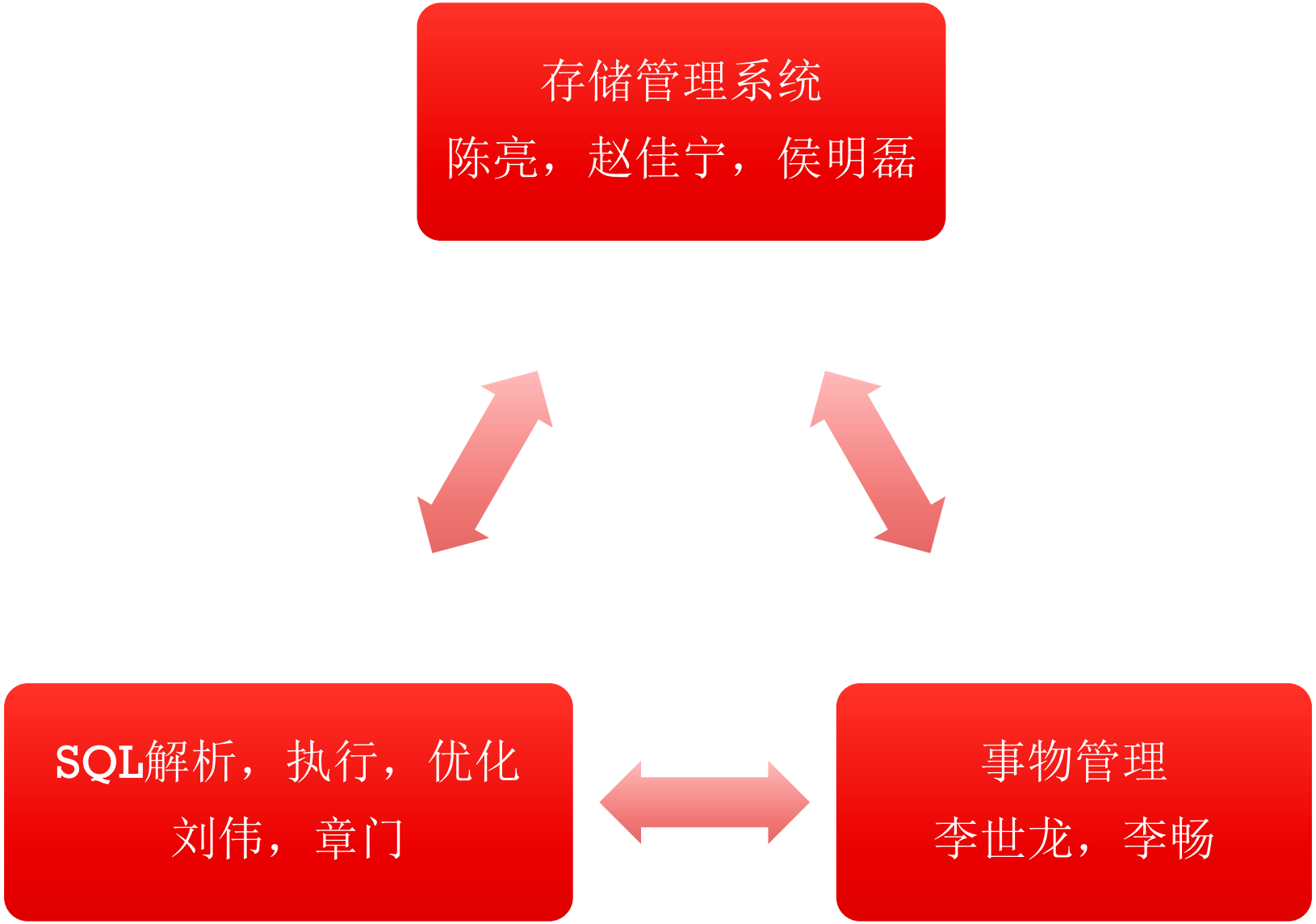
数据库系统实现

组长：陈亮

组员：李世龙，李畅，刘伟，章门，赵佳宁，侯明磊


存储管理系统
陈亮，赵佳宁，侯明磊
SQL解析，执行，优化
刘伟，章门
事物管理
李世龙，李畅

数据库存储管理系统实现

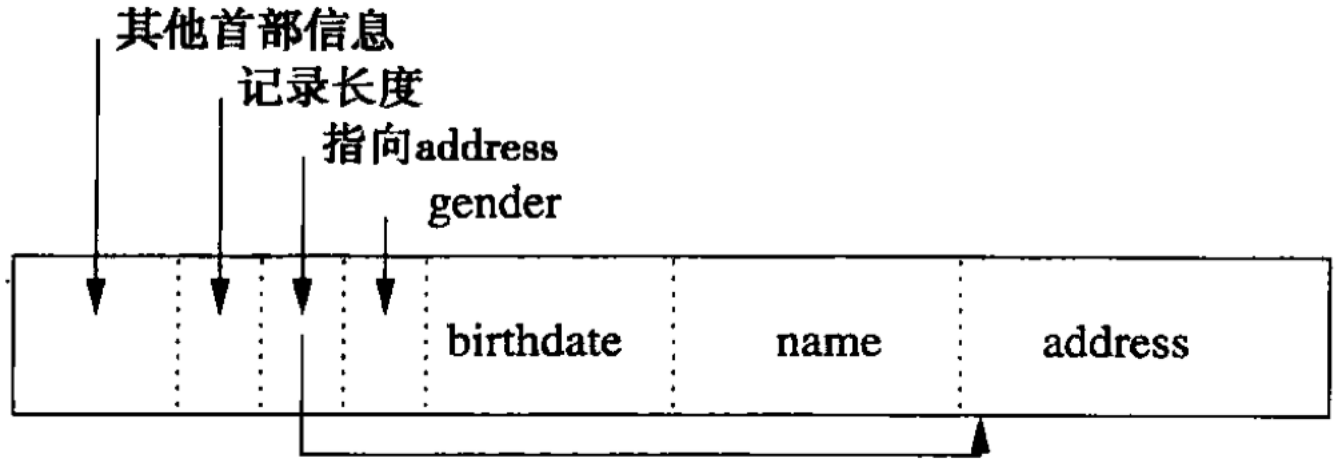
陈亮 赵佳宁 侯明磊

目录

辅助存储管理

记录的物理结构

- 将所有定长字段放在变长字段之前



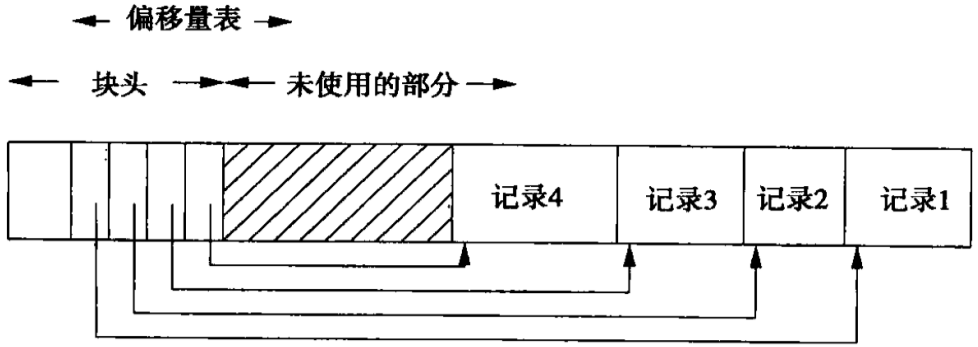
定长：gender, birthday, name
变长：address

块内指针&偏移量

为什么要指针？

- 每一个块，每一条记录都有两种地址形式：磁盘物理地址和虚拟内存地址。如果该数据项被拷贝到缓冲区中，则使用虚拟内存地址；当数据在磁盘中时，使用物理地址。
- 指针和地址经常是记录的一部分，可以避免记录的冗余。

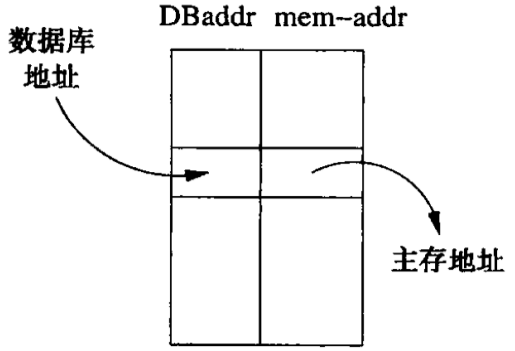
偏移量？



1.可以移动记录
2.存储变长记录
3.删除标记，易于删除

指针混写

- 什么是指针混写？
- 当数据在磁盘中时，我们肯定使用磁盘地址；但是当数据在内存中时，通过虚拟内存地址和物理地址我们都可以找到数据项。让指向该数据的指针使用内存地址更为高效。
- 我们需要一个表将当前虚拟内存中的所有磁盘地址转为当前的内存地址，并且更新指针为指向内存地址的指针

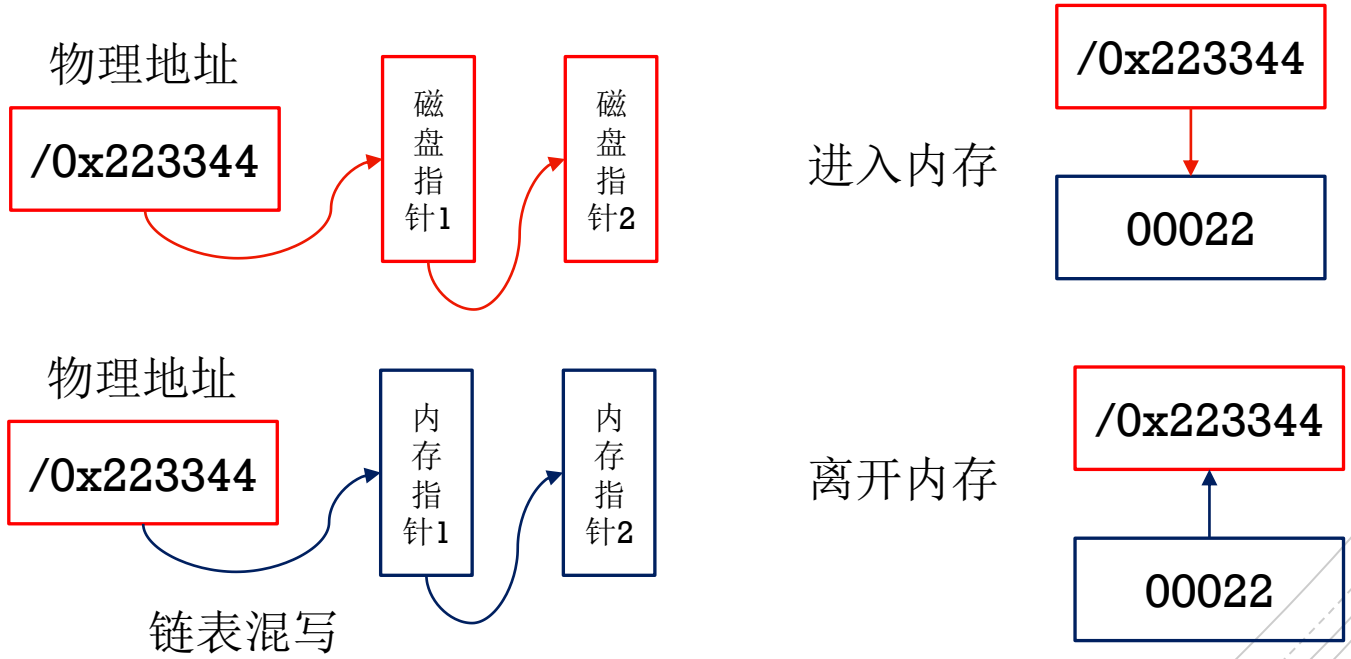


- 指针混写：避免将数据库地址重复转换为内存地址；当我们把块从磁盘移动到内存时，块内指针可以混写，从磁盘地址空间转换到虚拟内存地址空间

混写策略（自动混写）

- 当块内指针指向的数据在内存中时，将磁盘指针转换为内存指针。
- 需要知道：

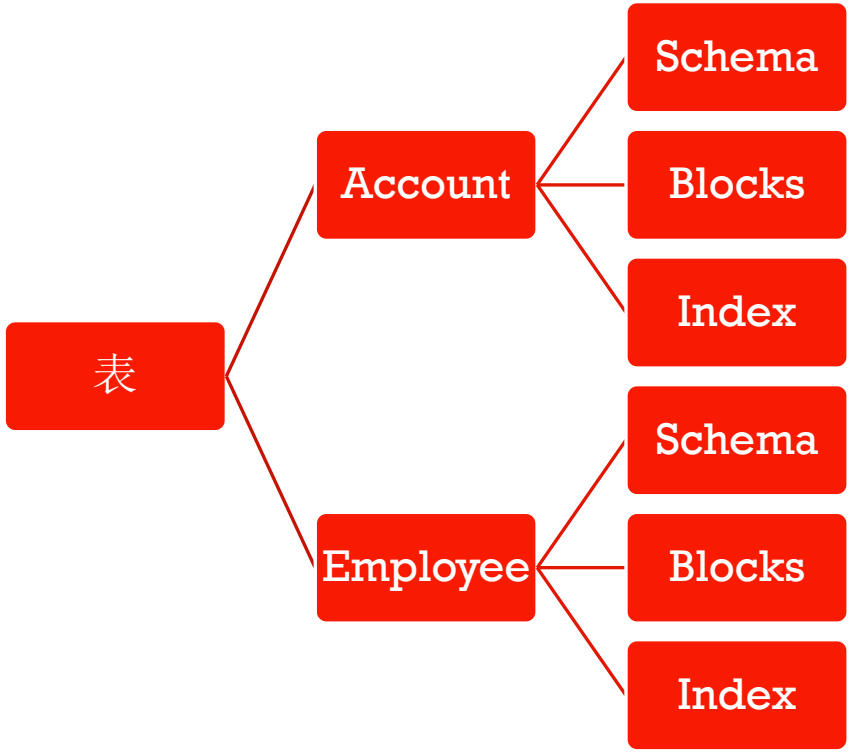
1. 哪些数据在内存中 以及其对应的 内存地址和磁盘地址(转换表)
2. 内存块中的指针指向哪些磁盘地址(链表)

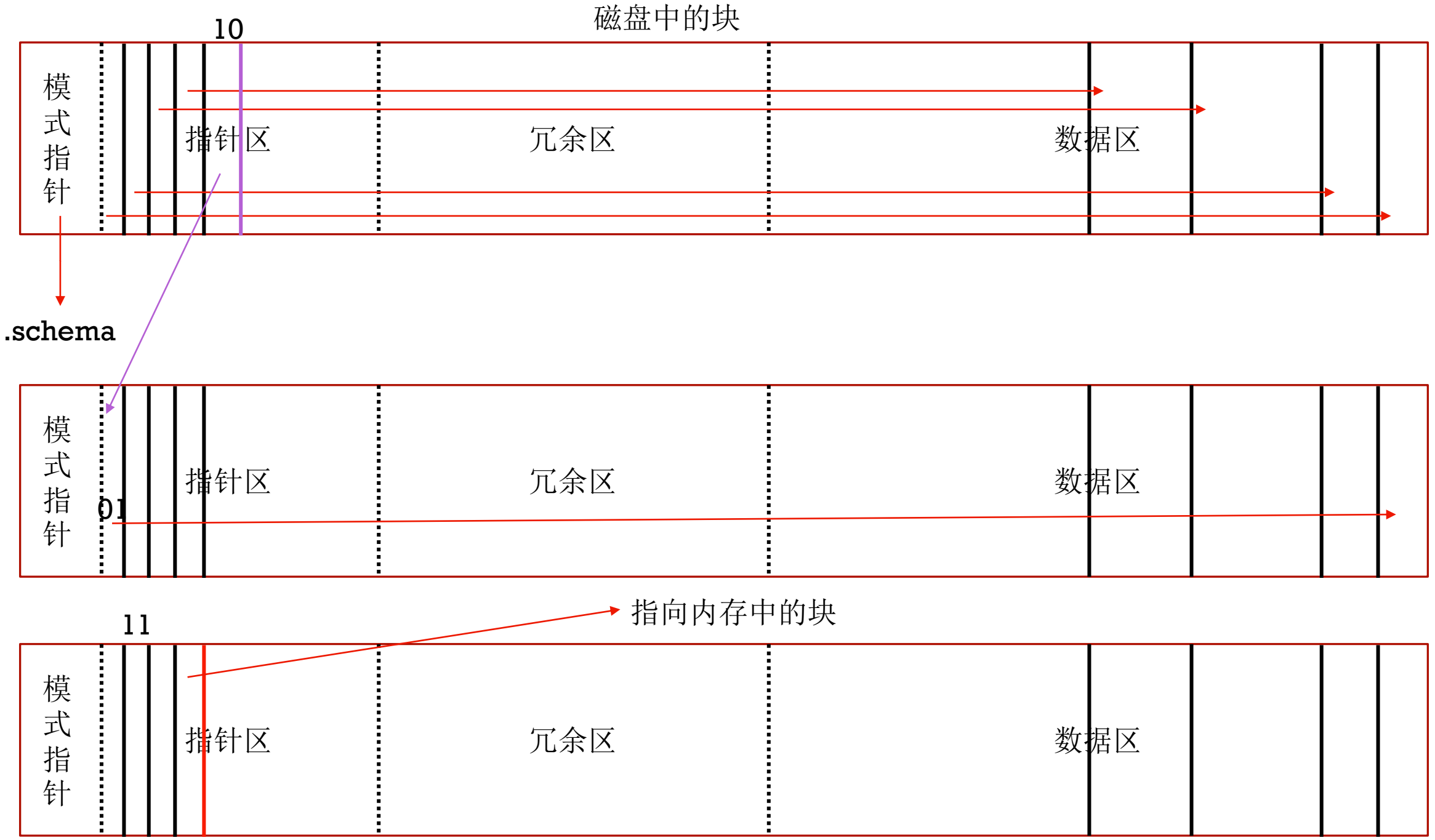


三重指针混写 (New)

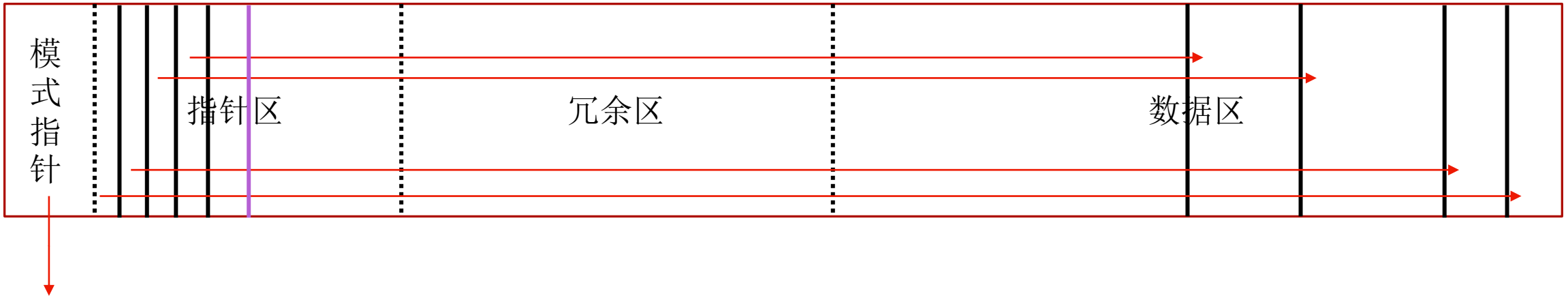
结合指针混写 和 偏移量表

- 用2个二进制位做标志位：
- 00：空指针 10：磁盘指针
- 01：块内偏移量 11：内存指针

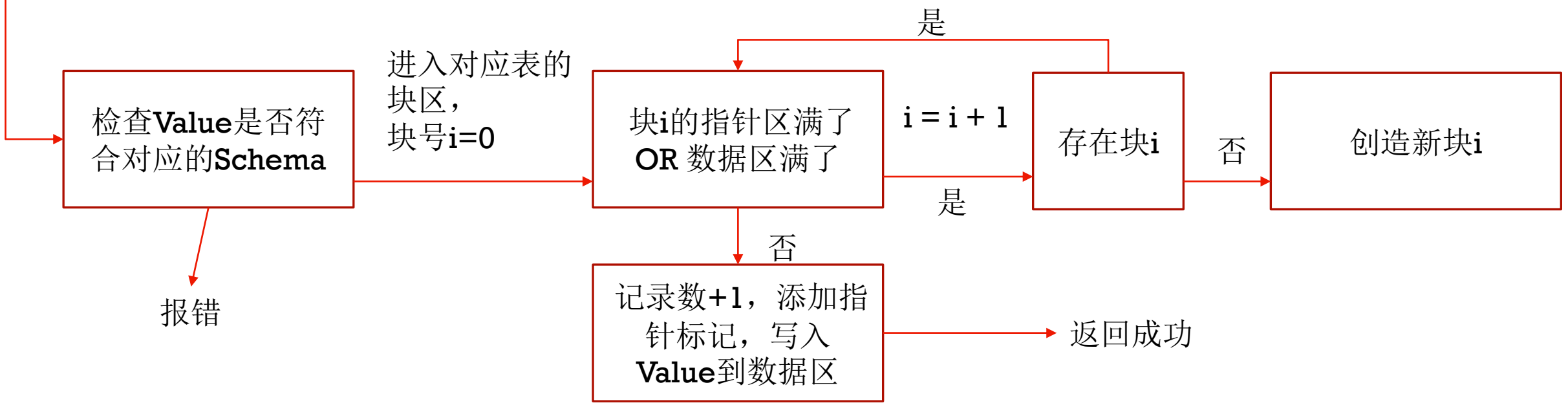



磁盘中的块
10
模式指针
指针区
冗余区
数据区
.schema
模式指针
01
指针区
冗余区
数据区
指向内存中的块
11
模式指针
指针区
冗余区
数据区

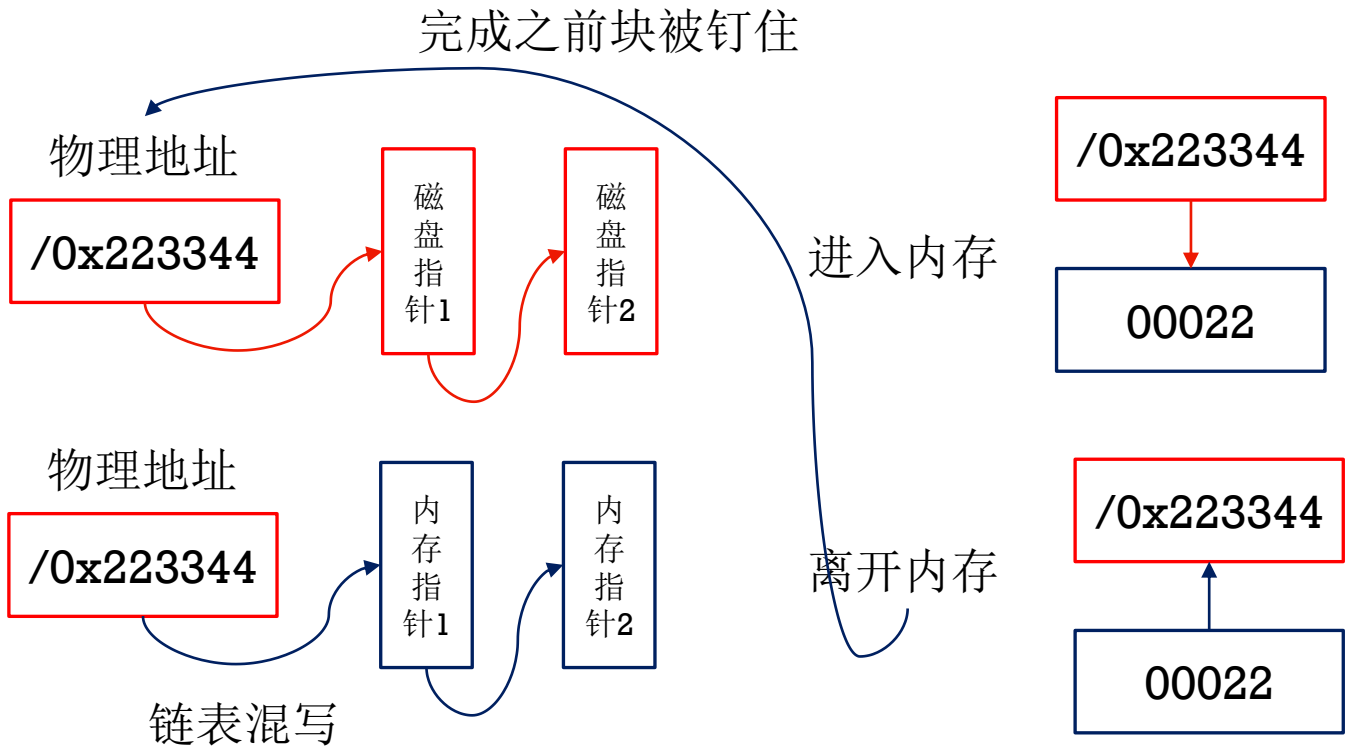
INSERT INTO TABLE_A VALUE(a, b, c)



.schema



被钉住的块1


完成之前块被钉住
物理地址
/0x223344
磁盘指针1
磁盘指针2
进入内存
/0x223344
00022
物理地址
/0x223344
内存指针1
内存指针2
离开内存
/0x223344
00022
链表混写

索引结构

索引，B+树

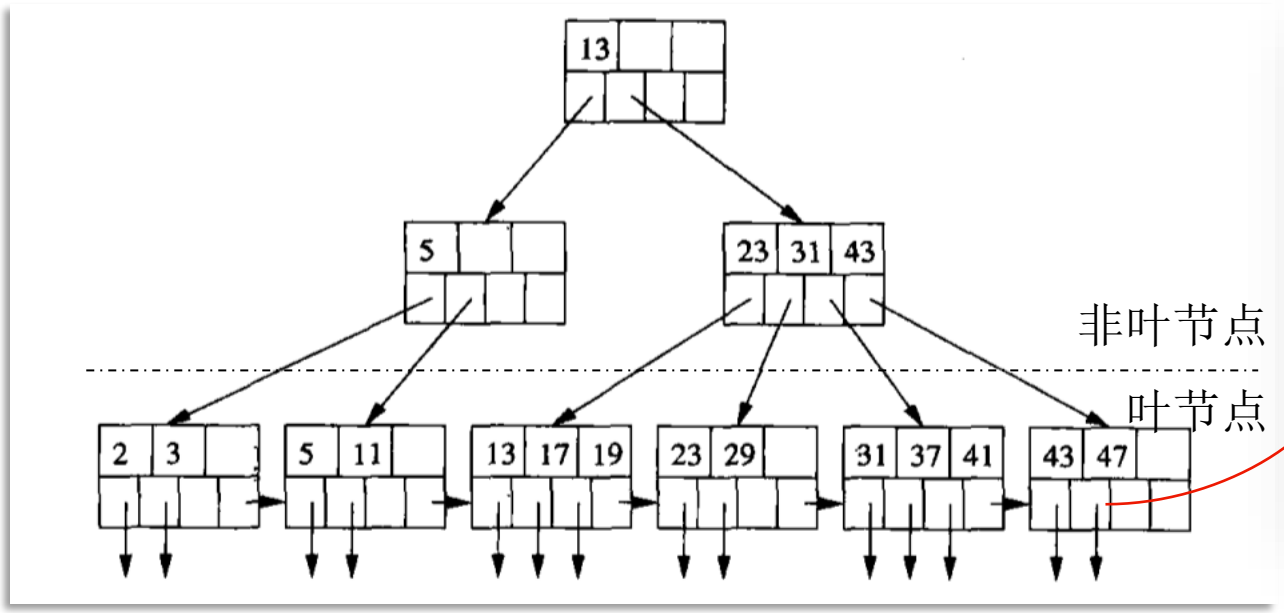
顺序索引文件的缺点

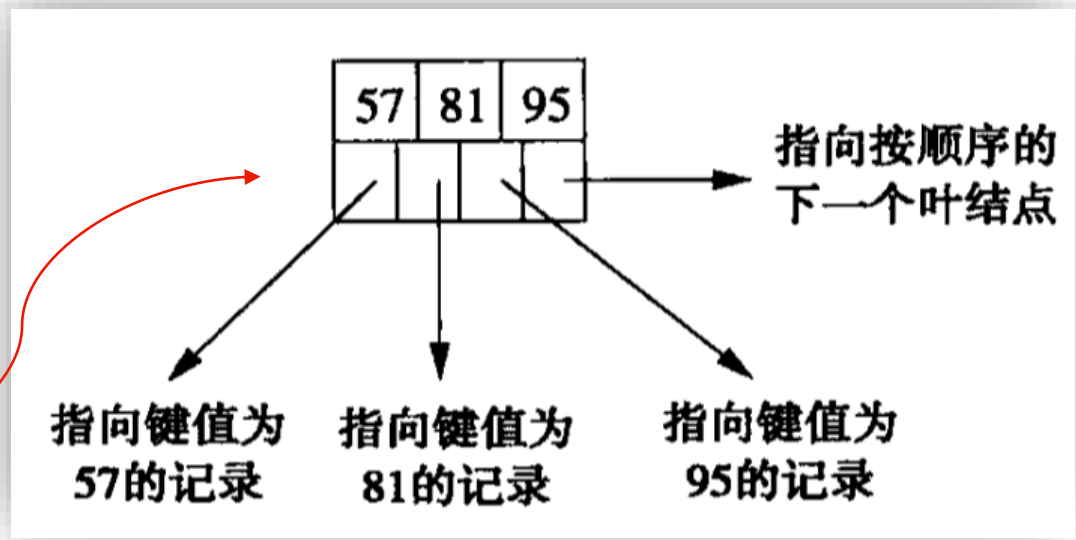
- 随着文件的增大，索引查找性能和数据顺序扫描性能都会下降

B+树

- B+树索引结构是在数据插入和删除的情况下仍能保持其执行效率的几种使用广泛的索引结构之一。B+树采取平衡树的结构，每个非叶结点有[n/2]~n 个子女，n为树的阶数。
- B+树结构会增加文件插入和删除的处理的性能开销，同时会增加空间开销。

B+树逻辑结构

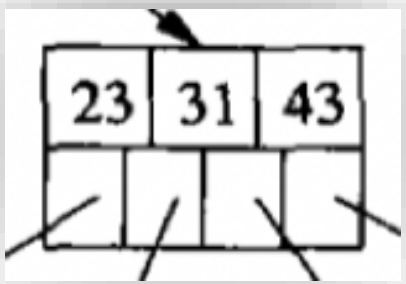

13
5
23 31 43
非叶节点
叶节点
2 3
5 11
13 17 19
23 29
31 37 41
43 47


57 81 95
指向按顺序的
下一个叶结点
指向键值为
57的记录
指向键值为
81的记录
指向键值为
95的记录

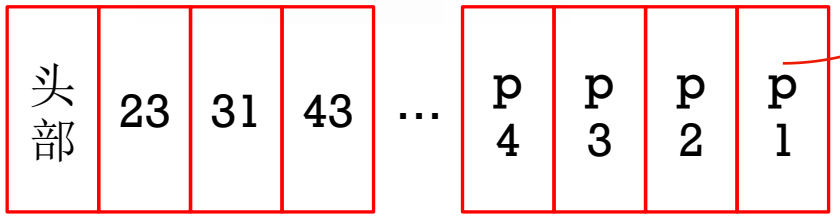
逻辑结构

B+树物理结构

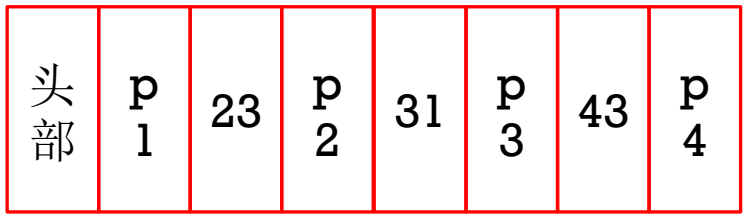
非叶节点



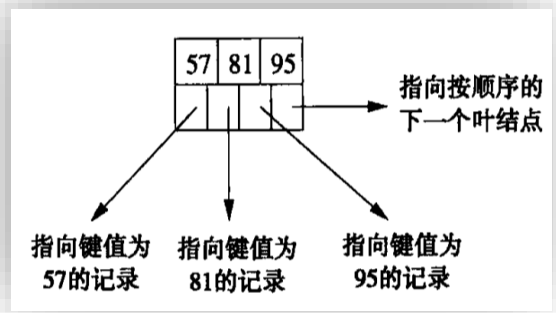
指向第一个键大于等于43索引块的物理地址



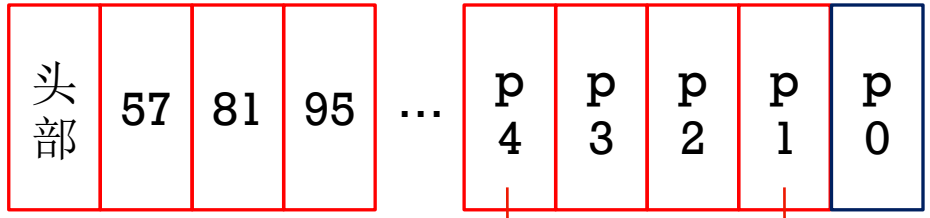
键域　　值域，指针域



叶节点



指向按顺序上一个叶节点



指向按顺序的下一个叶节点

指向值为57的记录

B+树
增删改查

查

1.**单一节点存储更多的元素**（这样该节点下分支变多了，树变矮胖了），使得查询的IO次数更少。

2.**所有查询都要查找到叶子节点**，查询性能稳定。

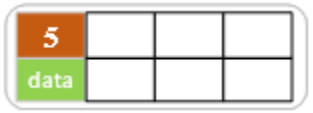
3.所有**叶子节点**形成**有序链表**，便于**范围查询**。

B+树
增删改查

- 每个非根结点key的个数最小为**n/2**，最大为**n-1**。
- 所有的**中间节点元素都同时存在于子节点**，在子节点元素中是**最大（或最小）元素**。

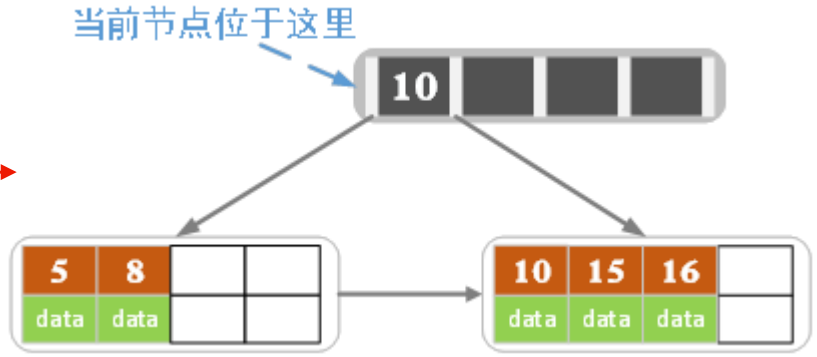
增 （5阶B+树）

空树中插入5号索引



依次插入8、10、15号索引

插入16号索引

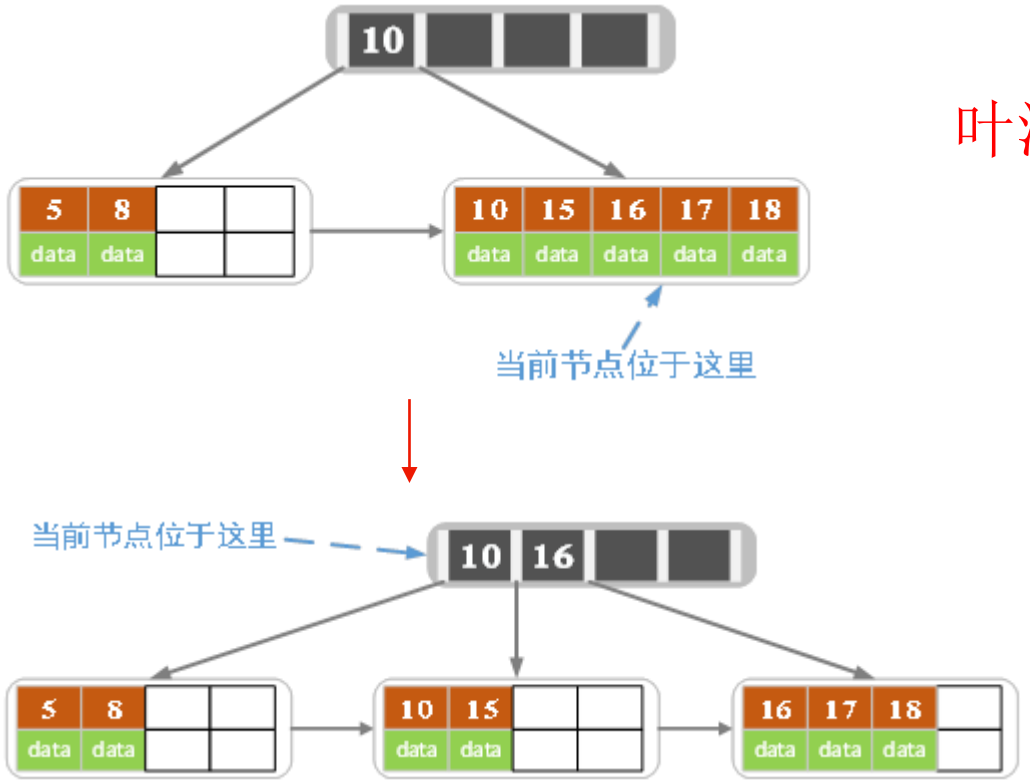


根满

B+树
增删改查

增

插入17、18号索引

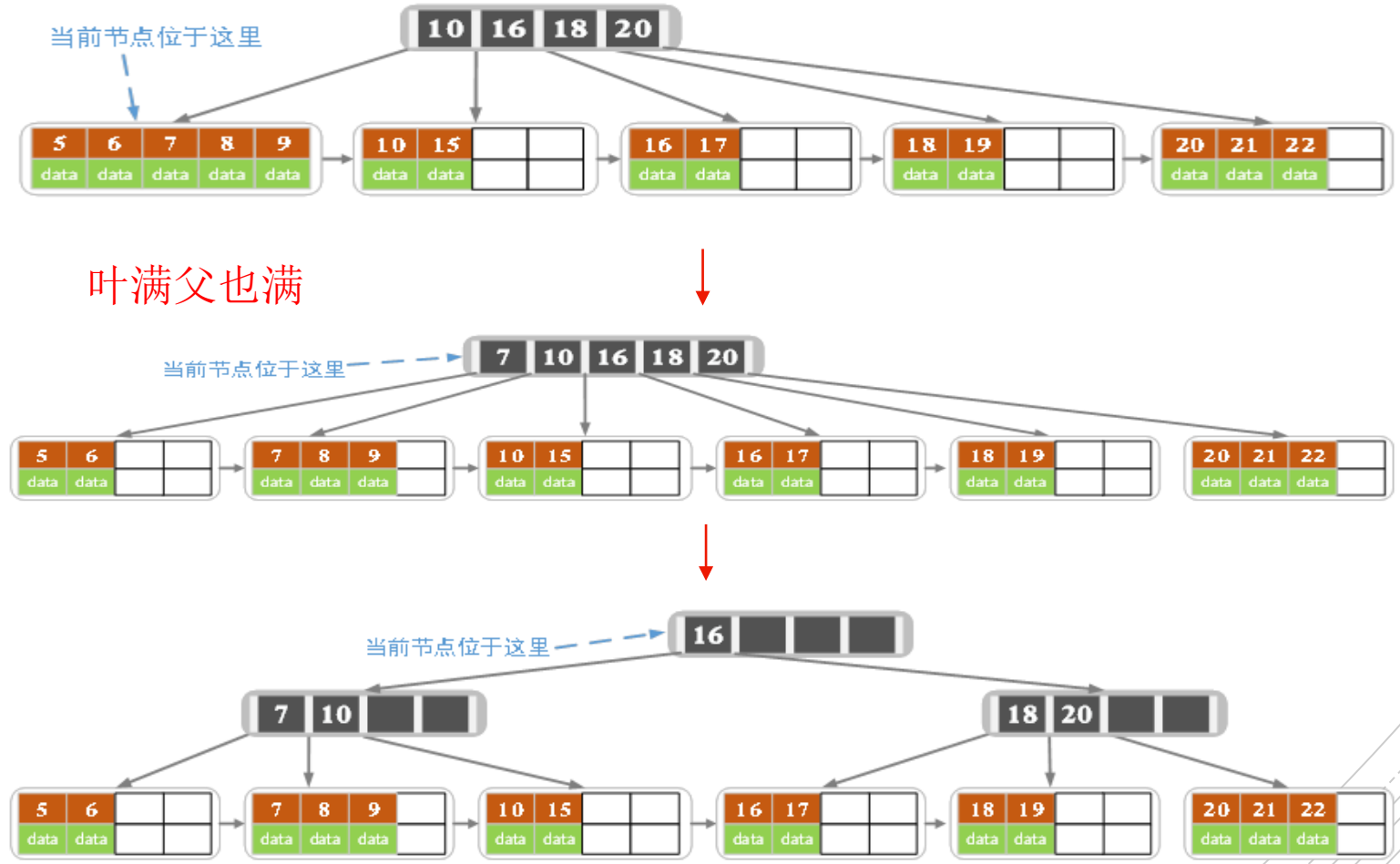


叶满父未满

B+树
增删改查

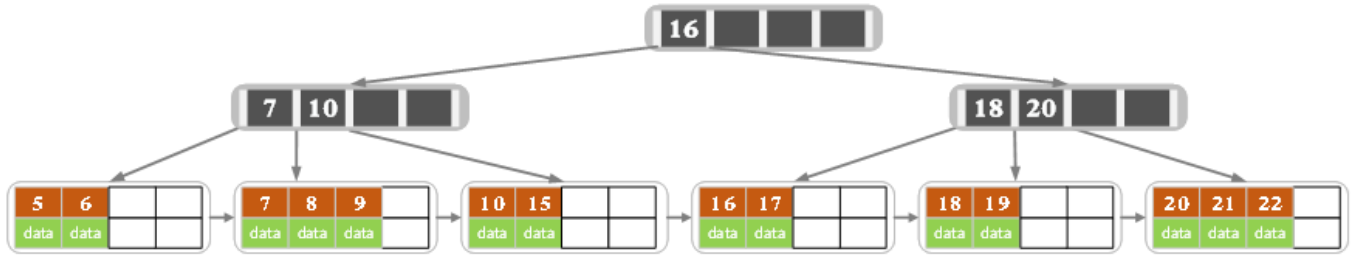
增

插入若干索引



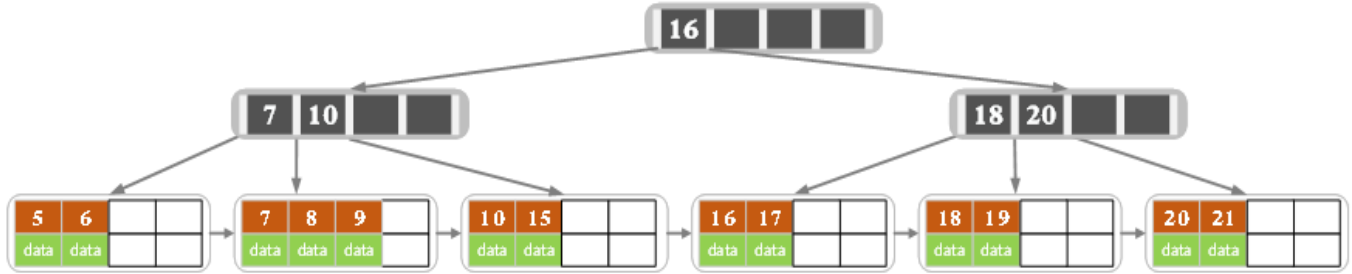
B+树
增删改查

删



最简单的删除

非叶节点无此索引，删除后仍满足B+树定义

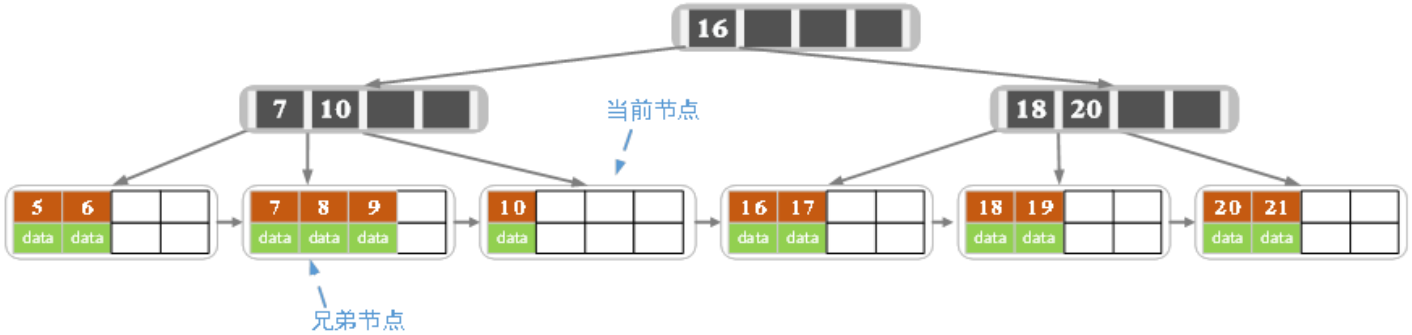


删除22号索引

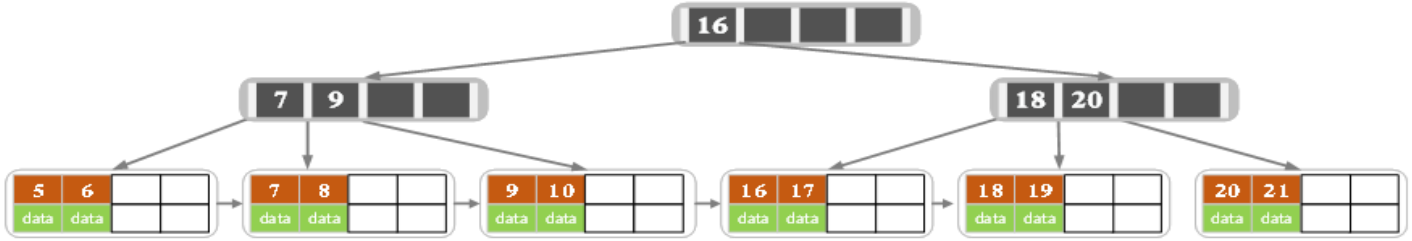
B+树
增删改查

删

删除15号索引



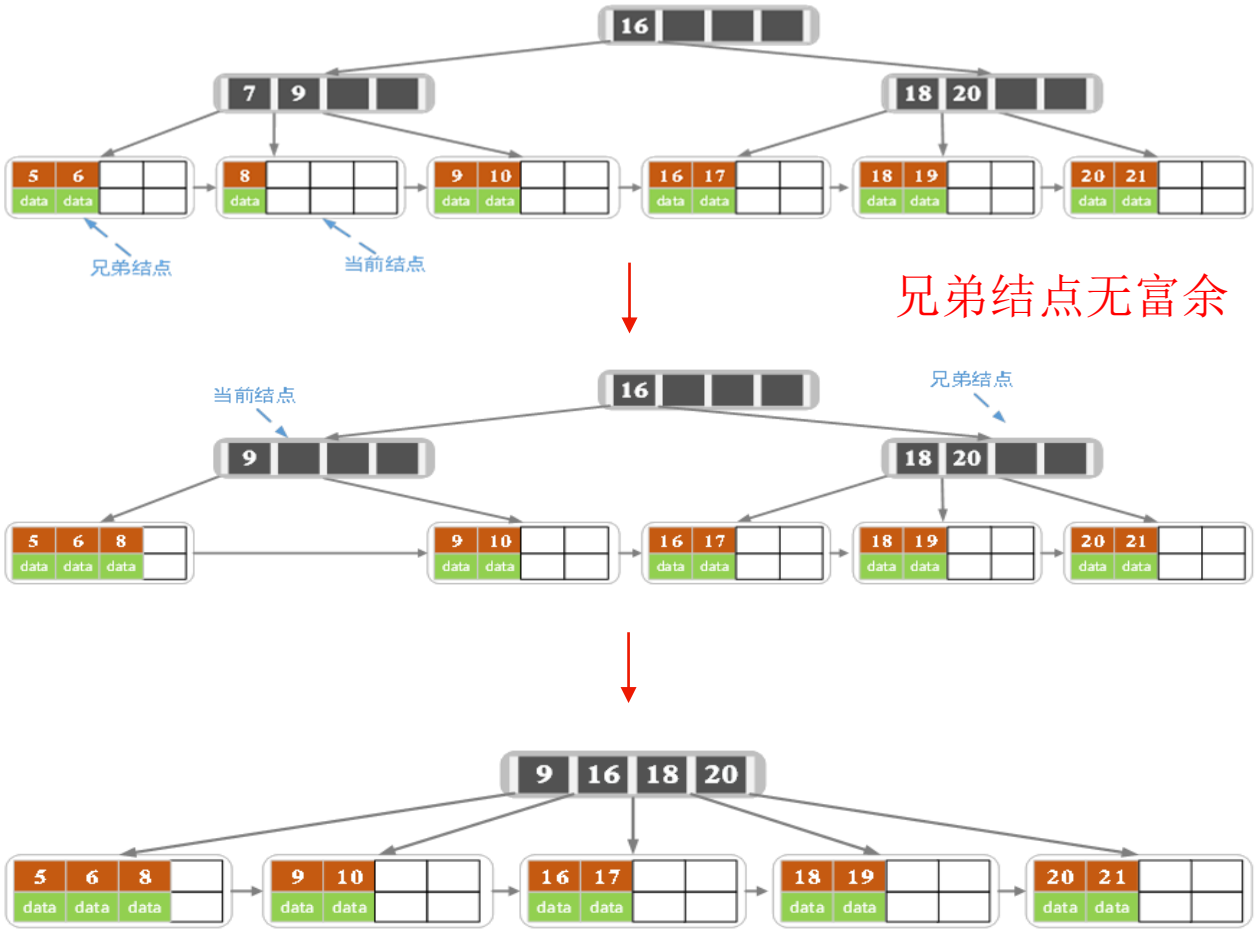
兄弟结点有富余



B+树
增删改查

删

删除7号索引



缓冲区管理

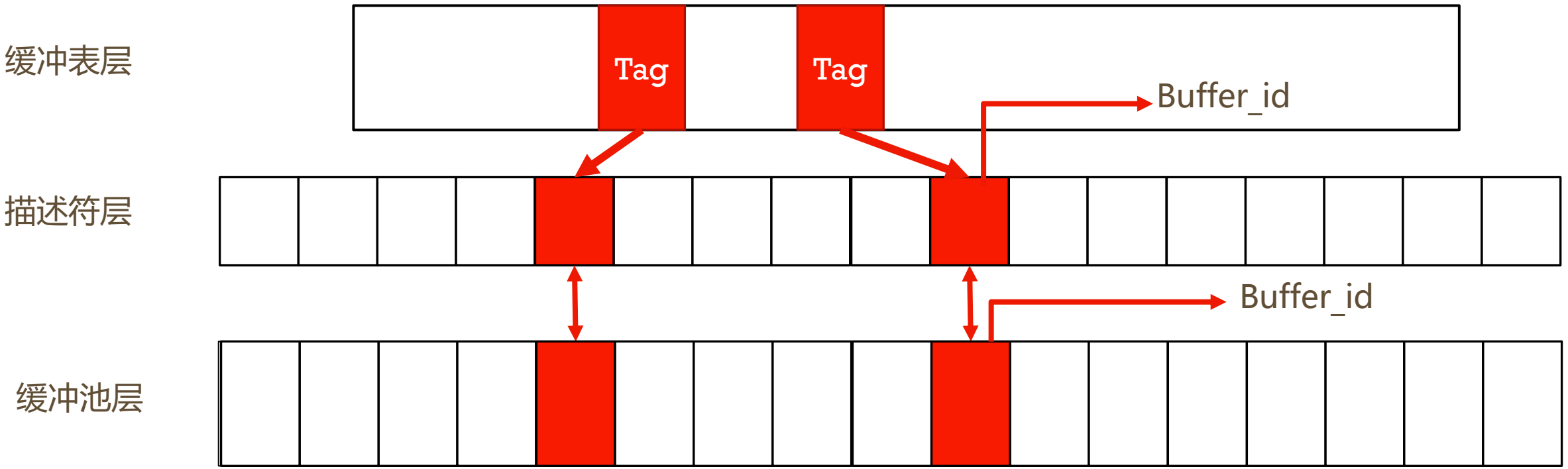
缓冲区替换策略

被钉住的块

块的强制写出

缓冲区管理器的结构

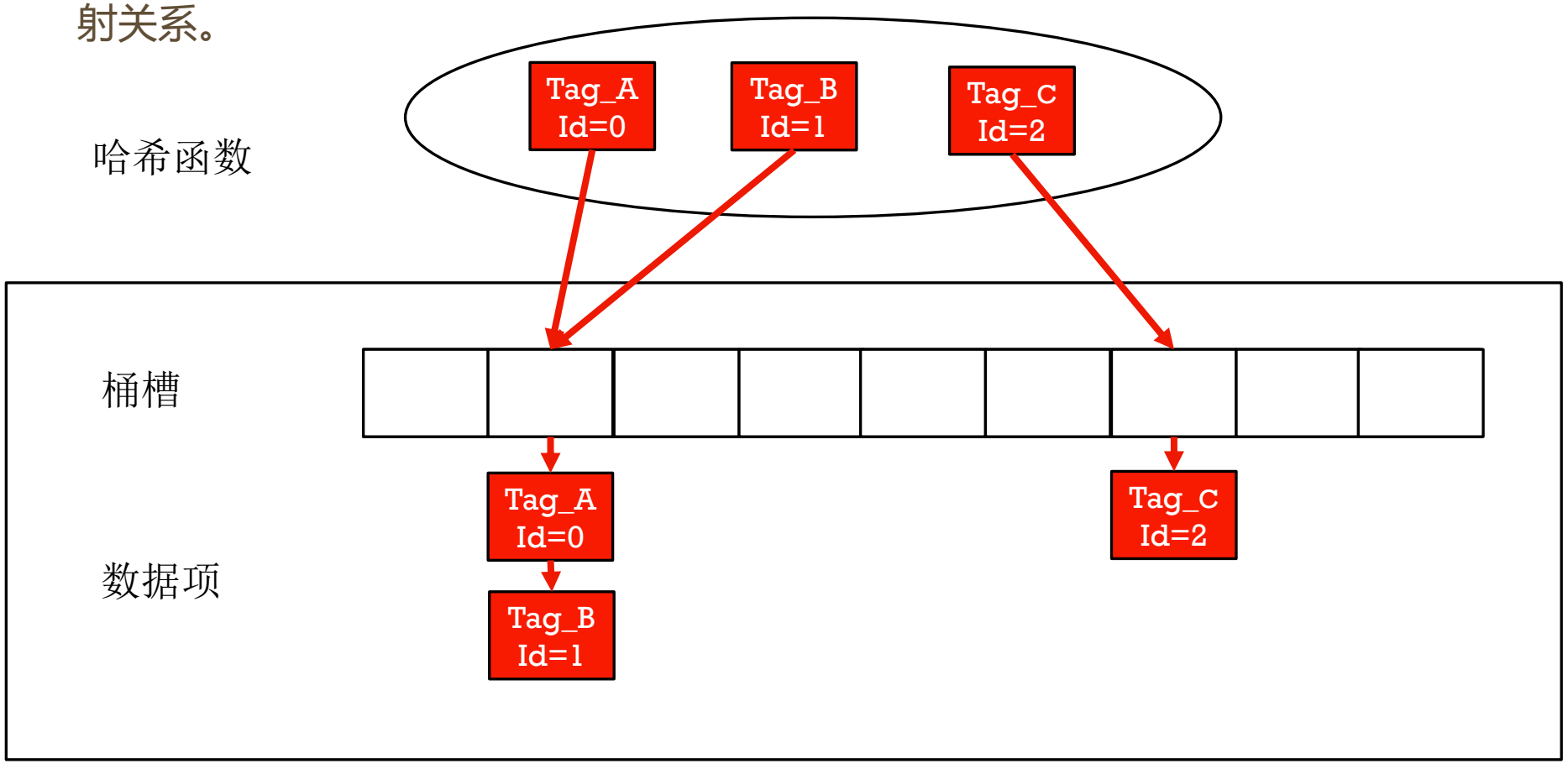
缓冲区管理器由三层组成，即缓冲表层、缓冲区描述符层和缓冲池层，



缓冲管理器结构

缓冲区管理器的结构

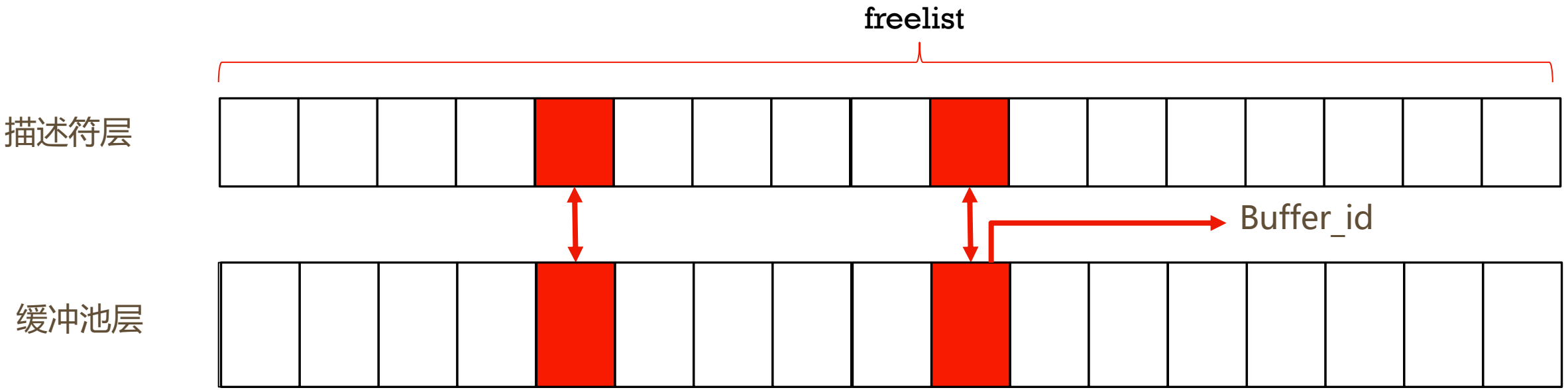
缓冲表层是一个散列表，它存储着页面的buffer_tag 与描述符的buffer_id 之间的映射关系。



缓冲区管理器的结构

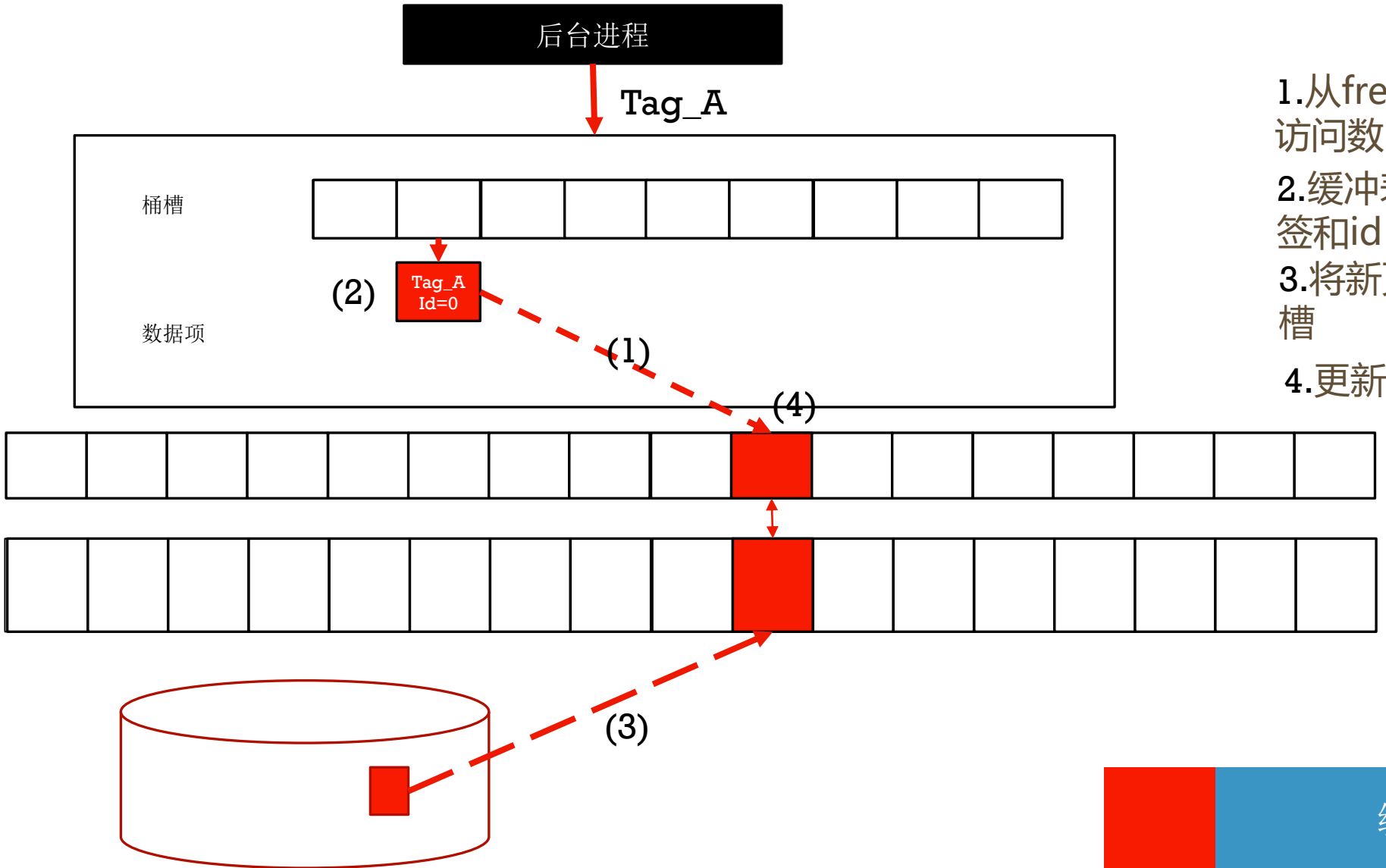
缓冲区描述符层是一个由缓冲区描述符组成的数组。每个描述符与缓冲池槽一一对应，并保存着相应槽的元数据（I/O锁，是否为脏页，页号等）。该层还用于置换算法。

缓冲池只是一个用于存储关系数据文件（例如表或索引）页面的简单数组。缓冲池数组的序号索引也就是buffer_id。



缓冲管理器结构

缓冲区管理器的结构



1.从freelist取空描述符，钉住，访问数+1

2.缓冲表插入该项，保存了该页标签和id

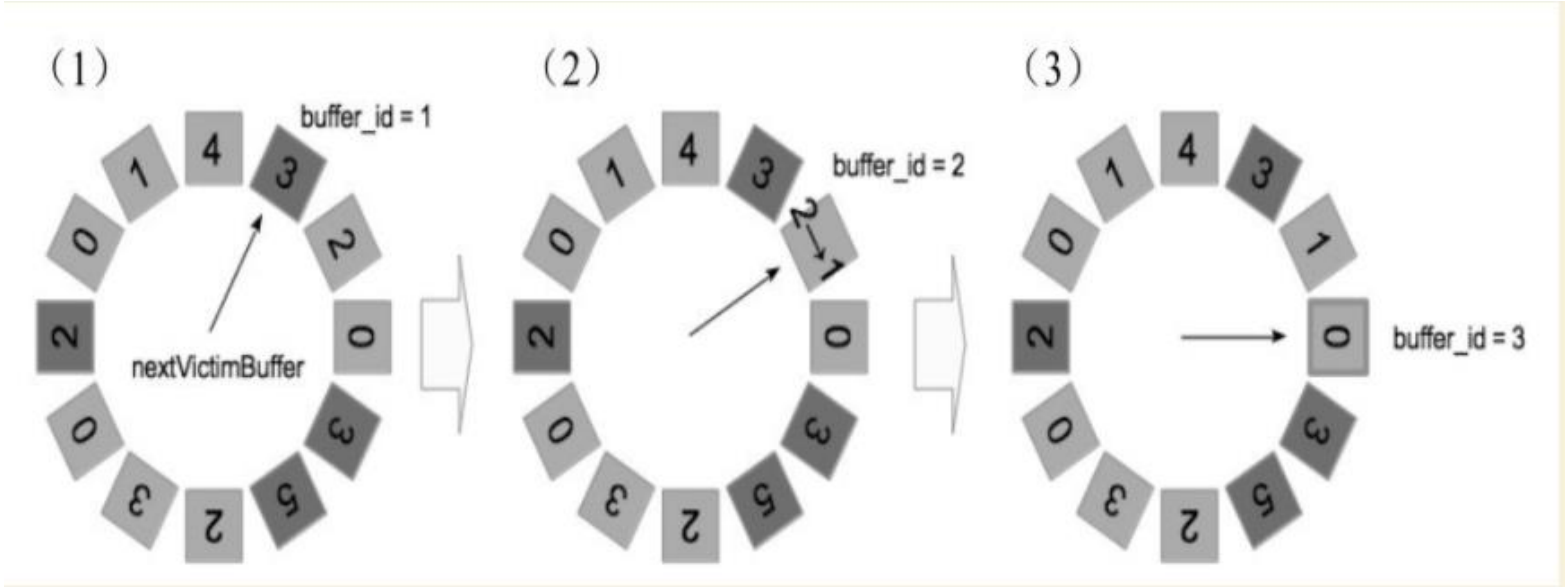
3.将新页面从存储器加载到缓冲池槽

4.更新页面描述内容

缓冲区替换策略

缓冲区替换策略

时钟扫描算法



指针指向初始页面
IF 缓冲区没有被钉住
IF pin=0 then （当前用户数）
替换该页面，结束
ELSE
pin-1
END IF
ENDIF
迭代，指向下一个页面

被钉住的块

当一个块上的更新操作正在进行时，大多数恢复系统不允许将该块写回磁盘。该特性对从崩溃中恢复十分重要。

1.在页面置换算法中要避免驱除被钉住的块

2.B-树的根防止被替换可以“钉住”

3.对于那些需要保持完整性的块，保证处理时全在内存

块的强制写出

在某些情况下，尽管不需要一个块所占用的缓冲区空间，但必须把这个块写回磁盘的写操作。

1.对缓冲区进行线性扫描以查找文件的页面
IF 页面属于传入的文件描述符
（不论这页有没有被钉住，只要是脏页了就强制写入。）
IF 是脏页
写入
slot=next

SQL解析、执行、优化

章门 刘伟

目录

语法分析

语法分析

语法分析

语法分析

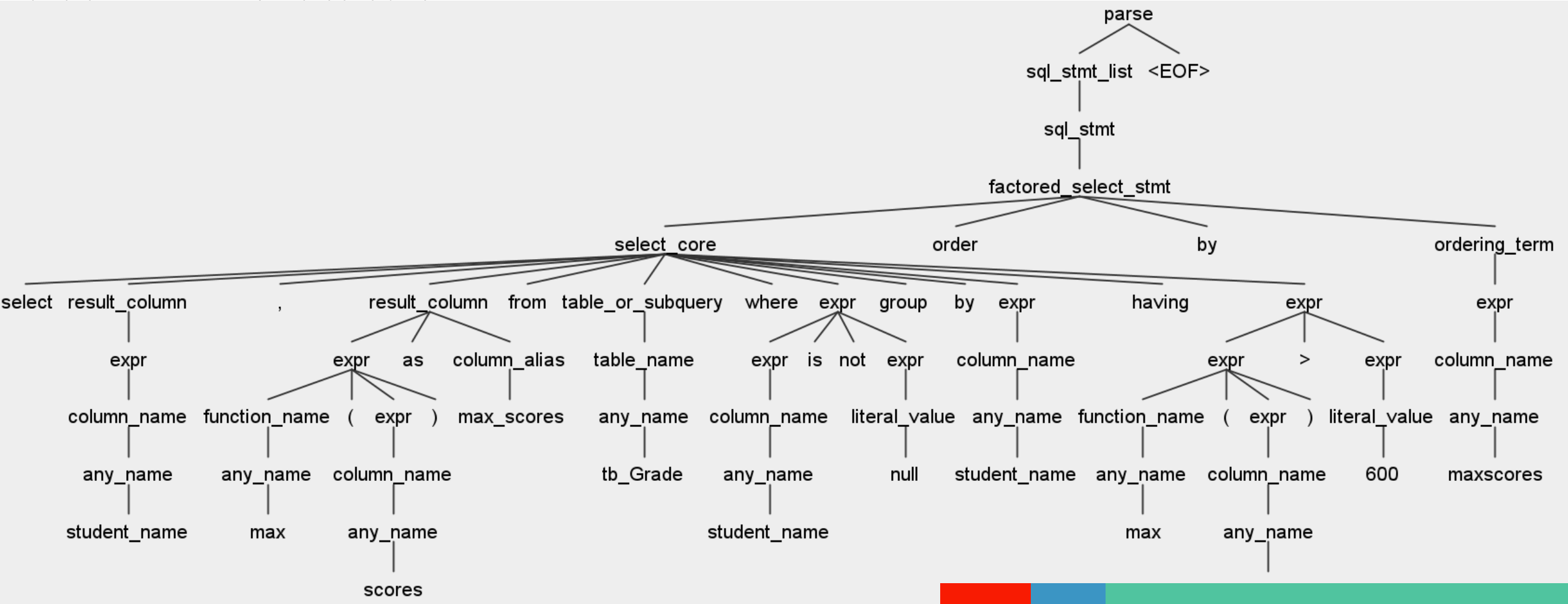
- 使用**sqlparse**库，生成语法树
- 例：

```
select student_name, max(scores) as max scores
from tb_Grade
where student_name is not null
group by student_name
having max(scores) > 600
order by maxscores
```

```
Out[10]: [<DML 'select' at 0x1D8AD74FB80>,
          <Whitespace ' ' at 0x1D8AD74FB20>,
          <IdentifierList 'studen...' at 0x1D8AD7564A0>,
          <Whitespace ' ' at 0x1D8AD755100>,
          <Newline ' ' at 0x1D8AD755160>,
          <Keyword 'from' at 0x1D8AD7551C0>,
          <Whitespace ' ' at 0x1D8AD755220>,
          <Identifier 'tb_Gra...' at 0x1D8AD756580>,
          <Newline ' ' at 0x1D8AD7552E0>,
          <Where 'where ...' at 0x1D8AD756200>,
          <Keyword 'group ...' at 0x1D8AD7556A0>,
          <Whitespace ' ' at 0x1D8AD755700>,
          <Identifier 'studen...' at 0x1D8AD7565F0>,
          <Whitespace ' ' at 0x1D8AD7557C0>,
          <Newline ' ' at 0x1D8AD755820>,
          <Keyword 'having' at 0x1D8AD755880>,
          <Whitespace ' ' at 0x1D8AD7558E0>,
          <Comparison 'max(sc...' at 0x1D8AD7566D0>,
          <Whitespace ' ' at 0x1D8AD755C40>,
          <Newline ' ' at 0x1D8AD755CA0>,
          <Keyword 'order ...' at 0x1D8AD755D00>,
          <Whitespace ' ' at 0x1D8AD755D60>,
          <Identifier 'maxsco...' at 0x1D8AD756660>]
```

语法分析例

```
select student_name, max(scores) as max scores
from tb_Grade
where student_name is not null
group by student_name
having max(scores) > 600
order by maxscores
```



逻辑查询计划

语法分析树到逻辑查询计划

逻辑查询计划的优化

从语法分析树到逻辑查询计划

DDL与DML的解析

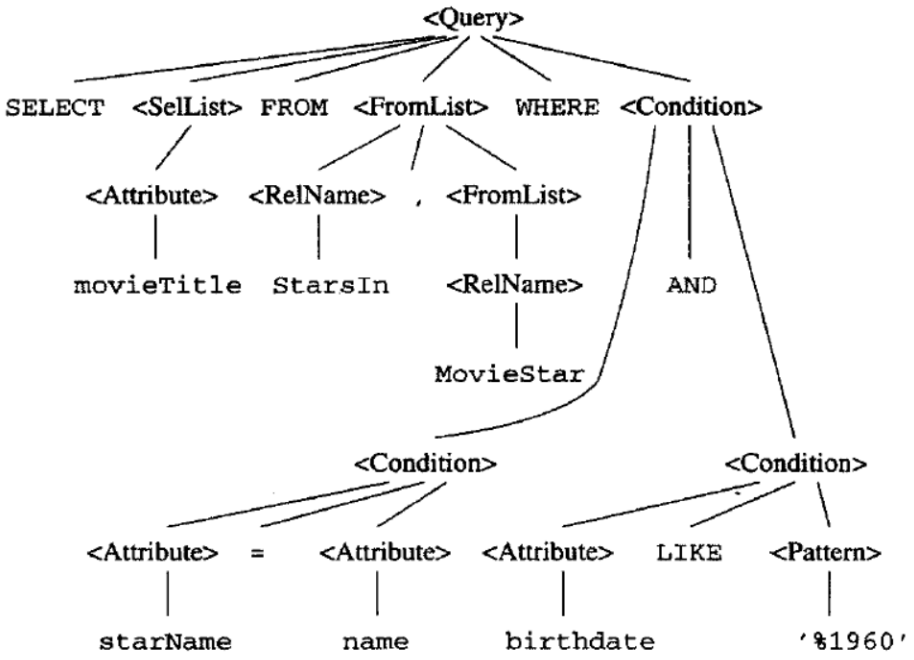
- DDL（数据库定义语言）与部分DML（数据库操作语言）的解析较为简单，如insert、update、delete、create等，不需要额外的优化与调整，直接翻译为对应的操作即可，因此不再赘述。
- 后文主要是针对查询(select)相关的操作进行解析，转化为逻辑查询计划与物理查询计划。

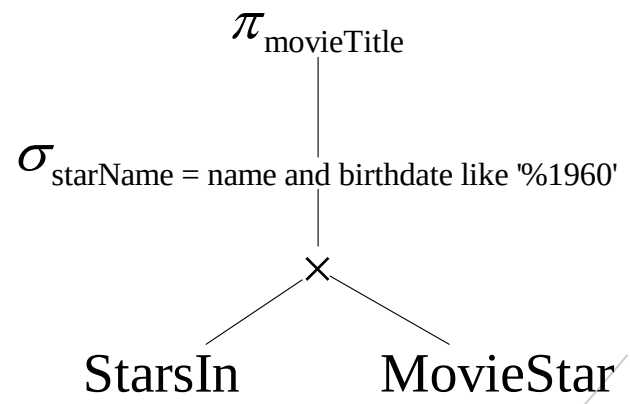
从语法分析树到逻辑查询计划

转化为关系代数（没有子查询的情况）

- 例：

```
select movieTitle
from StarsIn, MovieStar
where starName = name
      and birthdate like '%1960'
```





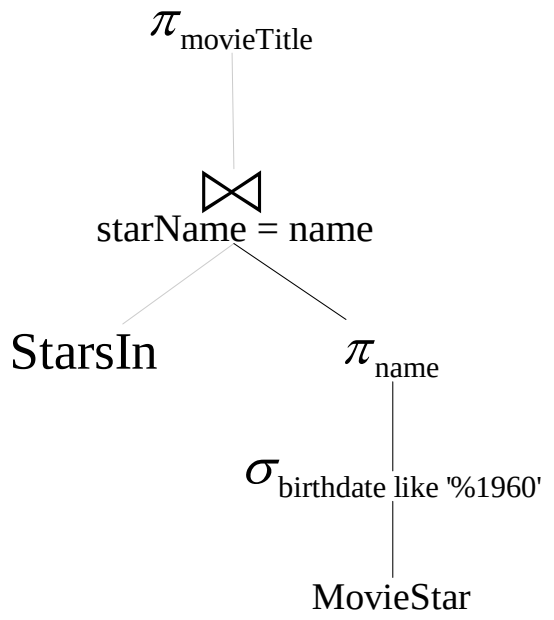
StarsIn (movieTitle, movieYear, starName)
MovieStar (name, address, gender, birthdate)

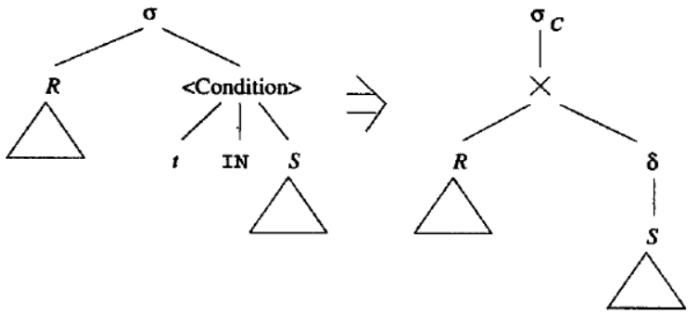
从语法分析树到逻辑查询计划

转化为关系代数（含有子查询，子查询独立）

- 例：

```
select movieTitle
from StarsIn
where starName in (
    select name
    from MovieStar
    where birthdate like '%1960'
);
```





StarsIn (movieTitle, movieYear, starName)
MovieStar (name, address, gender, birthdate)

从语法分析树到逻辑查询计划

转化为关系代数（含有子查询，子查询相关）

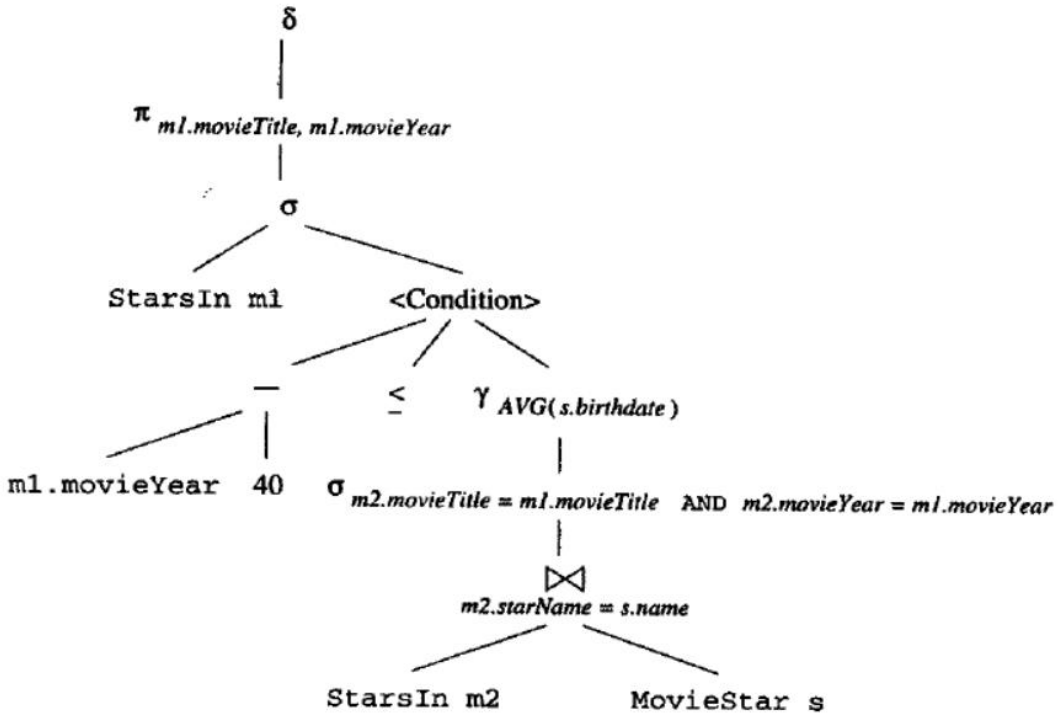
- 延迟选择
- 当子查询引用外层查询定义的元素时，在先进行子查询的情况下，需要延迟选择子查询中涉及到的条件，直到子查询与外层查询的拷贝相结合。
- 例：

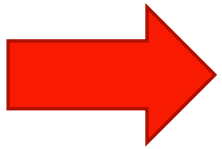
```
select distinct m1.movieTitle, m1.movieYear
from StarsIn m1
where m1.movieYear - 40 <= (
    select avg(birthdate)
    from StarsIn m2, MovieStar s
    where m2.starName = s.name and
        m1.movieTitle = m2.movieTitle and
        m1.movieYear = m2.movieYear
);
```

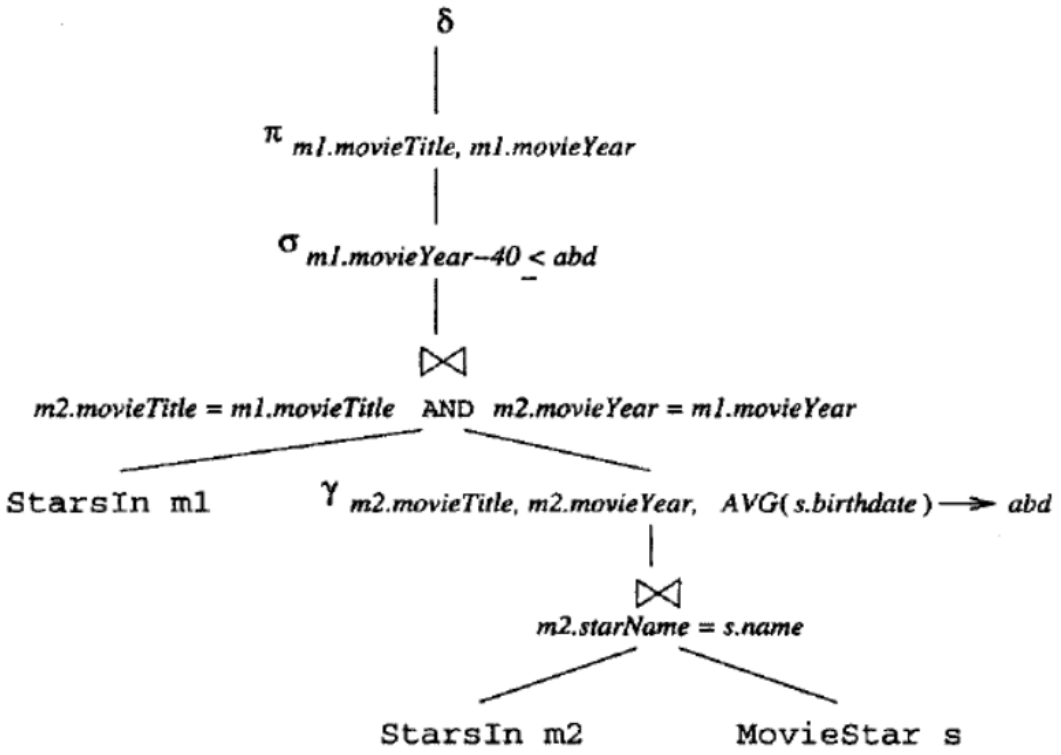
StarsIn (movieTitle, movieYear, starName)
MovieStar (name, address, gender, birthdate)

转化为关系代数（含有子查询，子查询相关）

- 延迟选择例







StarsIn (movieTitle, movieYear, starName)
MovieStar (name, address, gender, birthdate)

逻辑查询计划的优化

涉及选择的定律

- 选择运算可以明显减少关系的大小，因此在不改变表达式结果的情况下，选择应该在语法树上尽可能下移
- 查询优化的首要策略

$$\sigma_{C_1 \text{ and } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$$

$$\sigma_{C_1 \text{ or } C_2}(R) = (\sigma_{C_1}(R)) \cup (\sigma_{C_1}(R))$$

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$

$$\sigma_C(R - S) = \sigma_C(R) - S$$

逻辑查询计划的优化

涉及投影的定律

- 可以在表达式树上的任何地方引入投影，只要它所消除的属性是其上的运算符从来不会用到的，并且不在整个表达式的结果之中。

逻辑查询计划的优化

有关连接与积的定律

- 计算连接的算法通常比直接进行笛卡尔积再选择快

$$R \bowtie_C S = \sigma_C(R \times S)$$

物理查询计划

运算代价估计

基于代价的选择

物理查询计划

投影大小

a（4）	b（4）	c（100）

T(R)=10000
B(R)=1250

- $B(R)$是存储关系 R 所有元组所需的块数。
- $T(R)$是关系 R 的元组数。
- $V(R, a)$是关系 R 的属性 a 的值计数，即关系 R 中属性 a 上所具有的不同值的数目。并且，$V(R, [a_1, a_2, \cdots, a_n])$表示关系 R 中属性 $a_1, \cdots, a_n$ 作为一起考虑时所出现的不同值的数目，即 $\pi_{a_1,a_2,\cdots,a_n}(R)$ 中的不同元组数。

$$U = \pi_{a,b}(R)$$

a（4）	b（4）

T(U)=10000
B(U)=200

物理查询计划

选择

$S=\sigma_{A=c}(R)$	• $T(S)=T(R)/V(R,A)$
$S=\sigma_{a<c}(R)$	• $T(S)=T(R)/3$
and连接的多条件	多个简单条件的级联
$S=\sigma_{C1\ \mathrm{OR}\ C2}(R)$	$n(1-(1-m_1/n)(1-m_2/n)$
条件矛盾	T=0

物理查询计划

连接

$R(X, Y) \bowtie S(Y, Z)$

- $T(R \bowtie S) = T(R)T(S)/\max(V(R, Y), V(S, Y))$

多属性连接

$R(a,b,c) \bowtie_{R.b=S.d \text{ AND } R.c=S.e} S(d,e,f)$

级联运算

多关系连接

$S = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

逐步运算

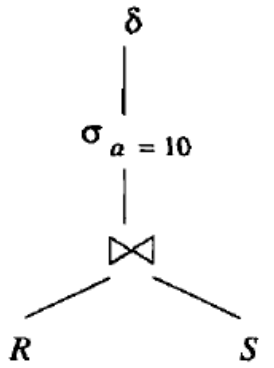
物理查询计划

并	较大值加上较小值的一半
交	较小值的一半
差	$T(R) - T(S)/2$。
消除重复	$T(R)/2$
分组与聚集	$T(R)/2$

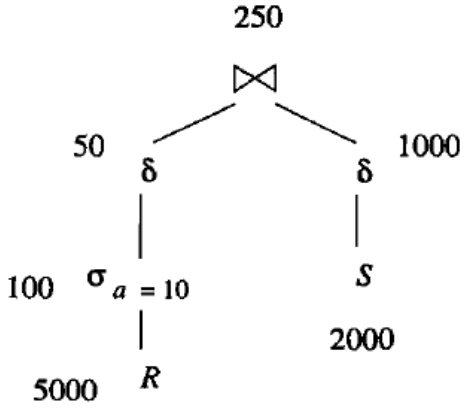
物理查询生成

基于代价的选择（启发式估计）

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

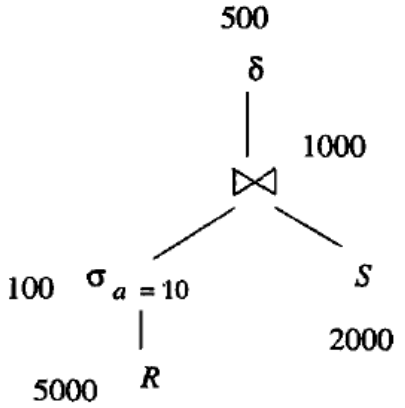


T、S、V的值需要统计得到，因此我们可以选取一部分记录作为整体的估计。



a)

100 + 50 + 1000 = 1150



b)

100 + 1000 = 1100

物理查询计划

计划生成

```
SELECT name, quantity, pages
FROM customer, orders, book
WHERE customer.id = orders.customer_id
      AND book.id = orders.book_id
      AND pages > 100
      AND quantity > 5;
```

```
9: PROJECTION name quantity pages
└─────8: INDEXED JOIN 4 and 7
    ├─────4: INDEXED JOIN 3 and 2
    │    ├─────3: SEARCH orders.quantity > 5
    │    └─────2: PROJECTION id name
    │         └─────1: SEARCH customer.id = orders.customer_id
    └─────7: PROJECTION id pages
         └─────6: SELECTION book.pages > 100
              └─────5: SEARCH book.id = orders.book_id
```

数据库事务管理系统实现

李世龙 李畅

事务管理

锁模式

加锁和释放锁的流程

死锁的检测预处理

什么是事务

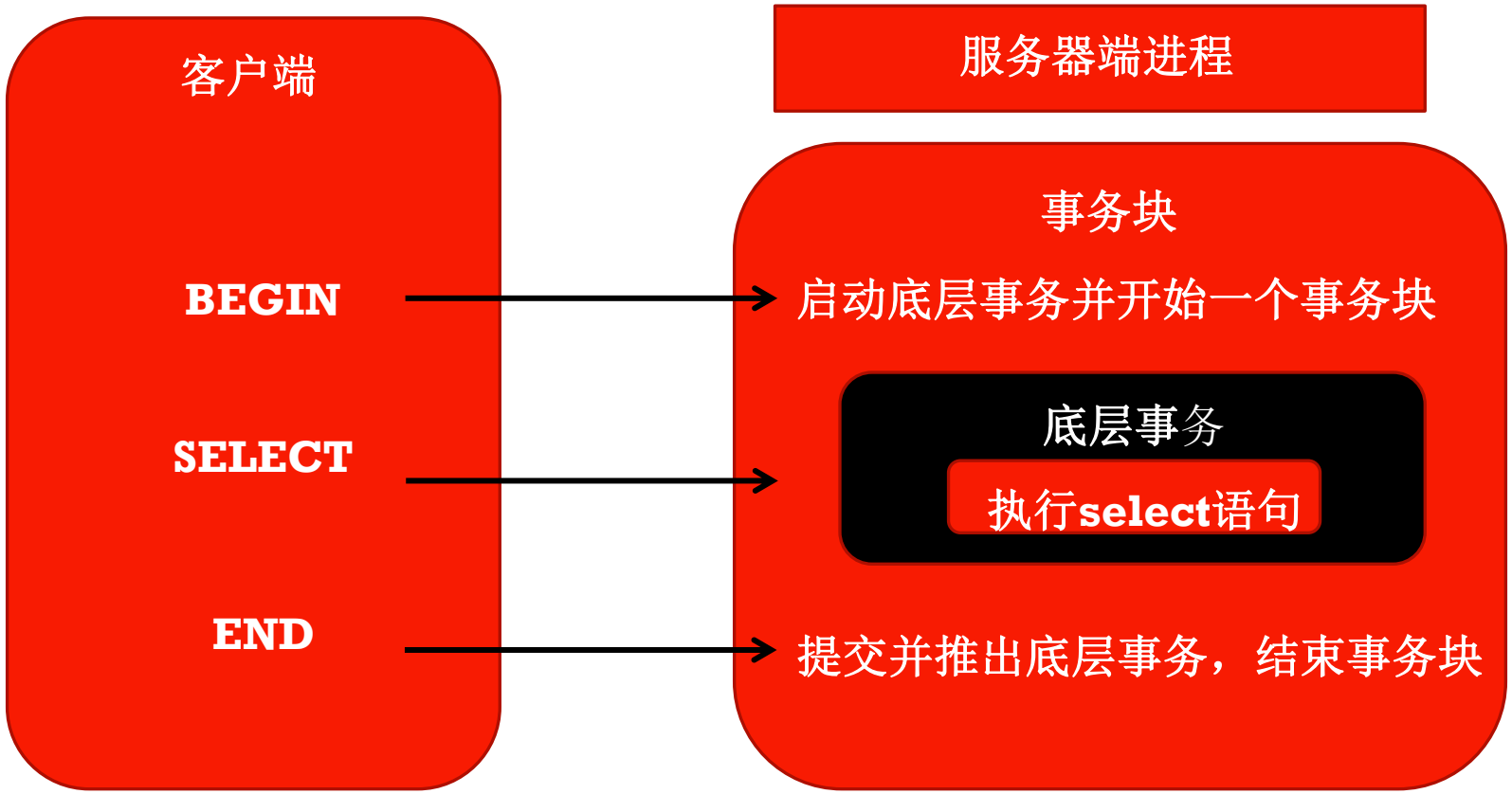
```
begin transaction T1
update student
set name='Tank'
where id=2006010
delete from student
where id=2006011
commit
```

在关系数据库中,一个事务可以是一条**SQL**语句、一组**SQL**语句或整个程序。

开始事务：**BEGIN TRANSACTION**（事务）
提交事务：**COMMIT TRANSACTION**（事务）
回滚事务：**ROLLBACK TRANSACTION**（事务）

事务中包含的操作被看做一个逻辑单元，这个逻辑单元中的操作要么全部成功，要么全部失败

这里说的事务块是指一连串不能中断的命令集合.
事务指的是单一一条SQL语句
一个进程最多只有一个事务块



事务的基本操作

事务管理

在语句层面操控事务

1.开始事务函数 StartCommand

在每一条SQL语句前执行，判断事务的当前状态，并改变。

2.提交事务函数 CommitCommand

在每一条SQL语句之后执行，判断事务的当前状态，并改变。

3.事务中断处理函数 AbortCommand

在事务出错时执行该函数，改变事物的状态

在事务快层面操纵事务

1.进入事务块 BeginBlock

在读取BEGIN命令执行该函数会改变事务状态。

2.退出事务块 EndBlock

在读取COMMIT命令或ROLLBACK命令后执行该函数改变命令状态。

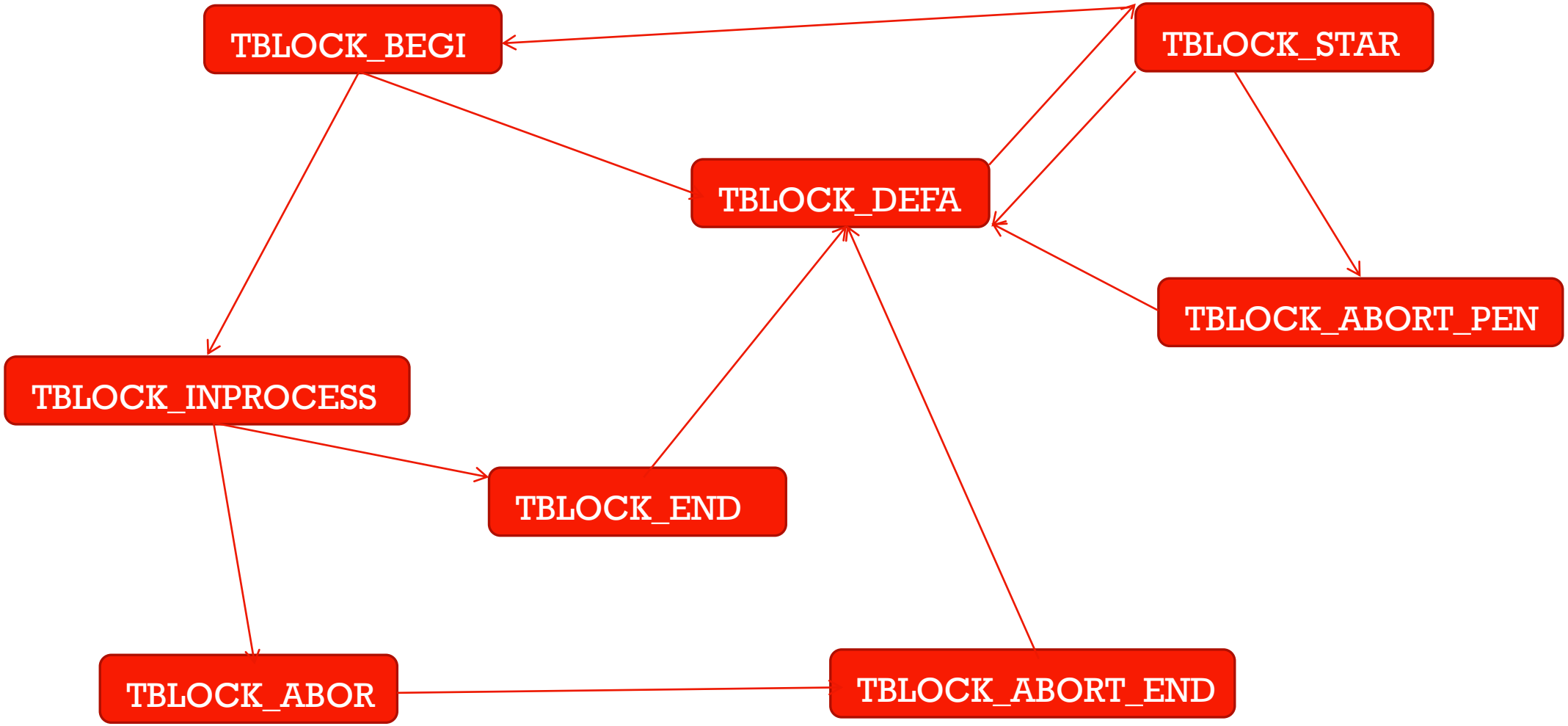
3.回滚事务快 RollbackBlock

在执行ROLLBACK命令后执行该函数改变命令状态。

以上这些函数都不执行具体的操作，只改变事务块的状态。事务块的本质是一个状态机。下面介绍事务状态的设计。

事务块的状态

```
typedef enum TBlockState{
    TBLOCK_DEFAULT,          //事务的起始状态
    TBLOCK_STARTED,          //执行StartCommand之后的状态
    TBLOCK_BEGIN,        //执行BEGIN命令后的状态
    TBLOCK_INPROCESS,        //事务块正在处理中
    TBLOCK_END,              //表明遇到事务结束命令
COMMIT的状态
    TBLOCK_ABORT,        //事务出错等待ROLLBACK
    TBLOCK_ABORT_END     //事务出错已接收ROLLBACK
    TBLOCK_ABORT_PENDING//事务处理中接收到ROLLBACK
}
```


TBLOCK_BEGI
TBLOCK_STAR
TBLOCK_DEFA
TBLOCK_ABORT_PEN
TBLOCK_INPROCESS
TBLOCK_END
TBLOCK_ABOR
TBLOCK_ABORT_END

事务管理

底层事务的基本操作

具体的事务由底层事务进行操作

1.StratTransaction 开始事务，完成事务的初始化操作

对表数据的操作交由执行器来完成

2.CommitTransaction 结束事务，并完成资源的回收

3.AbortTransaction 事务出错时执行，开中断释放资源

锁管理

锁模式

加锁和释放锁的流程

死锁的检测预处理

锁模式

锁模式有八种，按照排他性从低到高分别是
1.访问共享锁（AcessShareLock） 执行select时加到要执行的数据库对象上。
2.行共享锁（RowShareLock）执行select for update时加到数据库对象上。
3.行排他锁（RowExclusiveLock）对表进行增删改时加到表上
4.共享更新排他锁（ShareUpdateExclusiveLock）
5.共享锁（ShareLock）
6.共享行排他锁（ShareRowExclusiveLock）
7.排他锁（ExclusiveLock）
8.访问排他锁（AccessExclusiveLock）

所有的锁都有两个锁方式
1.Access锁定整个表
2.Rows锁定单独元组

表 7-5　锁冲突表

编号	名称	用途	冲突关系
1	AccessShareLock	SELECT	8
2	RowShareLock	SELECT FOR UPDATE/FOR SHARE	7 | 8
3	RowExclusiveLock	INSERT/UPDATE/DELETE	5 | 6 | 7 | 8
4	ShareUpdateExclusiveLock	VACUUM	4 | 5 | 6 | 7 | 8
5	ShareLock	CREATE INDEX	3 | 4 | 6 | 7 | 8
6	ShareRowExclusiveLock	ROW SELECT…FOR UPDATE	3 | 4 | 5 | 6 | 7 | 8
7	ExclusiveLock	BLOCK ROW SHARE/SELECT…FOR UPDATE	2 | 3 | 4 | 5 | 6 | 7 | 8
8	AccessExclusiveLock	DROP CLASS/VACUUM FULL	1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

加锁

1.先在本地锁表中通过加锁对象和加锁类型查询是否已拥有该类型的锁

1.a如果已有该类型的锁则在本地锁表的共享计数+1后退出。

1.b如果本地锁表没有，则在本地表中插入一条锁信息。

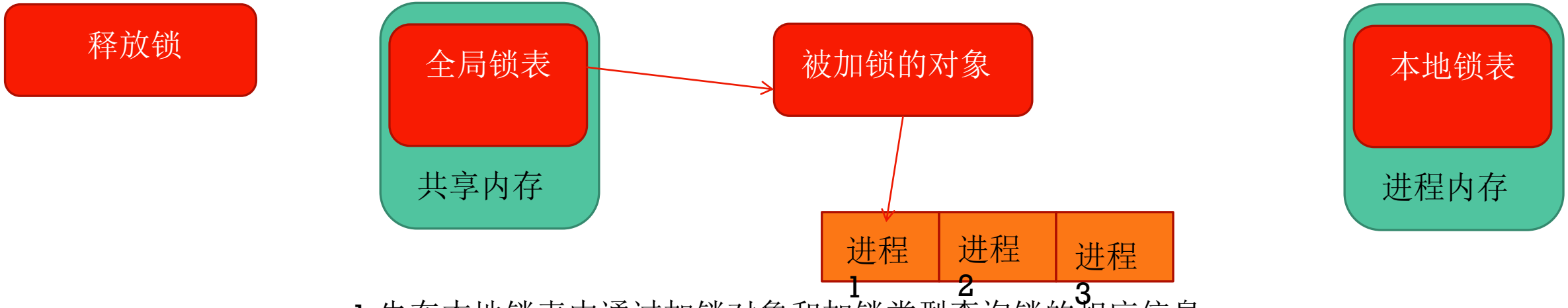
2.在全局锁表通过持有锁的对象、被加锁对象和加锁类型查询是否已拥有该类型的锁

2.a如果没找到则在全局锁表中加入一条锁信息。

3.根据之前的锁冲突表判断新加的锁与被加锁对象已持有的锁对象之间是否有冲突。

3.a如果有冲突则可以选择等待或退出，退出时要将全局锁表和本地锁表的相应信息删除。

锁管理



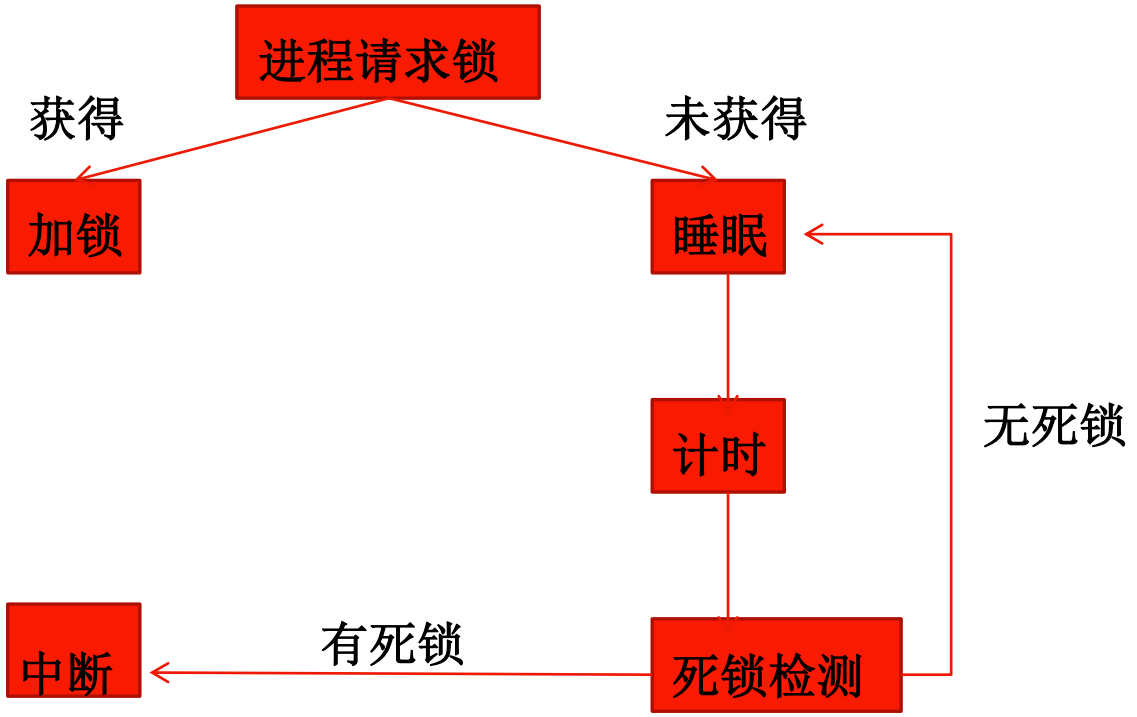
1.先在本地锁表中通过加锁对象和加锁类型查询锁的相应信息。

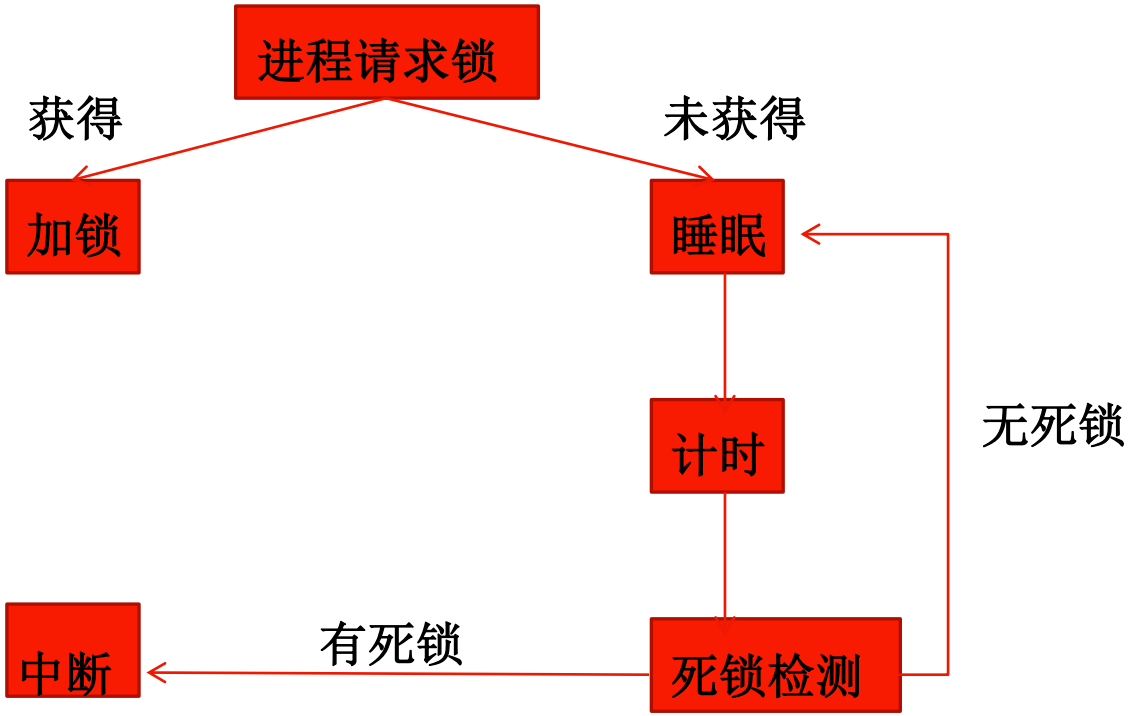
2.将该锁的共享计数减一。

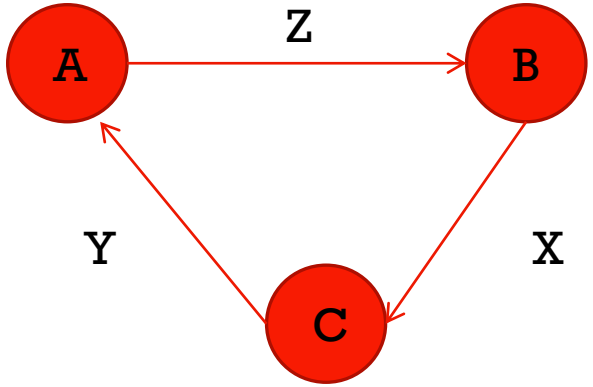
3.如果锁的共享计数大于一，则退出。

4.否则从本地锁表中删除该项锁信息，同时在全局锁表中查找并删除该项信息，同时从被加锁的对象中按顺序唤醒能唤醒的进程

死锁检测触发的条件







1.事务B加锁了对象Z
2.事务C加锁了对象X
3.事务A加锁了对象Y
4.事务B要求加锁对象X
5.事务C要求加锁对象Y
6.事务A要求加锁对象Z

申请每一个锁时都进行死锁检测。1、2、3、4、5都获得了锁或进入等待队列
当A申请Z的锁时发现图中形成了环，于是将A持有的锁释放，并将事务A整个重启，隔一段时间再重新运行。

日志管理

Undo 日志

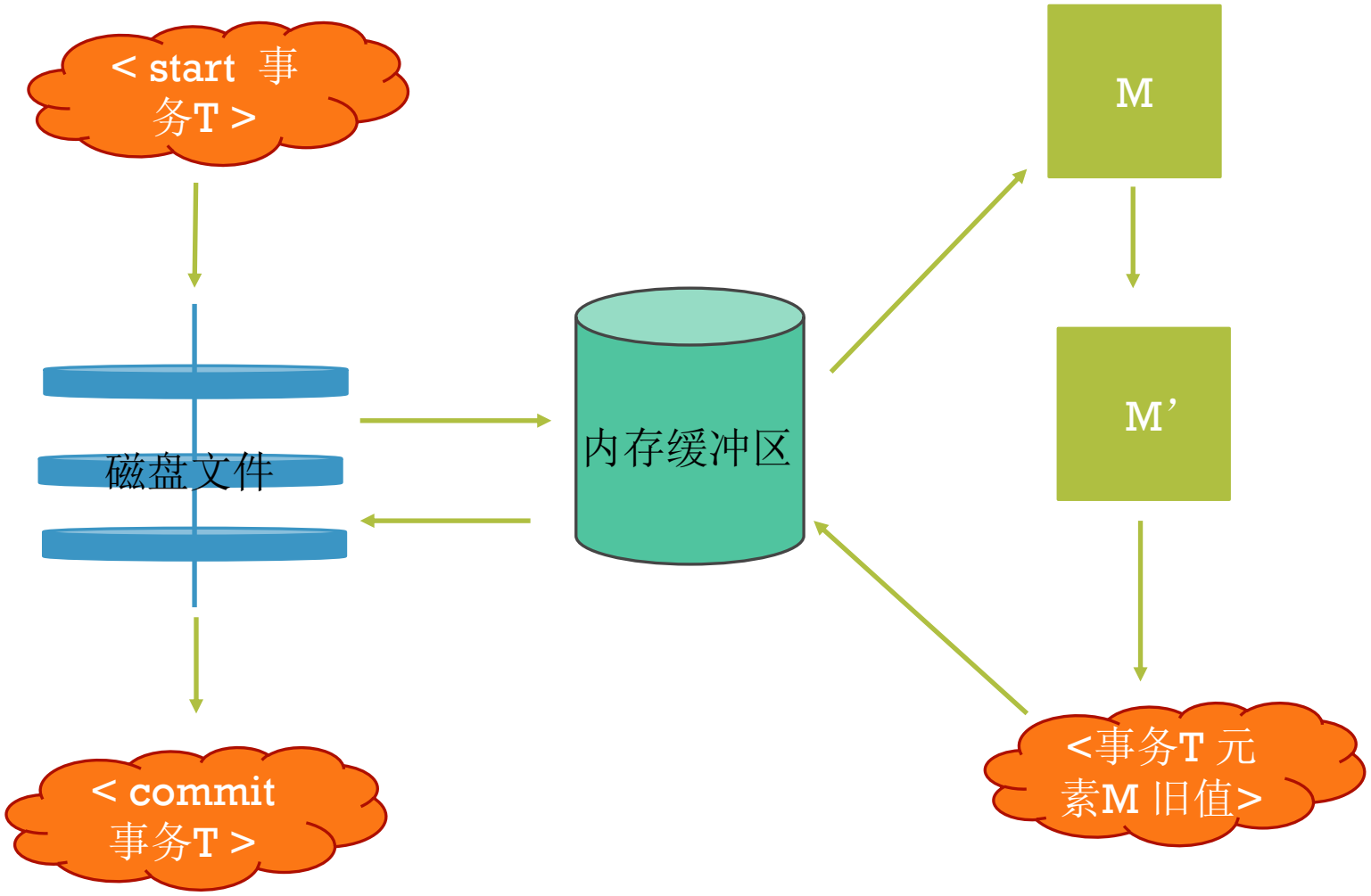
事务改变数据库内容：

必须在改变写入磁盘之前将改变的内容加入日志中并写入日志磁盘中

事务提交：

commit日志记录必须在事务写入磁盘之后写到磁盘中

Update Account set balance = 1000 where bankName = ‘A’



将<start 事务T>
写入日志磁盘文件中

读取数据库元素M，
修改M

将<事务T 元素M 旧值>
写入日志磁盘文件

将修改后的元素M
写入磁盘中

将<commit 事务T>
写入日志磁盘文件

日志管理

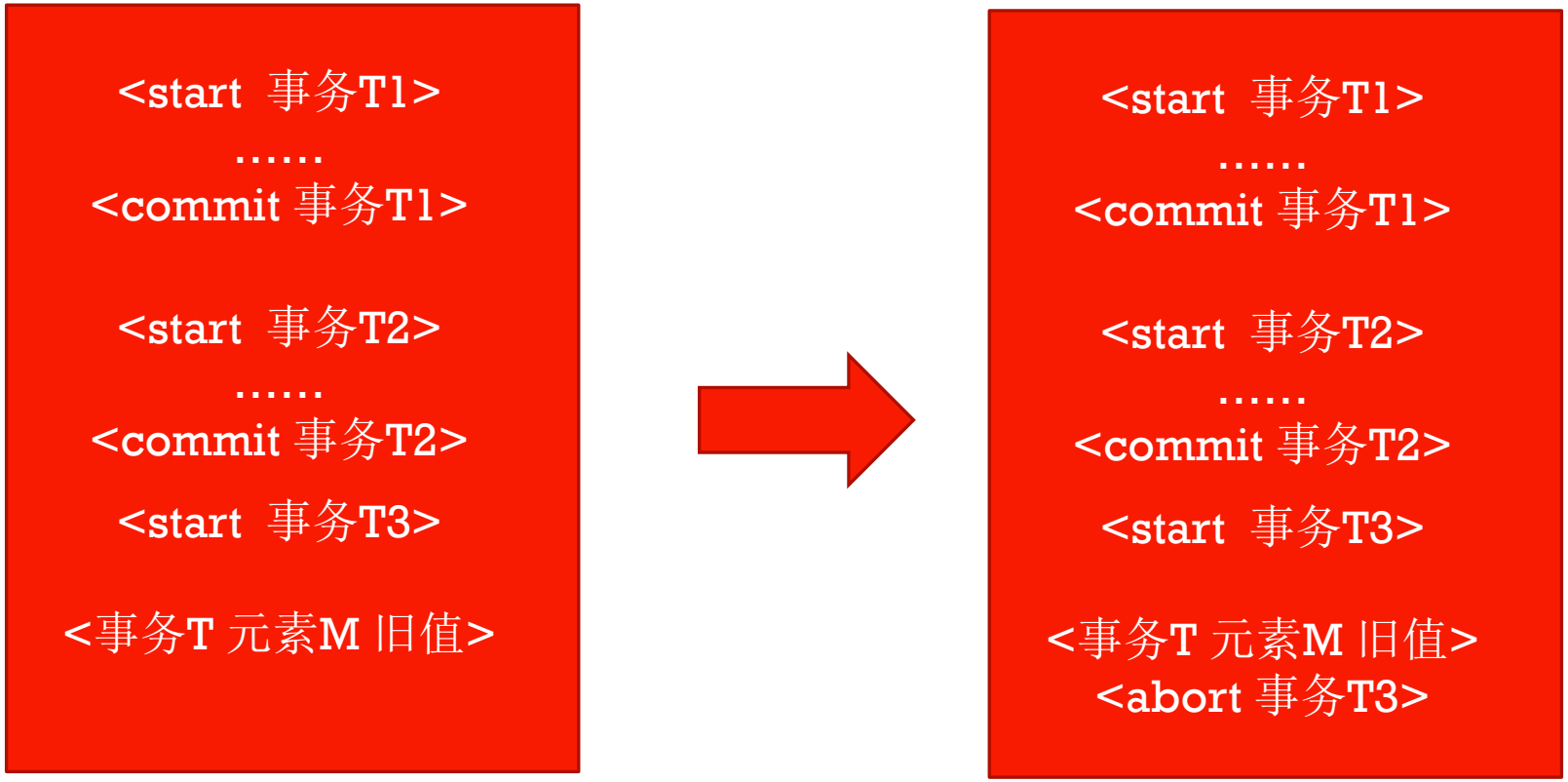
Undo 日志的恢复

从尾部扫描日志

未完成的事务

根据旧值修改

将<abort 事务T>
写入日志



Undo 日志检查点

停止接受新的事务

等待当前进行的事务commit 或者 abort

写入日志 <ckpt>

重新开始接受事务

<start 事务T1>
……
<commit 事务T1>

<start 事务T2>
……
<commit 事务T2>
<ckpt>

<start 事务T3>

<事务T 元素M 旧值>

Redo 日志

在修改磁盘上的元素之前:

保证commit 事务或者abort事务出现在日志文件中

将<start 事务T>
写入日志磁盘文件中

读取数据库元素M，
修改M

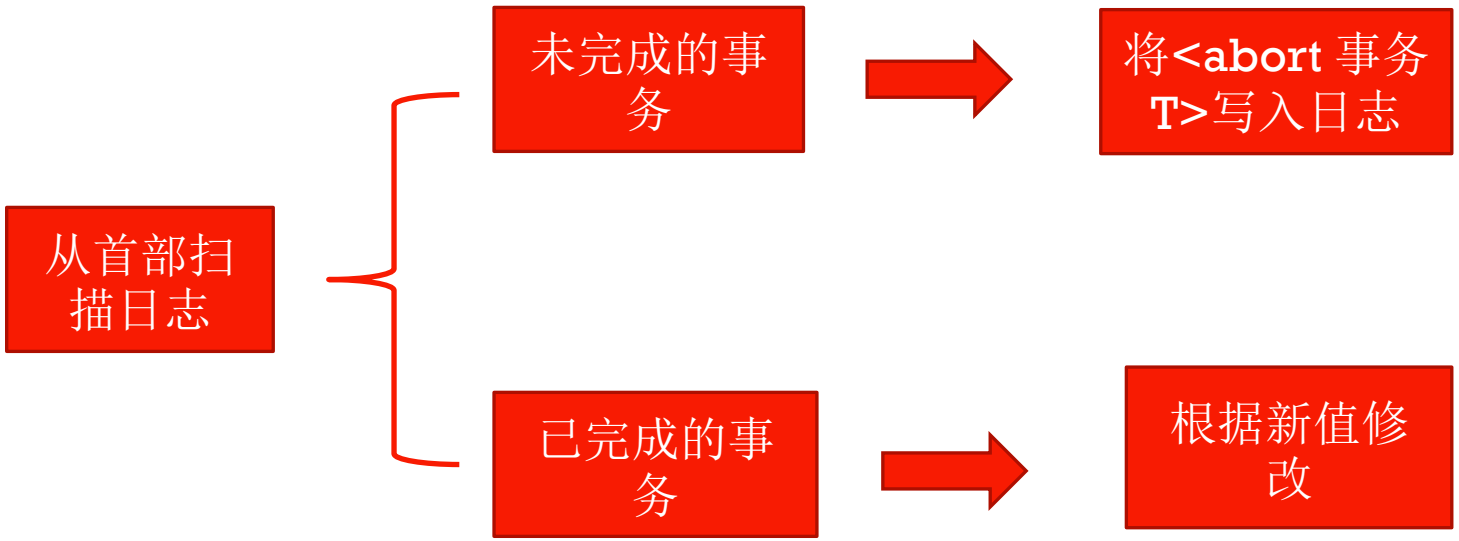
将<事务T 元素M 旧值>
写入日志磁盘文件

将<commit 事务T>
写入日志磁盘文件

将修改后的元素M
写入磁盘中

日志管理

Redo 日志



<start 事务T1>
……
<commit 事务T1>

<start 事务T2>
……
<commit 事务T2>
<ckpt>

<start 事务T3>

<事务T 元素M 新值>

Undo/Redo 日志

Undo 日志：

事务结束后立即写到磁盘中 增大磁盘IO数

Redo 日志：

事务提交前需要保存在缓冲区中 增加缓冲区数

Undo/Redo 日志:

在修改磁盘上的元素之前:保证更新数据内容出现在日志文件中

Undo/Redo 日志

恢复：

从前往后扫描 重做已提交的事务

从后往前扫描 撤销未提交的事务

<start 事务T1>
……
<commit 事务T1>

<start 事务T2>
……
<commit 事务T2>
<ckpt>

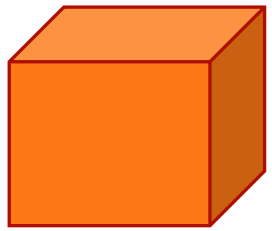
<start 事务T3>

<事务T 元素M 新值>

日志管理器实现

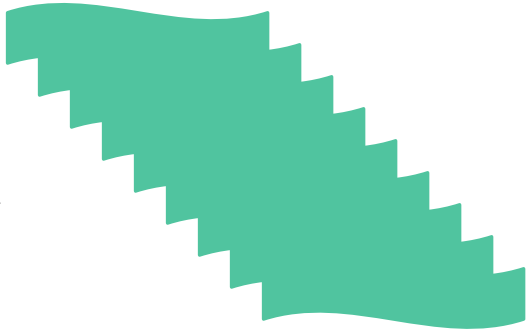
物理组织

磁盘文件：



<段文件>

32个



<页 8KB>

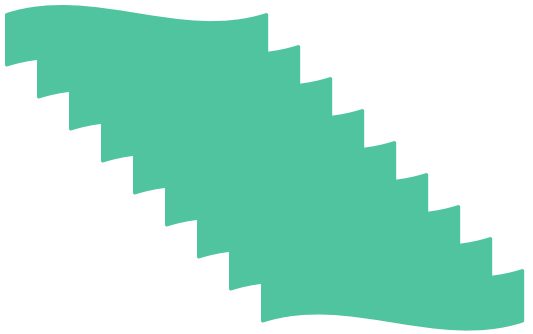
置换算法：

LRU

缓冲池：

<buffer pool>

8个



<页 8KB>

日志管理

日志管理器实现

CLOG

记录事务的最终状态

正在执行中 00

已提交 01

已终止 10

日志管理器实现

XLOG: 日志记录

段头信息：段文件大小 段文件中页面大小

页面头信息：首地址 第一条记录的时间序列

记录信息：日志序列号 事务ID 被修改数据在磁盘上的位置 记录长度 修改方式

日志管理器实现

缓冲池操作：

初始化

读取数据

写入数据

刷新

Clog操作：

查询事务状态

Xlog操作：

插入日志记录

日志文件的创建

日志文件的刷新

创建检查点

日志管理

Thanks !

- **To be done:**
  - 1. 不同模块间的融合
  - 2. 代码整合