目录

1 Charpter8 Overview of storage and indexing

在 DBMS 中，数据被抽象为记录的集合，抽象为一个包含一页或更多的页。选择构建一个好的索引集合是一个 DBA 提高性能的最强大工具。

1.1 外部存储中的数据

数据库中的数据一般持久化在外部存储中，诸如硬盘。读取的最小单位是页 (page) , 一般设为 4KB 或 8KB。IO 操作是一个数据库的瓶颈，数据库系统会重点优化。CPU 通过一个称为缓存管理器 (buffer manager) 的软件来操作磁盘上的数据。当文件访问函数需要一些数据时，传给缓存管理器页的 rid，如果缓存管理器中没有该数据，则将磁盘中对应的数据读入缓存管理器。

1.2 文件组织和索引

在 DBMS 中，记录文件是一个重要的抽象，被代码中的文件和获取方法所实现。它还得支持 scan，允许每次访问一个文件中的一个记录。索引是一种通过组织硬盘中数据记录优化数种取回操作的数据结构。一个索引能够允许我们高效得取得满足在该索引构建的域搜索条件的数据。同时我们也可以在其他搜索词上构建额外的所以，帮助提高查找速度。这里有三种方法来存储一个索引记录：

1. 实际的记录数据
2. (k, rid), 其中 rid 是与索引 k 对应的数据记录的记录 id
3. <k, rid-list>, 其中 rid-list 是索引 k 对应的所有记录的 rid 组成的 list

方法 1 能够用来替代排序或未排序文件方法。在构建索引的时候，方法 1 最多只能使用一次，以避免过多的重排序。方法 2 和方法 3 都是使用 rid 指针指向记录，方法三的空间利用效率更高，但是必须维护一个 length 字段来记录 rid-list 的个数。

1.3 聚簇索引

当记录的文件存储顺序和索引数据条目的顺序相同或相近时，我们称之为聚簇索引，否则就是非聚簇索引。当使用上节所讲的方法 1 构建的索引是聚簇索引。而方法 2 和 3 仅当数据记录在文件中按该索引排序的时候，才是聚簇索引。在现实中，在数据更新时，维护有序的操作代价太高昂以至于文件很少保持有序，所以在现实中，一般使用方法 1 构建的索引是聚簇索引，而其他两种是非聚簇索引。

在查找一个范围时，是否为聚簇索引对 io 操作代价有巨大的影响。如果是聚簇索引，那么符合条件的记录被连续地存储在文件上，这样就只需访问有限的几个页即可。然而当不是聚簇索引时，几乎每条记录都要引起一个 page 的读写。

1.4 主、次索引

索引建立的字段中包含主键即为主索引，其他的都为次索引。(在这里需要注意的是，主索引和次索引可能会和前文的方法 1 和方法 2、3 建立的索引术语冲突，因为这些术语并没有标准)

1.5 索引数据结构

索引的数据结构一般包括 hash 和类树结构。

1.5.1 基于 hash 的索引

基于 hash 的索引是构建在某一搜索键值上的，可以想象，当建立在非 uniqe key 上时，索引冲突会发生，此时相同键值的数据以链表的形式连接。通过哈希索引，我们可以在一或两个磁盘 io 就可访问到该 hash 对应的主页 (primary page)。

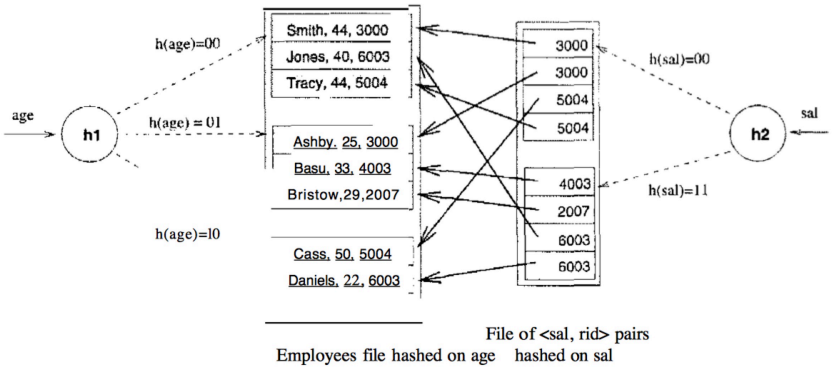

Figure 8.2 Index-Organized File Hashed on *age*, with Auxiliary Index on *sal*

图 1: hash 索引例子

在上图中，左边即为方法 1, 每个 hash 直接指向一个数据记录，右边为方法 2。在右边，每个 hash 指向一条由 (sal, rid) 组成的数据条目。

1.5.2 基于树的索引

j

1.6 文件组织形式的比较

1.6.1 代价模型

本书采用了一个简单的代价模型来衡量数据库操作。其要素如下：

符号	含义
B	page 总数
R	每页的记录数
D	读写磁盘的平均时间
C	处理记录的平均时间

2 Charpter9

2.1 Page format (页格式)

通过抽象，DBMS 上层认为数据是一系列的记录。但是在底层中，可以这么理解：每一页都由很多槽组成，而每个槽 (slot) 包含一条记录。记录由 (page_id, slot_number) 来唯一辨识，即 rid(虽然也可以使用一个 uuid 指向这个元祖，但是考虑到维护成本，一般采用的都是直接使用该元组作为 rid)

2.1.1 固定长度的记录

如果记录的长度是固定的话，那么每个记录槽都是固定长度的，这样就能够连续地放在一个页中。这样，操作的难点就在于如何跟踪这些空槽和如何在一页中遍历所有的记录以及如何处理记录的删除。有如下几种算法：算法 1: 将 N 条 record 记录连续存放在 page 中，当需要删除 record_{i} 时，删除它，然后将 $\text{record}_{\text{N-1}}$ 移到 record_{i} 所在的 slot 上。这样，所有的空闲 slot 都在最后面，此时只要维护一个指针，就可以实现 O(1) 的插入效率。然而当有外部指向的记录被删除时，该方式就不能运作。

算法 2: 通过在页头 (page header) 中加入一个 bit vector, 一对一对应 record slot 是否空闲，来实现。在插入时，需要遍历该 vector 以找到第一个为 0 的 bit。

一般来说，页头中也会包含一些诸如下一页等信息。

2.1.2 变长的记录

因为每个记录是变长的，所以我们再也不能将每一页分为固定个数的 slot。这里有两个问题：

1. 插入: 如何分配合理的空间
2. 删除: 保证删除得到空间在以后能够得以利用。

这里面，在页中移动记录成为至关重要的解决基石。:

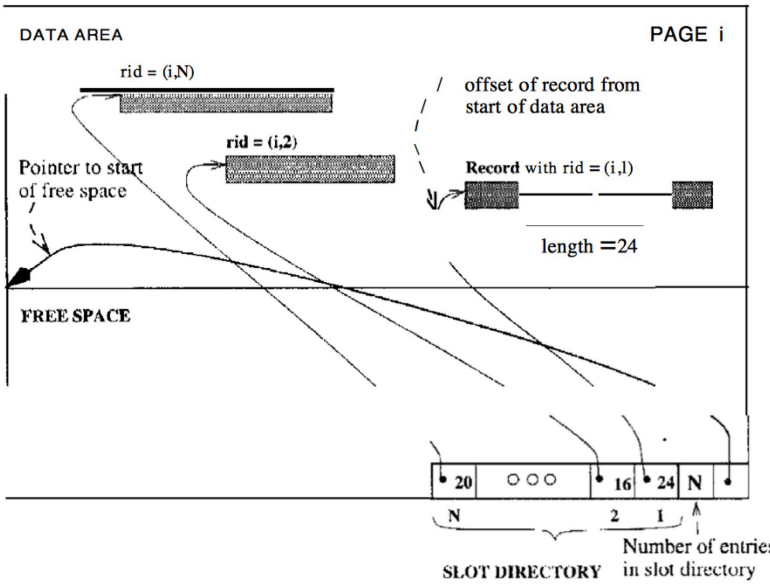

图 2: 变长记录数据组织图

其数据结构可以组织为一下:

```
struct slot_directory{
    int* slot;
    int length; // The length of slot
};
struct page_header{
    int* start_of_free_space;
    int free_space_length; //空闲空间的长度
    int N; // Number of entries in slot directory
    struct slot_directory* slot_ds; //slot direcotry vector
};
```

在删除操作中，直接将 page_header 中的 slot ds 对应的指针置为 null，无其余操作。在插入 $record_i$ 时: 如果 free_space_length > $record_i$.length，则插入。否则，重组该页。

```
def insert(page_header, record_i):
    if record_i.length <= free_space_length:
        page_header.N+=1
        page_header.slot_ds.push(record_i)
        page_header.free_space_length -= record_i.length
    else:
        reorganize(page_header)
        if page_header.length < record_i.length:
            page_header = page_header.next
            insert(page_header, record_i)
        else:
            insert(page_header, record_i)
```

2.1.3 TODO 加入 Record format