

Note of Database Review

Key definition

- Primary key: only one attribute
- Super key: a set of attributes that defines a unique instance
- Candidate key: minimum true subset of a super key
- Foreign key: referencing relation outside of the table

Relation algebra

- **Sigma**: select --> select a tuple
- **Pi**: shadow --> select an attribute
- **IXI**: join two table by their common attributes
- **X**: Decal multiplication
- **U**: union of two tuple sets, and eliminates the common tuples
- **rho**: rename, rho x(y) --> to rename y as x
- **minus**: **a - b** means something in **a** but not in **b**.
- **<--**: assign a value to something
- **G**: group function, $G_{\text{sum}(\text{attributeName})}(\text{tableName})$
sum, average, count, count_distinct
 $G_{f_1(a_1), f_2(a_2)}(\text{tableName}), g_i$ means attribute
- **^**: and
- **v**: or

SQL query

- Trigger format:
 - create trigger **triggerName** after/before **insert/delete/update** (of **attributeName**)
on **tableName**
referencing new row as **nrow**
referencing old row as **orow**
for each row
when (**nrow.attributeName** in/not in (
select **attributeName**
from **tableName**)
and **orow.attributeName** in/not in (

select))

begin (**atomic**)

rollback

end;

- alter/drop trigger **triggerName** disable

E-R diagram

- **Arrows**

- $A \text{ --- } B \text{ --- } C$, means b is a relation between A and C, and it's a one-to-one relation;
- $A \text{ --- } B \text{ --- } C$, means multi Cs are related to only one A;
- $A \text{ --- } B \text{ --- } C$, means multi As and multi Cs

- **Weak entity set:** without a primary key; (strong entity set: has a primary key.)

- The discriminator of a weak entity set will be underlined with a dotted line: ---. (discriminator is a set of attributes that defines a unique tuple, aka partial code)
- The relation set between a weak entity set and a strong one will be a double diamond.
- The double line means a multi-value attribute (an attribute that can have more than one value)

- **Relation set**

- Multi-multi, 2 sets related: union of primary keys of the two sets
- One-one, 2 sets: any primary of the two sets
- Multi-one, 2 sets: the primary key of the 'multi' one.
- Multi-multi, n sets: union of the primary keys of all the sets
- Multi-one, n sets: the 'one' is unique in order to have only one meaning, so --- union of the primary keys of multi ones

Functional Dependent

- $A1 \rightarrow A2$ means if $t1[a1] == t1[a1]$, then $t1[a2] == t2[a2]$.
 - t1 does not have to be a super key or primary key
 - A trivial FD means $a2 \subseteq a1$, while an untrivial one does not.

Normal Forms

- 1NF: **all the domains of attributes are atomic**
- BCNF: **all the FDs in F^+ , they are either trivial FDs, or a1 is a super key of relation R**
 - How to decompose a non-BCNF schema:
 - Example: $a \rightarrow b$, but a is not a super key or primary key, and b is not included by a (not trivial);
 - **$a \cup b \rightarrow (R - (b - a))$**
 - Note: only the duplications of FD can be eliminated, not all

- 3NF: weaker than BCNF, by including another possibility saying **the attributes in (b - a) is included in a candidate key of R.**

F+

- F is a set of FDs, and F+ is the complete closure of FDs.
- Armstrong's axiom: used to calculate the F+.
 - **Reflexive rule:** if a is an attribute set and $b \subseteq a$, then $a \rightarrow b$
 - **Augmentation rule:** if $a \rightarrow b$ and y is an attribute set, then $ya \rightarrow yb$
 - **Transitivity rule:** if $a \rightarrow b$ and $b \rightarrow y$, then $a \rightarrow y$
- Addition to Armstrong's axiom: can be proved using the basic 3 axioms.
 - **Union rule:** if $a \rightarrow b$ and $a \rightarrow y$, then $a \rightarrow by$
 - **Decomposition:** if $a \rightarrow by$, then $a \rightarrow b$ and $a \rightarrow y$
 - **Pseudotransitivity rule:** if $a \rightarrow b$ and $yb \rightarrow s$, then $ay \rightarrow s$
- Algorithm of calculating the F+:

```

1  F+ = F
2
3  repeat:
4
5      for FD in F+:
6
7          use reflexive rule and augmentation rule
8
9          add the results in F+
10
11     for FD1 and FD2 in F+:
12
13         if FD1 and FD2 can be related with transitivity rule:
14
15             add the results in F+
16
17 until F+ does not change.
```

- Algorithm to calculate the closure of attribute set a (the closure includes all the attributes dependent of a):

```

1  result = a
2  repeat:
3      for FD(b -> y) in F do:
4          if b ⊆ result:
5              result = result U y
6  until result does not change.
```

Extraneous

- Extraneous: if deleting an attribute does not change the closure F^+ , then the attribute is extraneous.
 - If A belongs to a and F logically imply $(F - \{a \rightarrow b\}) \cup \{(a - A) \rightarrow b\}$, then A is extraneous in a.
 - If A belongs to a and $(F - \{a \rightarrow b\}) \cup \{a \rightarrow (b - A)\}$ logically imply F, then A is extraneous in b.
 - Eg1: if $AB \rightarrow C$ and $A \rightarrow C$, then B is extraneous in $AB \rightarrow C$.
 - Eg2: if $AB \rightarrow CD$ and $A \rightarrow C$, then C is extraneous in $AB \rightarrow CD$.
 - Note: the conditions will **always be true** if the logical implimention is reversed.

Canonical covers

- Canonical cover: a dependent set F_c where F logically imply F_c and F_c logically imply F.
 - All the FD in F_c does not include an extraneous attribute.
 - All the FD in F_c , the left part is unique. (that is to say, $a \rightarrow b$ and $a \rightarrow c$ should be written as $a \rightarrow bc$)
 - Note: canonical covers are not necessarily unique.
- Algorithm to calculate an F_c :

```

1  Fc = F
2      repeat:
3          use Union rule to replace all the FD with same left sets
4          find an FD(a -> b) in Fc where a or b includes an extraneous
           attribute
5          delete the attribute from a -> b
6      until Fc does not change.
```

Decompositions

- Decomposition: $a \mid X \mid b == R$.
- Lossless decomposition
 - Either $a \wedge b \rightarrow a$ or $a \wedge b \rightarrow b$ belongs to F^+ .
 - Also can be mentioned as: **$a \wedge b$ is a super key on a or b.**
- Dependency-preserving decomposition
 -
- BCNF decomposition

```

1  result = [R]
2      done = False
3
4      # calculate F+:
5      .....
6
7      while not done:
8          for Ri in result:
```

```

9         if Ri is not BCNF:
10             find such an a -> b that
11                 1. hold on Ri
12                 2. is not trivial
13                 3. a -> Ri does not belong to F+
14                 4. a and b have no common attributes
15
16             then result = (result - Ri) U (Ri - b) U (a, b)
17         else done = True.

```

- This algorithm is **lossless decomposition** because when we replace Ri with (Ri - b) and (a, b), $a \rightarrow b$, and $(Ri - b) \wedge (a, b) = a$.
- If $a \wedge b \neq \text{empty set}$, the attributes of $a \wedge b$ will not be in (Ri - b), and $a \rightarrow b$ will **NOT** hold on R.

• 3NF decomposition

```

1  Fc = cononical cover of F
2  i = -1
3
4  for FD(a -> b) in Fc:
5      i += 1
6      Ri = ab
7
8  for Rj in R(0:i):
9      if Rj does not include the candidate of R:
10         i += 1
11         Ri = any candidate key of R.
12
13 # optional: delete duplicate relations, until no dup exists.
14 if Rj is included in Rk:
15     i -= 1
16     delete Rj.

```

• 4NF

-

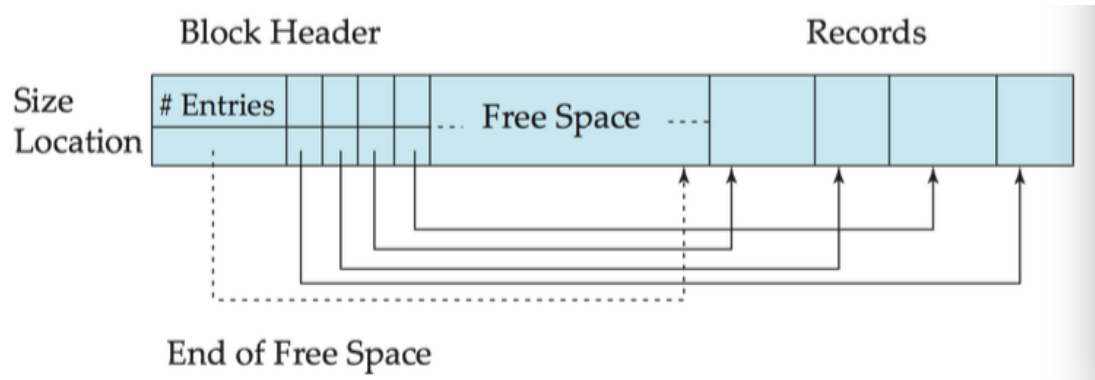
variable length record

• Implementation

- Problems:
 - How to describe a record so that a single attribute can easily fetch it.
 - How to store the record in a block so that the record can be easily fetched.
- **Null bitmap**
 - Overall structure:

Fixed attributes' values + (offset, lengths)s + Null bitmap + variable attributes' values

- Store the fixed-size attributes first, and then the variable ones.
 - At the head of variable part in each record, the attribute's (**offset, length**) information are stored, followed by the value of it.
 - A string of 0 and 1 will tell which attributes are NULL (1 if NULL) after the (offset length) info.
- **Slotted-page structure**



- Each block has a blockhead that stores:
 1. the number of entries;
 2. a pointer to the end of free space in the block;
 3. an array consisting of position and size of each record.
 - Note:
 1. the free space is a continuous space, from the end of the array and the first record.
 2. when inserting a record, the new record will be placed at the end of the free space.
 3. when deleting a record, the size of this record in the array will be set -1, and the records before it will be moved and the pointer to the end of free space will be moved too.
- Big objects are not directly stored in the database, but often stored by their pointers using **B+ tree**.

File storage

- **Heap file organization:** records are placed in random order.
- **Sequential file organization:** records are placed in the order of their searching-code.
- **Hasing file organization:** calculate a hashing function on one attribute of the records and store the records according to the hashing results.
- **Multiple clustering file organization:** several tables are stored in the same file.

Sequential file organization

- Pointer linked list
- Insert:
 1. locate the place to insert a record.

2. if there is enough space to insert, do it; else the record will be inserted into a overflow block.
 3. Anyway, the pointer needs to be adjusted so that the record can be searched with searching-code.
 4. In some cases, the file needs reorganization.
- Delete: using pointer linked list.

Buffer

- **Buffer replacement strategy:** when there's little space left, the **LRU** (least recently used) block will be written to disk files and removed.
- **Pinned block:** when writting on a block, theblock is pinned and should not be written to files.
- **Forced output of block:** when the system break down, the block should be immediately written back to files.
-

B+Tree

- Time complexity: $O(\log_{\text{ceil}(n/2)}(N))$, and n is the maximum number of pointers in a node, and N is the number of records.
- Note: the pointer of index node will point at the leaf whose value equals to the value **ahead of** the pointer.
- Search

```

1  def find(value v):
2      c = root
3      while c != leafNode:
4          let i be the minimum value such that v <= c.k[i]
5          if i > MAXN: # no such value
6              p[m] = the last not null pointer of c
7              c = p[m]
8          elif v == c.k[i]:
9              c = c.p[i+1]
10         else: # smaller than k[i]
11             c = c.p[i]
12
13     # c is a leaf node
14     find c.k[i] = v
15     if i exist:
16         return c, i

```

- Insert
 - Follow the steps of search.
 - If the node can be placed in the node, insert it;

- Else split the node; if the parent node is also full, split it too, and the the inserted new node will go to upper levels.
- Note that every time a node split, the parent needs to add one pointer to point at it.
- Delete
 - Follow the steps of search.
 - Delete the node:
 - If the node is root:
 - If the root only has one child or its root can be merged, delete the node and let the child be root.
 - Change the root value.
 - If the leaf node has two values (deleting one value will make a rather empty node), check the leaf node before it.
 - a. If the sibling (the node before the node to delete) can be merged with it, merge them.
 - b. Else lend one value from the sibling.
 - Change the parent node: **delete** one value if a merge happened, or **alter** one value if a lend happened.
- Calculation

maxn = the maximum of pointers in an index node.

Eg: maxn = 4, 10000 index items (which means 10000 **records**).

- Node —> Height:
 - There are 2-3 keys in a leaf node, and 2-4 children in an index node.
 - minimum height = $\log_4 (10000/3) + 1$.
 - maximum height = $\log_2 (10000/2) + 1$.
- Height —> Node:
 - leafNodes = $\text{ceil}(10000/3)$.
 - Total nodes = $\text{ceil}(10000/3) + \text{ceil}(\text{ceil}(10000/3)/4) + \text{ceil}(\text{ceil}(\text{ceil}(10000/3)/4)/4) + \dots + 1$.
- Bulk load
 - When inserting a lot of tuples, sort them first and insert in order will be more efficient than inserting randomly.
 - When creating an index from current tuples, we can directly build a B+-tree, that is called 'Bottom-up build', which is faster than inserting ('up-bottom method').

Hashing

- Close addressing: use a overflow bucket (implemented by a **linked list**)
- Open addressing: use **linear probing** or other methods if a overflow happened; because delete will be hard for it, a database will **NOT** use this method.

Query execution

- Cost estimation

	算 法	开 销	原 因
A1	线性搜索	$t_s + b_s * t_r$	一次初始搜索加上 b_s 个块传输, b_s 表示在文件中的块数量
A1	线性搜索, 码属性等值比较	平均情形 $t_s + (b_s/2) * t_r$	因为最多一条记录满足条件, 所以只要找到所需的记录, 扫描就可以终止。在最坏的情形下, 仍需要 b_s 个块传输
A2	B* 树主索引, 码属性等值比较	$(h_i + 1) * (t_r + t_s)$	(其中 h_i 表示索引的高度)。索引查找穿越树的高度, 再加上一次 I/O 来取记录; 每个这样的 I/O 操作需要一次搜索和一次块传输
A3	B* 树主索引, 非码属性等值比较	$h_i * (t_r + t_s) + b_s * t_r$	树的每层一次搜索, 第一个块一次搜索。 b_s 是包含具有指定搜索码记录的块数。假定这些块是顺序存储(因为是主索引)的叶子块并且不需要额外搜索
A4	B* 树辅助索引, 码属性等值比较	$(h_i + 1) * (t_r + t_s)$	这种情形和主索引相似
A4	B* 树辅助索引, 非码属性等值比较	$(h_i + n) * (t_r + t_s)$	(其中 n 是所取记录数。)索引查找的代价和 A3 相似, 但是每条记录可能在不同的块上, 这需要每条记录一次搜索。如果 n 值比较大, 代价可能会非常高
A5	B* 树主索引, 比较	$h_i * (t_r + t_s) + b_s * t_r$	和 A3, 非码属性等值比较情形一样
A6	B* 树辅助索引, 比较	$(h_i + n) * (t_r + t_s)$	和 A4, 非码属性等值比较情形一样

图 12-3 选择算法代价估计

Cost of Link

- n = number of records, b = number of blocks.

- **Nested-loop join**

- Worst case:
 - Search times = $n1 + b1$.
 - Transport times = $n1 * b2 + b1$.
- Best case:
 - Search times = 2.
 - Transport times = $b1 + b2$.
- Total = $T_s + T_t$.

- **Block nested-loop join**

- Worst case:
 - Search times = $2b1$.
 - Transport times = $b1 * b2 + b1$.
- Best case:
 - Search times = 2.
 - Transport times = $b1 + b2$.
- Total = $T_s + T_t$.

Very important !!!

- *Improved block nested-loop join*

- 在块嵌套循环连接算法中，外层关系可以不用磁盘块作为分块的单位，而以内存中最多能容纳的大小为单位，当然同时要留出足够的缓冲空间给内层关系及输出结果使用。也就是说，如果内存有 M 块，我们一次读取外层关系中的 $M - 2$ 块，当我们读取到内层关系中的每一块时，我们把它与外层关系中的所有 $M - 2$ 块做连接。这种改进使内层关系的扫描次数从 b_r 次减少到 $\lceil b_r / (M - 2) \rceil$ 次，这里的 b_r 是外层关系所占的块数。这样全部代价为 $\lceil b_r / (M - 2) \rceil * b_r + b_r$ 次块传输和 $2 \lceil b_r / (M - 2) \rceil$ 次磁盘搜索。

- **Indexed nested-loop join**

- $\text{Total} = b_1 * (T_s + T_t) + n_1 * c.$

- **Merge join**

- $\text{Search time} = \text{ceil}(b_1/b_b) + \text{ceil}(b_2/b_b).$
- $\text{Transport time} = b_1 + b_2.$
- Merge sort
 - $\text{Search} = 2 * \text{ceil}(b_r/M) + \text{ceil}(b_r/b_b) * (2 * \text{ceil}(\log \text{floor}(M/b_s) - 1 (b_r/M)) - 1).$
 - $\text{Transport} = b_r * (2 * \text{ceil}(\log M - 1 (b_r/M)) + 1).$
 - $\text{Merge times} = \text{ceil}(\log M - 1 (b_r/M))$

- **Hashing join**

- NOT recursive partitioning:
 - $\text{Search times} = 2 * \text{ceil}(b_1/b_2) + \text{ceil}(b_2/b_b) + 2 * n_h. (B_b: \text{blocks offered to buffer})$
 - $\text{Transport times} = 3 * (b_1 + b_2) + 4 * n_h. (n_h: \text{partitioning number})$
- recursive partitioning:
 - $\text{Search times} = 2 * (\text{ceil}(b_1/b_b) + \text{ceil}(b_2/b_b)) * \text{ceil}(\log M - 1 (b_2) - 1).$
 - $\text{Transport times} = 2 * (b_1 + b_2) * \text{ceil}(\log M - 1 (b_2) - 1) + b_1 + b_2. (M: \text{number of buffers for each partition})$
- Big main storage:
 - $n_h = 0.$
 - $\text{Search times} = 2.$
 - $\text{Transport times} = b_1 + b_2.$

Equivalence rules

- Cascade of sig: $\text{sig cond1}^{\wedge} \text{cond2} (E) = \text{sig cond1}(\text{sig cond2} (E))$
- Commutative: $\text{sig cond1} (\text{sig cond2} (E)) = \text{sig cond2} (\text{sig cond1} (E))$
- Cascade of pi: $\text{pi l1} (\text{pi l2} (\text{pi l3} (\text{pi l4} (...(\text{pi l}_n (E)))))) = \text{pi l1} (E), \text{ where } l_1 \subseteq l_2 \subseteq l_3 \subseteq ... \subseteq l_n.$
-

ACID

- **Atomicity:** done or not done.
- **Consistency:** constrains that make sure for the data is consistent.
- **Isolation:** each transaction is done free of other transactions' disturbance.

- **Durability:** the results will stay still in the DB when the transaction is done.
- Works related to ACID:
 - A - recovery system (commit.)
 - C - constrains (like primary key, unique, etc.)
 - I - concurrency-control system (locks.)
 - D - recovery system (write the changes to disk files.)

Serializable

- Precedence graphs
 - Rules: (assume operation t1.a is ahead of operation t2.b)
 - If a = r and b = w, draw t1 -> t2.
 - If a = w and b = r, draw t1 -> t2.
 - If a = w and b = w, draw t1 -> t2
 - In a word, if a or b is write, draw a line between the former(a) to the latter(b).
 - Note: if there is no **loops** in the graph, then the schedule is serializable and vice versa.
- If a schedule is serializable, give the series according to the graph, in the format of <T1, T2, T3>.

Schedule and Isolation

- Status of a schedule
 - **Active:** initial status.
 - **Partially committed:** after the last query is being executed.
 - **Failed:** fail to execute.
 - **Aborted:** failed and rolled back.
 - **Committed:** finished.
 - **Terminated:** committed or aborted.

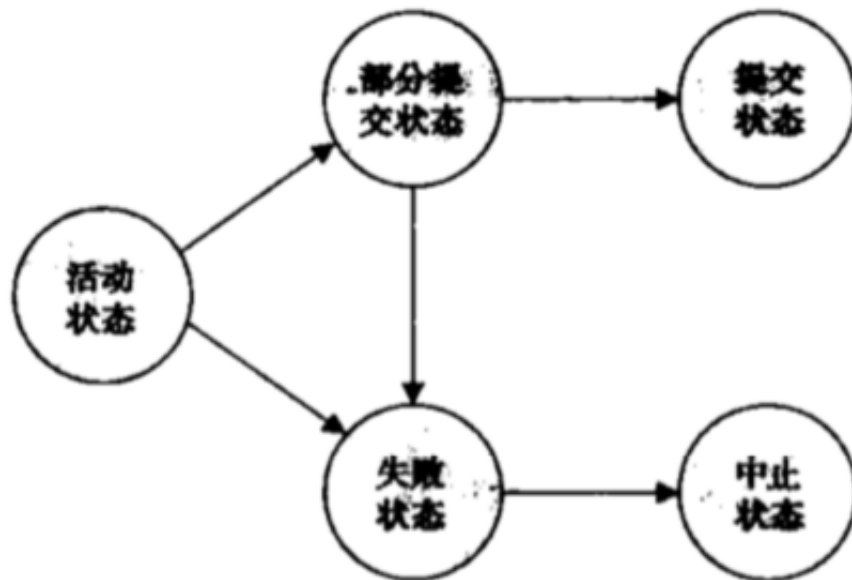


图 14-1 事务状态图

- **Recoverable:** A recoverable schedule is one where, for each pair of Transaction T_i and T_j such that T_j reads data item previously written by T_i the commit operation of T_i appears before the **commit** operation T_j .
- **Cascadeless:** A cascadeless schedule is one where for each pair of transaction T_i and T_j such that T_j reads data item, previously written by T_i the commit operation of T_i appears before the **read** operation of T_j . (if not, once T_i fails, all the transactions dependent on T_i will have to roll back, and that is called cascade rollback.)
- **Every Cascadeless schedule is also recoverable schedule.**
- **Class of Isolation** (from highest to lowest)
 - Serializable
 - Repeatable read
 - Read commit
 - Read uncommitted

Lock

- Types of lock
 - **S** — shared lock: read but not write
 - **X** — exclusive lock: no read or write
 - S and S are compatible, but S and X / X and X are not.
 - A transaction can request for a lock, but before the system give it the lock, the transaction has to wait for the previous owner of lock to unlock it.
 - If a transaction is waiting for so long that it can never be executed, it is called **starve**.
- **2PL** — two-phase locking protocol
 - **growing** phase: a transaction can get a lock but not unlocking one.
 - **shrinking** phase: a transaction can unlock a lock but not getting one.
 - lock point: after getting the last lock.

- Lock conversion: an S-lock can be upgraded to X during growing phase, and an X-lock can be downgraded to S during shrinking phase.
- Strict mode: X-locks have to be unlocked **after** the transaction is committed.
- Rigorous mode: Each lock has to be unlocked **after** the transaction is committed.
- An easier mode:
 - When T_i tries to read, system will add a lock-S before executing the read operation.
 - When T_i tries to write, system will check if the item is under a lock-S, and will either:
 - Upgrade the lock-S;
 - Add a lock-X;
 and then do the write operation.
 - When T_i is terminated (either committed or failed), the locks owned by it will be unlocked.
- Lock manager:
 - Each data item has a linked list.
 - Black rectangle means the lock has been granted, and white ones means waiting for granting.
 - How to maintain:
 -
 - Eg:
 - data item: I7, I23, I912, I4, I44.

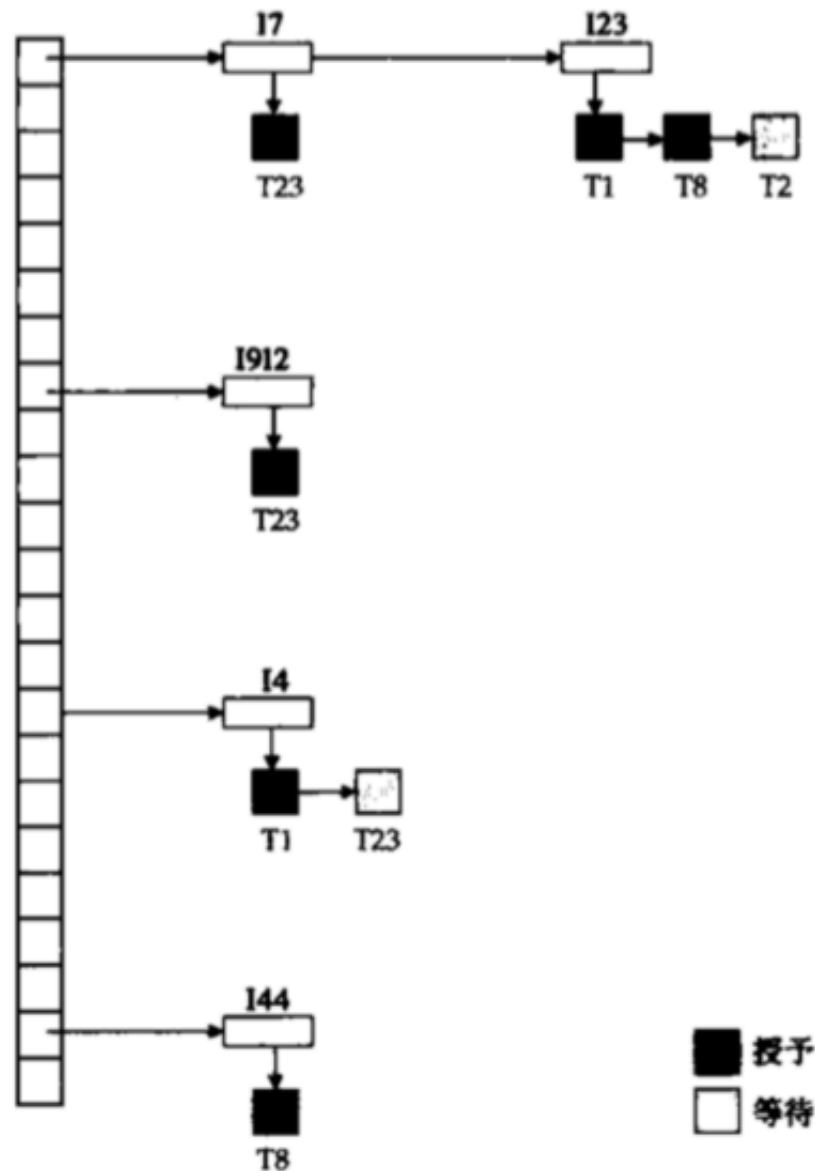


图 15-10 锁表

-

XML — some examples

- **Head of XML file:** `<xs: schema xmlns:xs = "http://www.fuckDatabase.com">`
- **Sequence of attributes:** `<xs: sequence>...</xs: sequence>`
- **Ways to define a relation set:**
 - `<xs: element name = "xxxx" type = "xs: string" // "xs: decimal" / "xxxx_type"(DIY type) />`
 - `//.....definition of other things.....`
 - `<xs: complexType name = "xxxx_type">`
 - `//.....definition of xxxx_type.....`
 - `</ xs: complexType>`
 - `</ xs: element>`

- `<xs:element name = "xxx">`

`<xs:complexType>`

//.....definition of xxx_type.....

`</ xs: complexType>`

- **MAX and MIN occurrence:** `<xs: element minOccurs = "0" maxOccurs = "unbounded">`

- **End of XML file:** `</ xs: schema>`

DTD

-