录都属于图4-34中第一个存储块，于是该块溢出。因为该块中只用一位来确定其成员资格，而 i=2，所以我们就不用调整桶数组。我们只需分裂该块，让0000和0001留在该块，而将0111存放到新块中，桶数组中由01项改为指向新块。这一次我们又很幸运，所有记录没有全分配到一个块中，所以我们不必递归地分裂。

假若现在要插入一个键值为 1000的记录。对应10的块溢出。由于它已经使用两位来确定其成员资格，这时需要再次分裂桶数组，并且把 i设为3。图4-35给出了这时的数据结构。注意，图中对应10的块被分裂成100的块和101的块，而其他块仍只使用两位来确定成员资格。 □

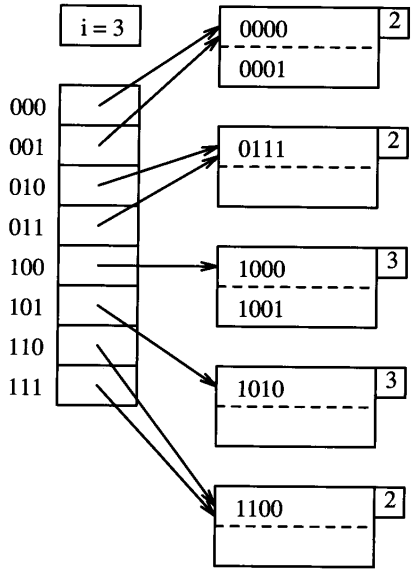

图4-35 使用3位二进制序列的散列表

4.4.7 线性散列表

可扩展散列表有一些重要的好处。最大的好处在于，当查找一个记录时，我们总是只需要查找一个数据块。我们还需要查找到一个桶数组的项，但如果桶数组小到可以存放在主存中，那么访问桶数组就不需要进行磁盘I/O。然而，可扩展散列表也有一些缺点：

1) 当桶数组需要翻倍时，要做大量的工作（当 i很大时）。这些工作会阻碍对数据文件的访问，或是使某些插入看来花费很长的时间。
2) 当桶数翻倍后，它在主存中可能就装不下了，或者把其他的一些我们需要保存在主存的数据挤出去。其结果是，一个运行良好的系统可能突然之间每个操作所需的磁盘 I/O开始大增，并且出现明显的性能下降。
3) 如果每块的记录数很少，那么很有可能某一块的分裂比在逻辑上讲需要分裂的时间提前许多。例如，如果像我们使用的例子一样，块可存放两个记录，即使记录的总数远小于 2^{20}，这也有可能出现三个记录的前 20位二进制位序列。在这种情况下，我们将不得不使

用i=20和一百万个桶数组项，尽管存有记录的块数远小于一百万。

另一种策略称为线性散列，其中桶的增长较为缓慢。在线性散列中我们发现的新要点为：

- 桶数n的选择总是使存储块的平均记录数保持与存储块所能容纳的记录总数成一个固定的比例，如80%。
- 由于存储块并不总是可以分裂，所以允许有溢出块，尽管每个桶的平均溢出块数远小于 1。
- 用来做桶数组项序号的二进制位数是 $[\log 2n]$,其中n是当前的桶数。这些位总是从散列函数得到的位序列的右（低位）端开始取。
- 假定散列函数值的i位正在用来给桶数组项编号，且有一个键值为 K的记录想要插入到编号为$a_1a_2\cdots a_i$的桶中；即$a_1a_2\cdots a_i$是$h(K)$的后i位。那么，把$a_1a_2\cdots a_i$当作二进制整数，设它为m。如果$m<n$，那么编号为m的桶存在并把记录存入该桶中。如果 $n\leq m<2^i$,那么桶m还不存在，因此我们把记录存入桶 $m-2^i-1$，也就是当我们把a_1（它肯定是1）改为0时对应的桶。

例题4.33　图4-36所示为一个n=2的线性散列表。我们目前只用散列值的一位来确定记录所属的桶。按照例4.31建立的模式，我们假定散列函数产生四位，并且用将散列函数作用到记录的查找键上所产生的值来表示记录。

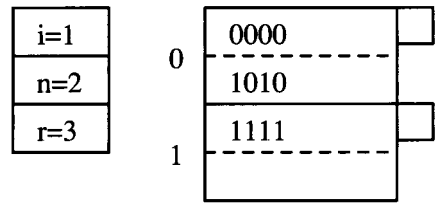

图4-36　线性散列表

我们在图4-36中看到两个桶，每个桶包含一个存储块，桶的编号为0和1。所有散列值以0结尾的记录存入第一个桶，而所有散列值以1结尾的记录存入第二个桶。

参数i(当前被使用的散列函数值的位数)、n（当前的桶数）和r（当前散列表中的记录总数）也是这一结构的一部分。比率 r/n将受到限制，使一般的桶都只需要约一个磁盘存储块。在选择桶数n时，我们采用的策略是使数据文件中记录的个数不超过 $1.7n$，即$r\leq 1.7n$。也就是说，由于每个存储块存放两个记录，桶的平均充满程度不会超过存储块容量的 85%。　□

4.4.8　线性散列表的插入

当插入一个新记录时，我们通过在 4.4.7节提出的算法来确定它所属的桶。也就是说，我们计算$h(K)$，其中K是记录的键，并确定 $h(K)$序列后面用做桶号的正确位数。我们把记录或者放入该桶，或者（在桶号大于等于 n时）放入把第一个二进制由 1改为0后确定的桶中。如果桶中没有空间，那么我们创建一个溢出块，并把它链到那个桶上，并且记录就存入该溢出块中。

每次插入，我们都用当前的记录总数 r的值跟阈值r/n相比，若比率太大，就增加下一个桶到线性散列表中。注意，新增加的桶和发生插入的桶之间没有任何联系！如果新加入的桶号的二进制表示为$1a_2a_3\cdots a_i$，那么我们就分裂桶号为 $0a_2a_3\cdots a_i$的桶中的记录，根据记录的后 i位值分别存入这两个桶。注意，这些记录的散列值都以 $a_2a_3\cdots a_i$结尾，并且只有从右数起的第 i位不同。

最后一个重要的细节是当 n超过2_i时的情况。这时， i递增1。从技术上来讲，所有桶的桶号都要在它们的位序前面增添一个 0，但由于这些位序列被解释成整数，因而就不需要做任何物理上的变化，还是保持原样。

例4.34 我们继续例4.33，考虑插入键值散列为0101的记录时的情况。因为位序列以1结尾，记录属于图4-36中的第二个桶。桶中有空间，因而不用创建溢出块。

但是，由于那里现在有四个记录在两个桶中，超过了1.7这一比率，因此我们必须把 n 提高到3。因为 $[\log_2 3]=2$，我们应该开始考虑把桶0和1改成桶00和01，但不需要对数据结构做任何改变。我们增加下一个桶到散列表中，该桶编号为10，接着，分裂桶00，桶00的序号只有第一位与新加的桶不同。在分裂桶时，键值散列为0000的记录保留在00桶，因为它以00结尾；而键值散列为1010的记录存入桶10，因为它以10结尾，所产生的散列表如图所示4-37。

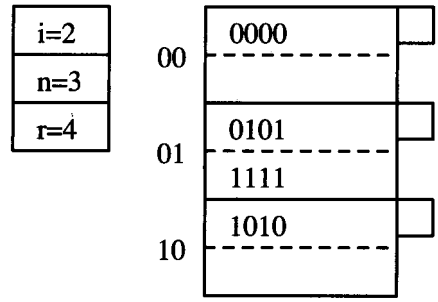

图4-37 增加第三个桶

下面，假定我们增加一个键值散列为 0001的记录。记录最后两位为01，且01桶目前存在，我们把记录存入该桶。不巧的是，该桶的块已经装满，所以我们增加一个溢出块。这三个记录被分配在这个桶的两个块中；我们选择按散列键的数值顺序来保存它们，但这个顺序并不重要。由于该散列表中记录与桶的比率为 5/3，小于1.7，故我们不需创建新桶。所产生的散列表如图 4-38所示。

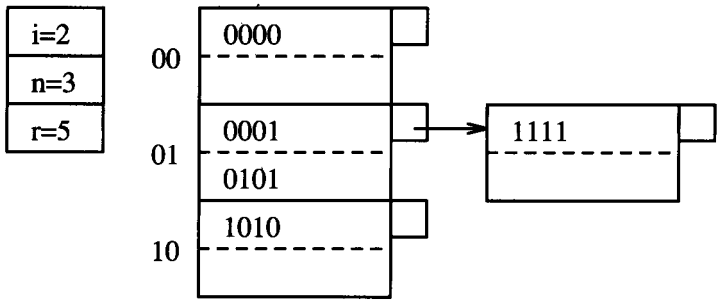

图4-38 必要时使用溢出块

最后，考虑插入键值散列为0111的记录。该记录最后两位为11，但桶11还不存在，因此我们把记录改为存入桶01，该桶号只是在第一位上与桶11不同，不是1而是0。新记录存入到该桶中的溢出块中。

但是，该散列表的记录与桶的比率已超过 1.7，因此，我们必须创建一个编号为11的新桶，该桶碰巧是新记录所需的桶。我们分裂桶01中的四个记录，散列值为0001和0101的记录保留在桶01，而散列值为0111和1111的记录存入新桶。因为桶01现在只有两个记录，我们可以删除其溢出块。现在，散列表如图4-39所示。

注意，当下次插入记录到图4-39中时，我们将会使记录与桶的比率超过1.7。那时，我们将把 n 提到5，并且 i 变成3。 □

例4.35 线性散列表的查询依照我们所描述的选择插入记录所属桶的过程。如果我们希望查找的记录不在该桶中，那么别的地方也不会有所需记录。举例来说，考虑图 4-37中的情形，其

中i=2且n=3。

首先，假定我们想查找键值散列为 1010的记录。由于i=2，我们查看最后两位10，把它们解释为二进制整数，即m=2。因为$m<n$，则编号为10的桶存在，因而我们到该桶中查找。注意，不能因为我们找到一个键值散列为 1010的记录就以为找到了所需的记录，我们需要检查记录的整个键来确定是否所需记录。

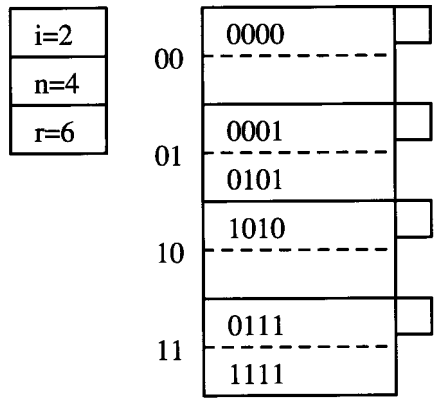

图4-39 增加第四个桶

接着，我们考虑查找键值散列为 1011的记录。现在，我们必须查看编号为 11的桶。由于该桶号作为二进制值整数m=3，并且$m \geq n$，所以桶11不存在。我们通过把第一位1改为0后重新定位到桶01中。可是，桶01中不存在键值散列为 1011的记录，因而我们所需查找的记录肯定不在散列表中。 □

习题

习题4.41 假若在图4-30的散列表中发生下列插入和删除，请说明将产生什么情况：

1) 记录g到记录j分别插入桶0到桶3。

2) 记录a和b记录被删除。

3) 记录k到n分别插入桶0到桶3。

4) 记录c和d被删除。

习题4.4.2 我们没有讨论在线性散列表或可扩展散列表中删除操作如何实现。定位被删除记录的机制应该是显而易见的。你认为应用什么方式实行删除操作？特别地，如果删除后表变小，允许压缩某些块，那么重构散列表有什么优缺点？

!习题4.4.3 本节内容假定索引键是唯一的。不过，只需对这些技术稍微做些修改就可用来处理有重复值的查找键。描述删除、查询和插入算法需做的修改，并说明当重复值出现在下列结构中时带来的主要问题：

*a) 简单散列表。

b) 可扩展散列表。

c) 线性散列表。

!习题4.4.4 实际中有些散列函数并不像理论上那样好。假定我们在整数键值 i上定义一个散列函数$h(i)=i^2 \bmod B$

*a) 如果$B=10$，该散列函数会出现什么问题？

b) 如果$B=16$，该散列函数又有什么好处？

c) 该散列函数对哪些B值有用？

习题4.4.5 在每个存储块可存放n条记录的可扩展散列表中，何时会出现需要递归处理溢出块的情况；即块中的所有记录对应到分裂所产生的两个块中的同一块。

习题4.4.6 假定键值散列为4位序列，就像本节中可扩展散列表和线性散列表的例子一样。但是，假定块中可存放三个记录而非两个记录。如果开始时散列表中有两个空存储块（对应于 0和1），请给出插入键值如下的记录后的结构：

*a) 0000，0001，…，1111，且散列方法是可扩展散列。

b) 0000，0001，…，1111，且散列方法是线性散列，其充满度阈值为 100%。

c) 1111，1110，…，0000，且散列方法是可扩展散列。

d) 1111，1110，…，0000，且散列方法是线性散列，其充满度阈值为 75%。

*习题4.4.7 假定我们使用可扩展散列表或线性散列表模式，但是有指向记录的外部指针。这些指针妨碍了记录在块之间的移动，而这些散列模式中有时需要如此。提出几种能修改结构的方法，从而允许外部指针。

!!习题4.4.8 线性散列模式中使用一个阈值常量 c，使当前桶的数目n和当前记录总数r之间有$r=ckn$的关系，其中k是每块可容纳的记录数。例如，在例 4.33中，我们使用$k=2$和$c=0.85$，因而每个桶的记录数为1.7，即$r=1.7n$。

a) 为了方便起见，假定每个键恰好能按预期的次数出现[⊖]。作为c、k和n的函数，并包括溢出块在内，这种结构需要多少个存储块？

b) 键一般都不会平均分布，而给定键（或键后缀）的记录一般满足泊松分布。也就是说，如果λ是给定键后缀所期望的记录数，那么实际记录数为 i，其概率是$e^{-\lambda}\lambda^i/i!$。在这种假定下，作为c，k和n的函数，期望使用的块数是多少？

*!习题4.4.9 假定有一个100 0000条记录的文件，且我们想把它散列到一个有 1000个桶的散列表中。每个存储块可存放100个记录，并且我们希望块尽可能满，但不允许两个桶共享一个块。存储这一散列表所需的最多和最少的存储块数各是多少？

4.5 小结

- 顺序文件：几种简单的文件组织，其产生方式是将数据文件按某个查找键排序，并在该文件上建立索引。
- 稠密索引：这种索引为数据文件的每个记录设一个键 -指针对。这些键-指针对按它们的键值顺序存放。
- 稀疏索引：这些索引为数据文件的每个存储块设一个键 -指针对。与指针相对应的键为该

⊖ 该假定并不意味着所有存储块都有数量相等的记录，因为有些桶所代表的键是其他桶的两倍。

指针所指向的存储块中第一个键值。

- 多级索引：在索引文件上再建索引，在索引的索引上再建索引，等等，这在有时候是很有用的。高级索引必须是稀疏的。
- 文件的扩展：随着数据文件和它的索引文件的增长，必须采取一些措施来为文件增加附加的存储块。为文件的基本块增加溢出块是一种可行的办法。在数据文件或索引文件的块序列表中插入更多的块，除非要求文件本身存放在连续的磁盘块中。
- 辅助索引：即使数据文件没有按查找键 K排序，我们也可在键 K上建立索引。这样，索引必须是稠密的。
- 倒排索引：文件与其包含的词之间的关系通常可通过一个词-指针对的索引结构来表示。指针指向“桶”文件的某个位置，该位置上有一个指向文件中词的出现的指针列表。
- B树：这些结构实质上是有着很好的扩充性能的多级索引。带有 n个键和$n+1$个指针的存储块被组织成一棵树。叶结点指向记录。任何时候所有索引块都在半满与全满之间。
- 范围查询：指查找键值在给定范围内的所有记录，有索引的顺序文件和 B树索引可为这类查询提供便利，但散列表索引不能。
- 散列表：同创建主存散列表一样，我们也可以基于辅存的存储块来建立散列表。散列函数将键值映射到桶，有效地将数据文件的记录分配到多个小组（桶）。桶用一个存储块和可能出现的溢出块表示。
- 动态索引：如果一个桶中的记录太多，势必降低散列表的性能，因而随着时间推移，桶的数量可能需要增加。允许合理增加桶的两种重要方法是可扩展散列和线性散列。它们都首先将键值散列到一个长位串，然后使用其中若干位来决定记录所属的桶，位的数目是可变的。
- 可扩展散列：这种方法允许在存在记录数太多的桶时将桶的数目加倍。它使用指向块的指针数组来表示桶。为了避免块过多，几个桶可以用同一个块表示。
- 线性散列：这种方法每当桶中的记录比例超出阈值时增加一个桶。由于单个桶的记录不会引起表的扩展，所以在某些情形下需要溢出块。

4.6 参考文献

B树最初是由Bayer和McCreight[2]提出来的。与本章里描述的 B+不同，B树的内部结点和叶结点都有指向记录的指针。[3]是对B树变体的一个综述。

散列用做数据结构可追溯到 Peterson[8]。可扩展散列在 [4]中提出，而线性散列则来自 [7]。Knuth写的书[6]包含了有关数据结构方面的许多内容，包括选择散列函数和设计散列表的技术以及关于B树变体的许多思想。B+树（内部结点没有键值）的明确陈述出现在 [6]的1973版中。

[9]中讨论了辅助索引和文档检索的技术，而 [1]和[5]是有关文本文档索引方法的综述。

1 R. Baeza-Yates, "Integrating contents and structure in text retrieval," *SIGMOD Record* 25:1 (1996), pp. 67-79

2 R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes,"" *Acta Informatica* 1:3 (1972), pp. 173-189

3 D. Comer."The ubiquitous B-tree," *ComputIIng Surveys* 11:2 (1979), pp. 121 13-1

4 R. Fagin, J. Nievergelt,N . Pippenger, and H. R. Strong, "Extendible hashing —a fast access inethod for dynamic files,",

ACM Trans. on Database Systems 4:3 (1979). pp. 315 - 344

5 C. Faloutsos. "Access methods for text," *Computing Surveys* 17:1 (1985), pp. 49-74

6 D. E. Knuth. *Thc Art of Computer Programming*, Vol. III, *Sorting and Scarchling. Thlird Editliotion*, Addison-Wesley, Reading MA, 1998

7 W. Litwin. "Linear hashing: a new tool for file and table addressing," *Proc. hill. Cotif on Very Large Databases* (1980) pp. 212-223

8 W. W. Petersoii. "Addressing for random access storage,"" *IBM J. Resca,rch and Deveclopment* 1:2 (1957). pp. 130-146

9 G. Salton. *Introdactlion to ModCT-n Information Retrieval*, McGraw-Hill, New York, 1983