索引和查询

2025年6月12日 星期四　10:55

主索引：基于主键建立的索引
辅助索引：基于非主键建立的索引

5.有一个学生表S(sno,sname)，在sname上建立辅助索引，请问在执行如下sql语句时，是否用到了该索引，请解释原因

```
select sname
from s
where sname like '%小%'
123
```

未用到
LIKE 的通配符在前，数据库无法利用索引的有序性

5、已知R（A，B，C），在属性（B，C）上添加索引，那么下面的语句是否用到这个索引，为什么?

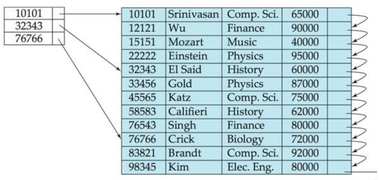
```
select B
from R
where B = 'b'
```

用到了
在(B, C)建立联合索引，根据最左前缀原则，先按B排序，再按C排序

5、R(A,B,C)上有B+树索引(A,B)，先有查询条件为10<A<30，求环境况下的查询代价（n为满足条件的记录的数量，h为B+树的高度）

考虑最坏情况，假设X是辅助索引(AB)
cost = (2h+n) * (ts + tr)

- 如果被删除的记录是具有某个搜索码值的唯一记录，那么这个搜索码值同时也被删除
- 单级索引项删除
  - 稠密索引
    - 搜索码的删除与文件记录的删除类似
  - 稀疏索引
    - 对于对应某个搜索码值的索引项，它被删除是通过用下一个搜索码值替换该索引项来完成的
    - 如果下一个搜索码值已经有一个索引项，此索引项被直接删除



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- 单级索引插入：
  - 用被插入记录的搜索码值进行一次检索
  - 稠密索引 - 如果搜索码值没有出现在索引中，将其插入
  - 稀疏索引 - 如果索引对文件中的每个块只存储一个索引项，除非创建了一个新的块，否则不对索引做任何改变
    - 如果创建了一个新的块，出现在新块中的第一个搜索码值被插入到索引项中
- 多级索引的插入和删除：算法是单级算法的简单扩展
- B^+树索引文件的优点
  - 在数据插入和删除时，能够通过小的自动调整来保持平衡
  - 不需要重组文件来维持性能

	算法	代价	原因
A1	线性搜索	$t_S + b_r * t_T$	一次初始搜索加上 b_r 次块传输，其中 b_r 表示文件中的块数量
A1	线性搜索，码上的等值比较	平均情形 $t_S + (b_r/2) * t_T$	因为最多有一条记录满足条件，所以一旦找到所需的记录，扫描就可以终止。但在最坏的情形下，仍需要 b_r 次块传输
A2	B^+ 树聚集索引，码上的等值比较	$(h_i + 1) * (t_T + t_S)$	（其中 h_i 表示索引的高度。）索引搜索遍历树的高度，再加上一次 I/O 来获取记录；每个这样的 I/O 操作需要一次寻道和一次块传输
A3	B^+ 树聚集索引，非码上的等值比较	$h_i * (t_T + t_S) + t_S + b * t_T$	树的每一层有一次寻道，第一个块有一次寻道。这里 b 是包含具有指定搜索码记录的块数，所有这些记录都是要读取的。假定这些块是顺序存储（因为是聚集索引）的叶子块并且不需要额外的寻道
A4	B^+ 树辅助索引，码上的等值比较	$(h_i + 1) * (t_T + t_S)$	这种情形和聚集索引类似
A4	B^+ 树辅助索引，非码上的等值比较	$(h_i + n) * (t_T + t_S)$	（其中 n 是所获取记录的数量。）在这里，索引遍历的代价和 A3 一样，但是每条记录可能存储在不同的块上，需要对每条记录进行一次寻道。如果 n 值比较大，代价可能会非常昂贵
A5	B^+ 树聚集索引，比较	$h_i * (t_T + t_S) + t_S + b * t_T$	和 A3、非码上的等值比较的情形一样
A6	B^+ 树辅助索引，比较	$(h_i + n) * (t_T + t_S)$	和 A4、非码上的等值比较的情形一样

B^+树
- 码上 聚集/辅助 (hi+1) * (tr+ts)
- 非码
  - 聚集 h^(tr+ts) + ts + b*tr
  - 辅助 (h+n)^(tr+ts)

4、考虑关系 $r_1(A, B, C)$，$r_2(C, D, E)$ 和 $r_3(E, F)$，A, C, E 分别是其主码，假定 r_1 有 1000 个元组，r_2 有 1500 个元组，r_3 有 750 个元组，估计 $r_1 \bowtie r_2 \bowtie r_3$ 的大小，并给出一个计算连接的高效策略。

估计大小：
(1) $r_1 \bowtie r_2$
C是 r_1 和 r_2 的公共属性
∵ r_1 有 1000 个元组，C为外码
r_2 有 1500 个元组，C为主码，假设 C 值均匀分布在 r_1 中
则连结结果大小为 1000（最大值）

(2) $(r_1 \bowtie r_2) \bowtie r_3$
$r_1 \bowtie r_2$ 的结果最多有 1000 条记录，其中有多个 E 值，对应的 r_3.E
∴ 最多有 1000 条

高效策略：

✅ **推荐连接顺序：**

```
text

先做 r1 ⋈ r2，然后 ⋈ r3
```

理由：

- `r2.C` 是主码，先和 `r1` 连接不会产生大中间结果
- `r2.E` 是 `r3` 的连接键，连完 r2 再连 r3 可直接匹配
- 避免把 r2 和 r3 连成一个大表再和 r1 连（代价高）

r_2 同时含有 r_1 和 r_3 的公共部分
∴ 把 r_2 放在中间

类似 S × SC × C

✅ **连接方式建议（物理操作）：**

1. 对 `r2.C` 建索引（或哈希），使用 **Index Nested Loop Join** 加速 `r1 ⋈ r2`
2. 对 `r3.E` 建索引（或哈希），使用 **Index Nested Loop Join** 或 **Hash Join** 处理第二次连接