B+树索引

本节主要内容

1 表索引

2 B+树索引

3 聚簇索引和非聚簇索引

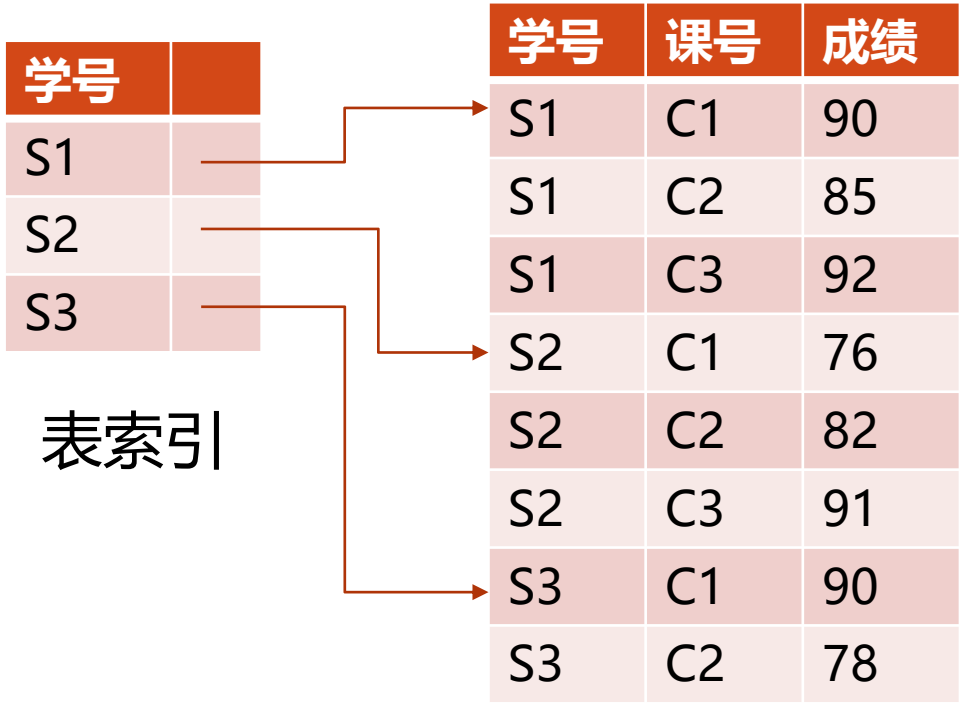
1 表索引

表索引

- 表索引建立在表属性子集的副本基础之上，是对这些属性的组织及排序，目的是为了使用这些属性的子集进行高效的检索（主要针对不是顺序扫描的检索方式）。

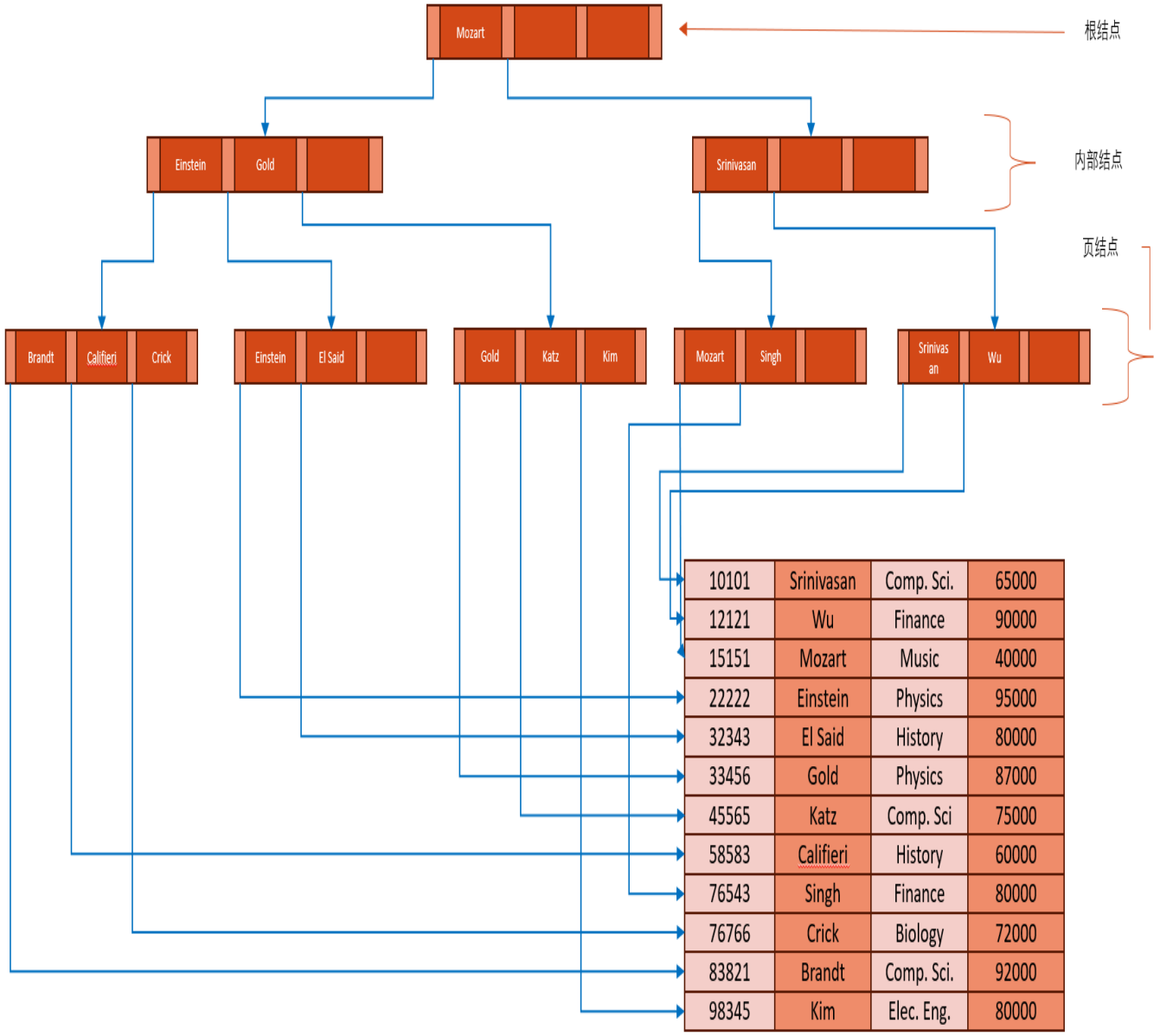
DBMS相关职责

- 维护表索引
  - 确保表的内容和索引在逻辑上是同步的。
- 执行每个查询时找出最佳的索引
- 在索引的数量和开销上进行权衡
  - 存储开销、维护开销



2 B+树索引

- 一种自平衡的树型数据结构
  - 由根结点，内部结点和叶结点构成
  - 只有一个根结点(保证搜索入口的唯一性，简化查询复杂度)
  - 从根结点到所有子结点的距离基本是一样的
- 一种多路搜索树，一个结点可以有两个以上的子结点。

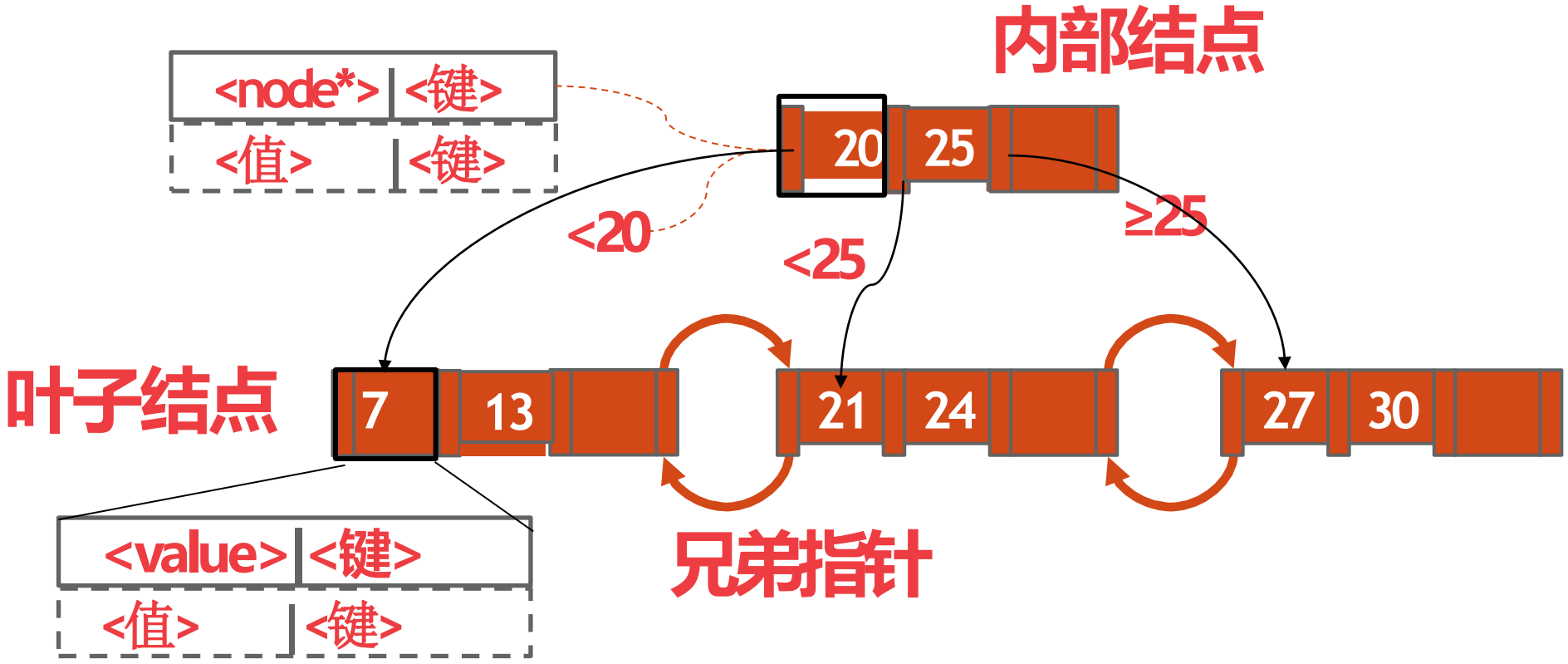


B+树的结点

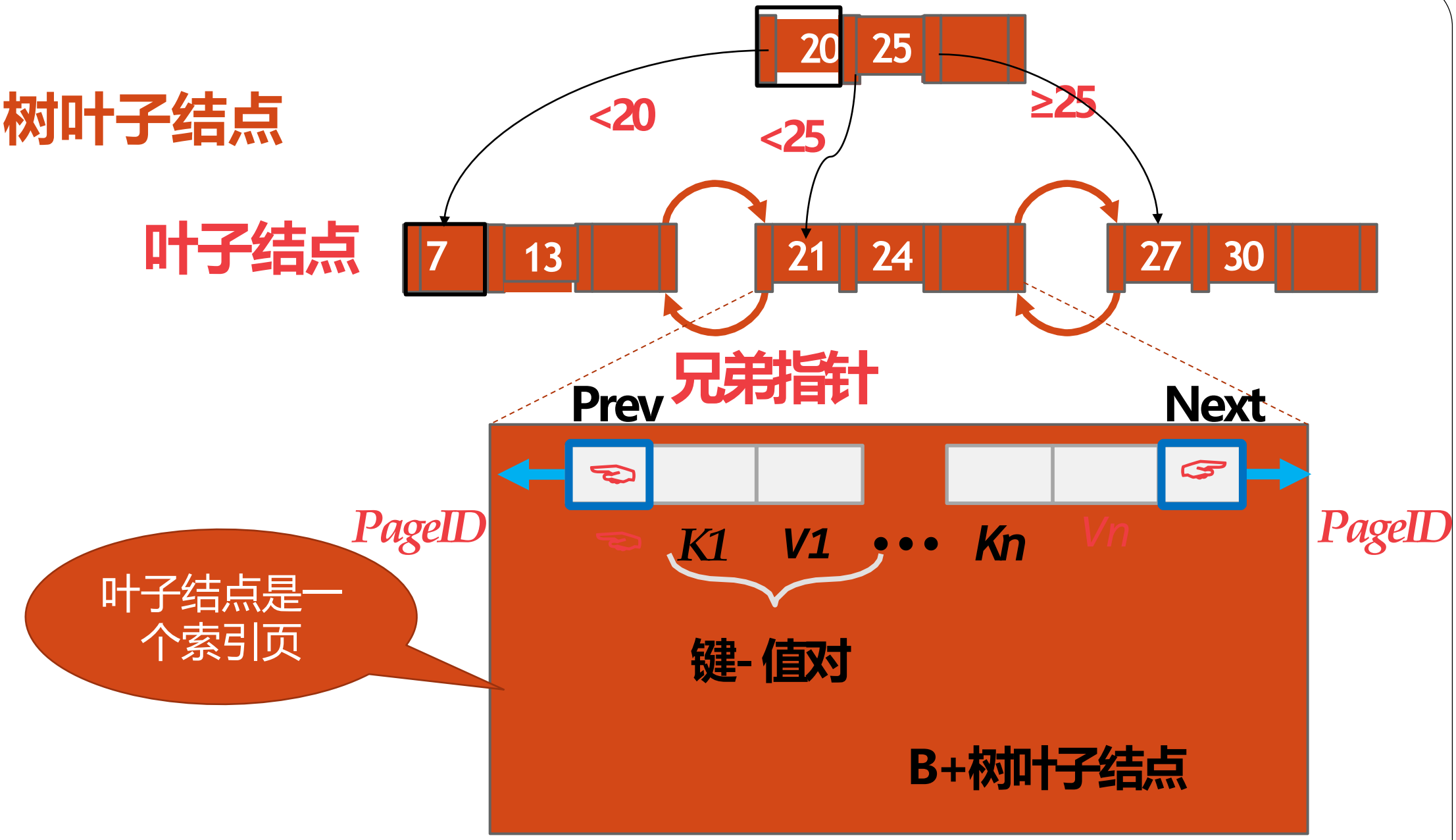
- B+树结点分为根结点、内部结点和叶结点
- 每个B+树结点由一个“键-值”数组组成
- 结点内的键值数组通常会按照键的顺序排序

“键-值”数组

P1	K1	P2	K2	.....	Pn-1	Kn-1	Pn

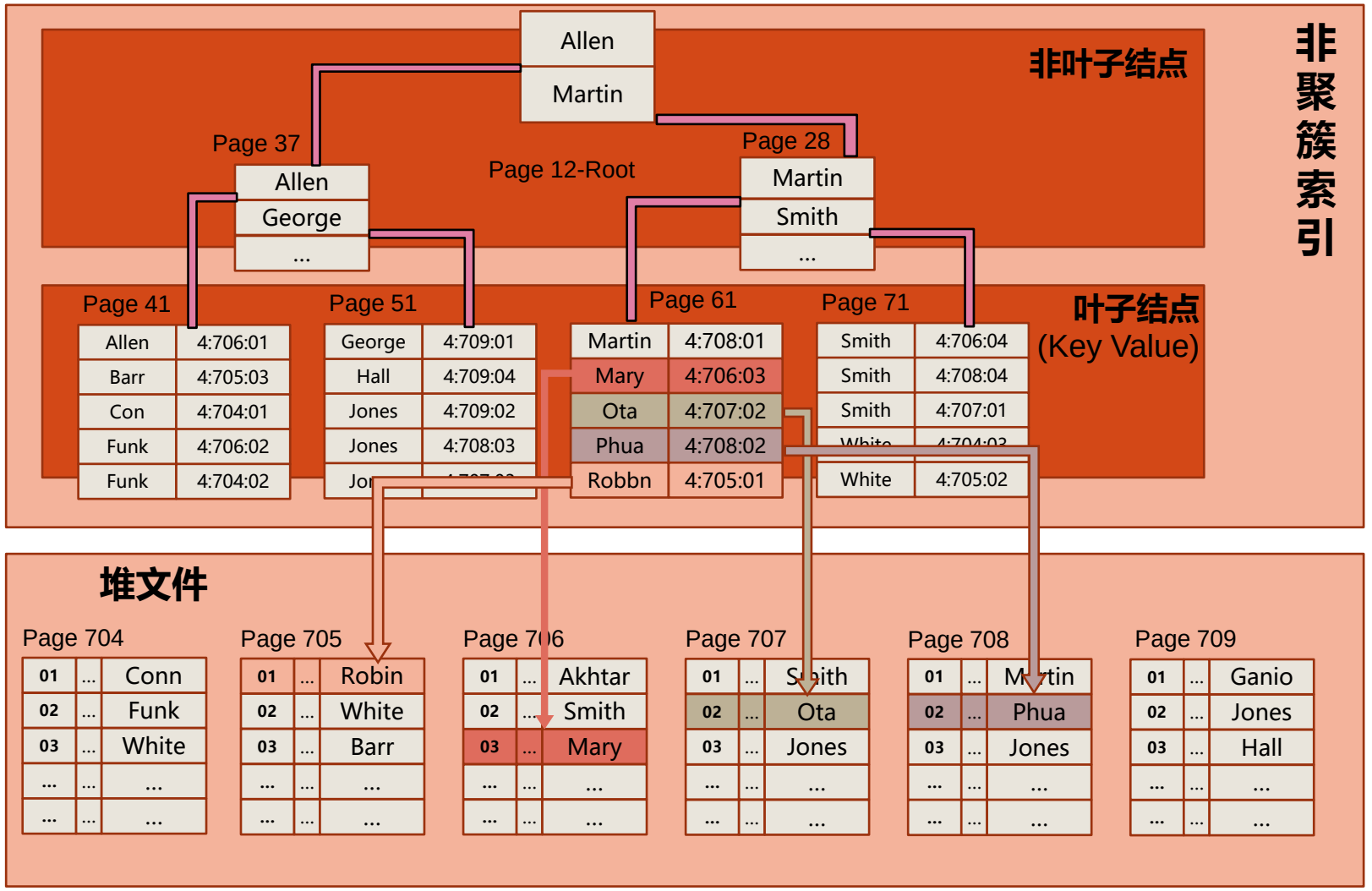


B+树叶子结点


叶子结点
20
25
<20
<25
≥25
7
13
21
24
27
30
兄弟指针
Prev
Next
PageID
PageID
K1
V1
Kn
Vn
键- 值对
B+树叶子结点
叶子结点是一个索引页

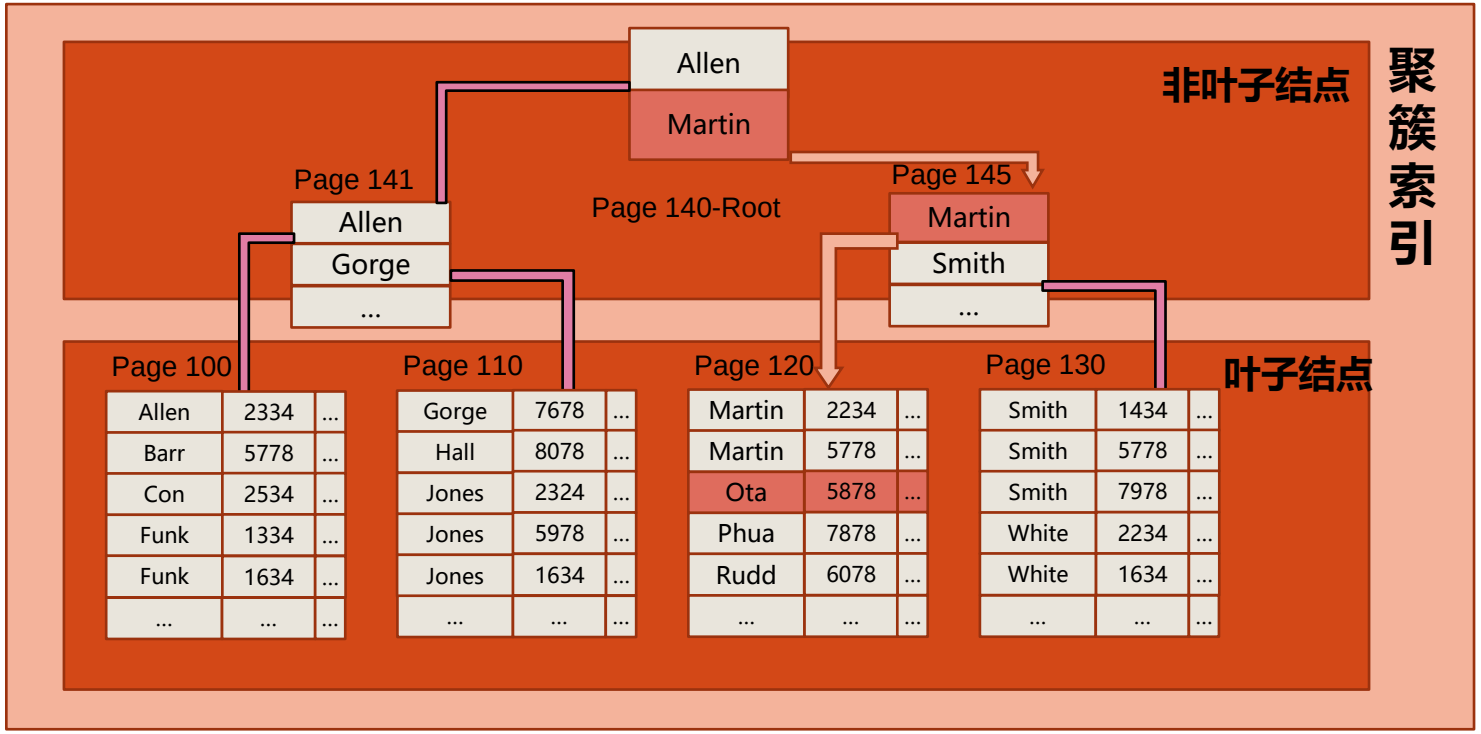
非聚簇索引（Nonclustered Indexes）

- 不影响原有的关系元组存储
- B+树的叶子结点的“值”一般指向元组所在的位置（元组ID）。



聚簇索引（Clustered Indexes）

- 关系按照“键”的排列顺序存储
  - 元组在heap页面集合中按照聚簇索引指定的顺序排序，
  - 叶子结点的值
    - 存放元组的实际数据
    - 也可以存放元组在堆文件的地址



缓冲池

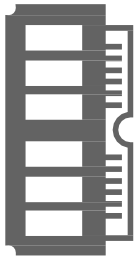
本节主要内容

1 缓冲池的工作原理

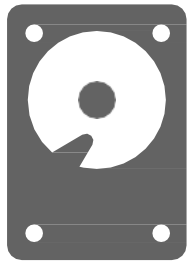
2 缓冲池结构

3 缓冲池替换算法

1 缓冲池的工作原理

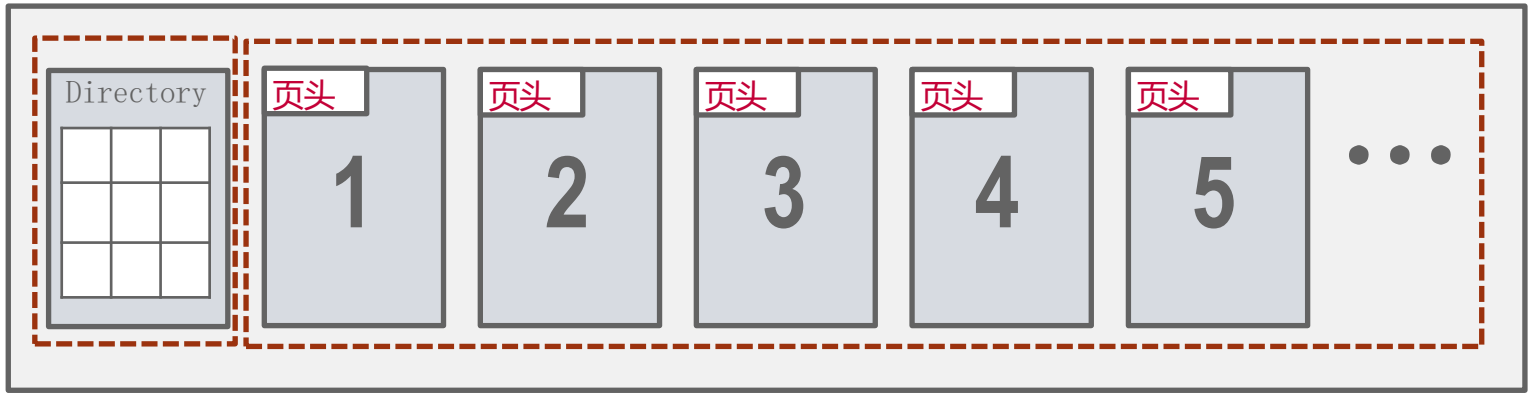


内存



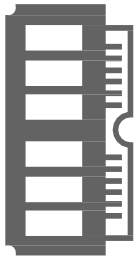
磁盘

数据文件



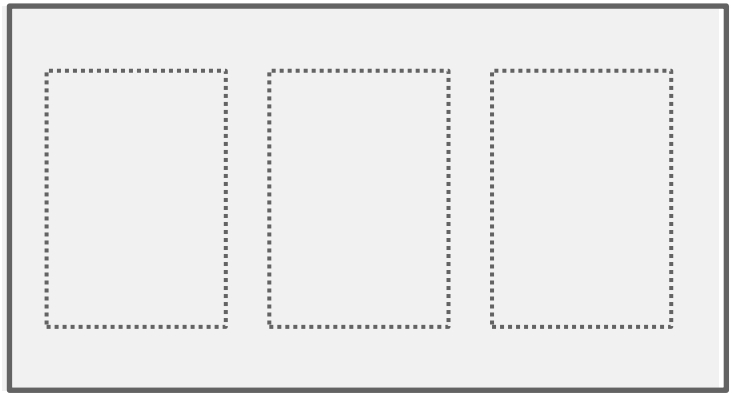
页

1 缓冲池的工作原理



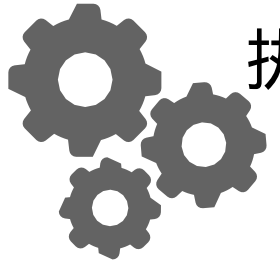
内存

缓冲池



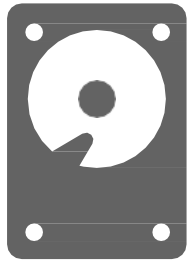
Buffer Pool

Buffer Cache



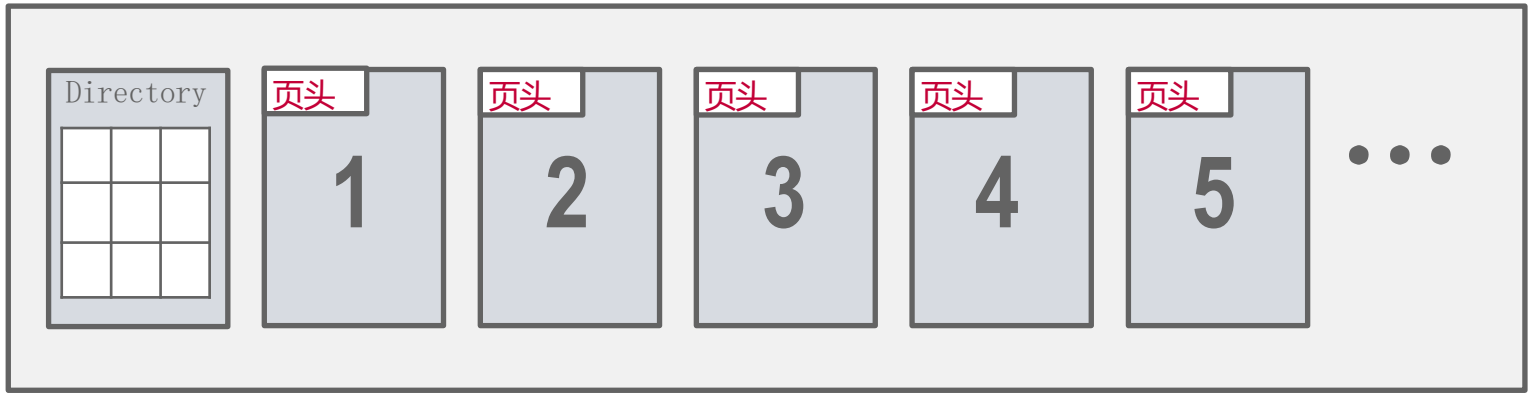
执行引擎

Buffer Pool Manager



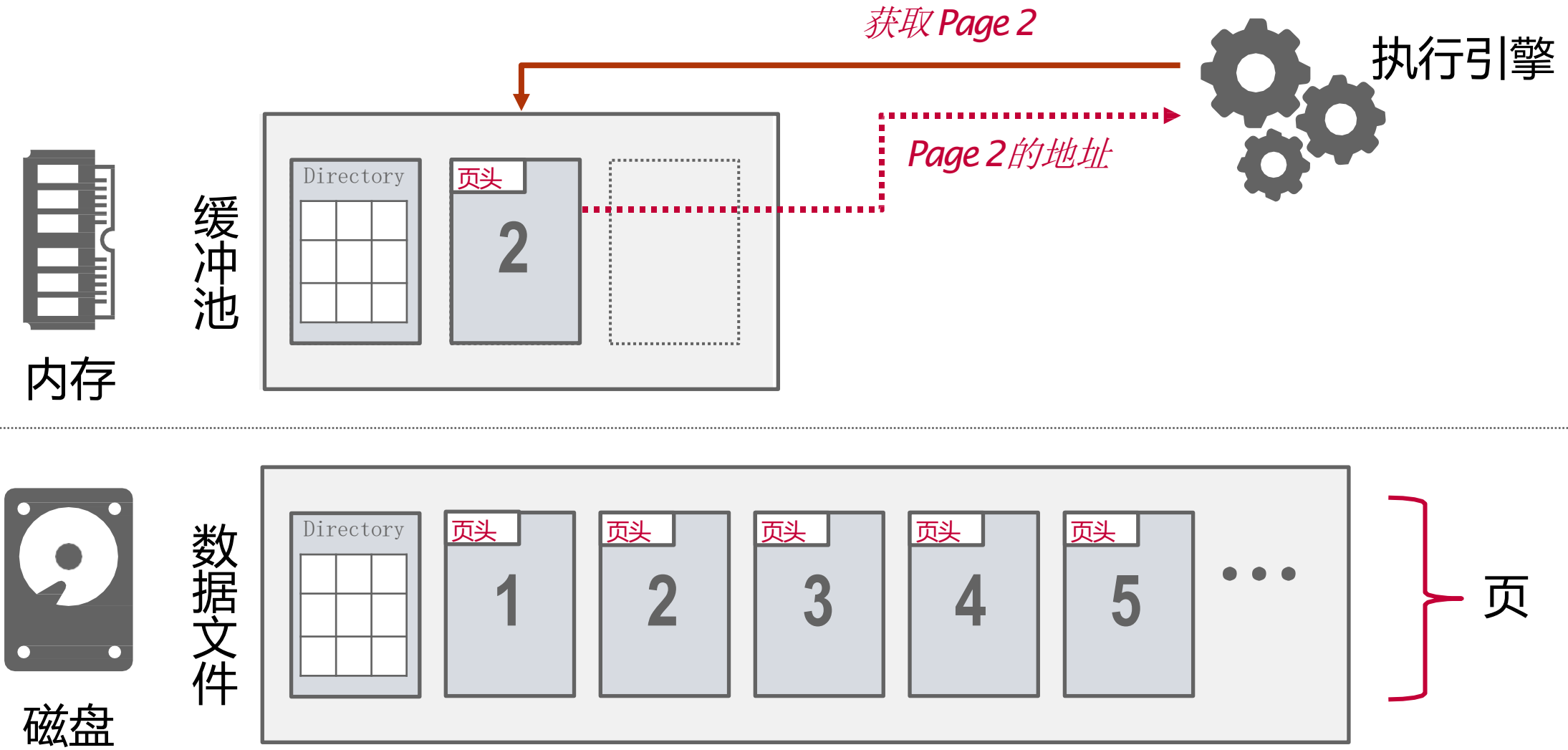
磁盘

数据文件

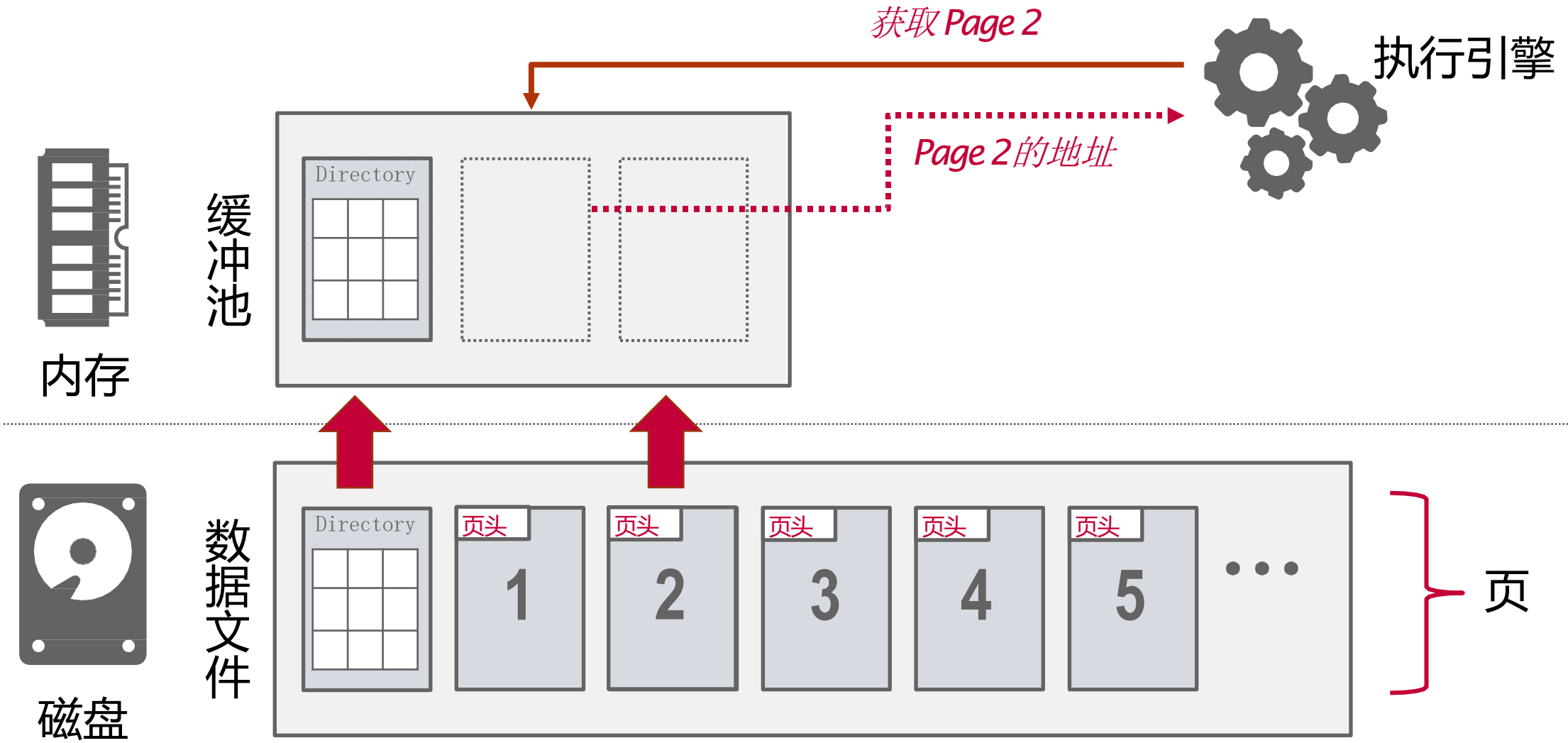


页

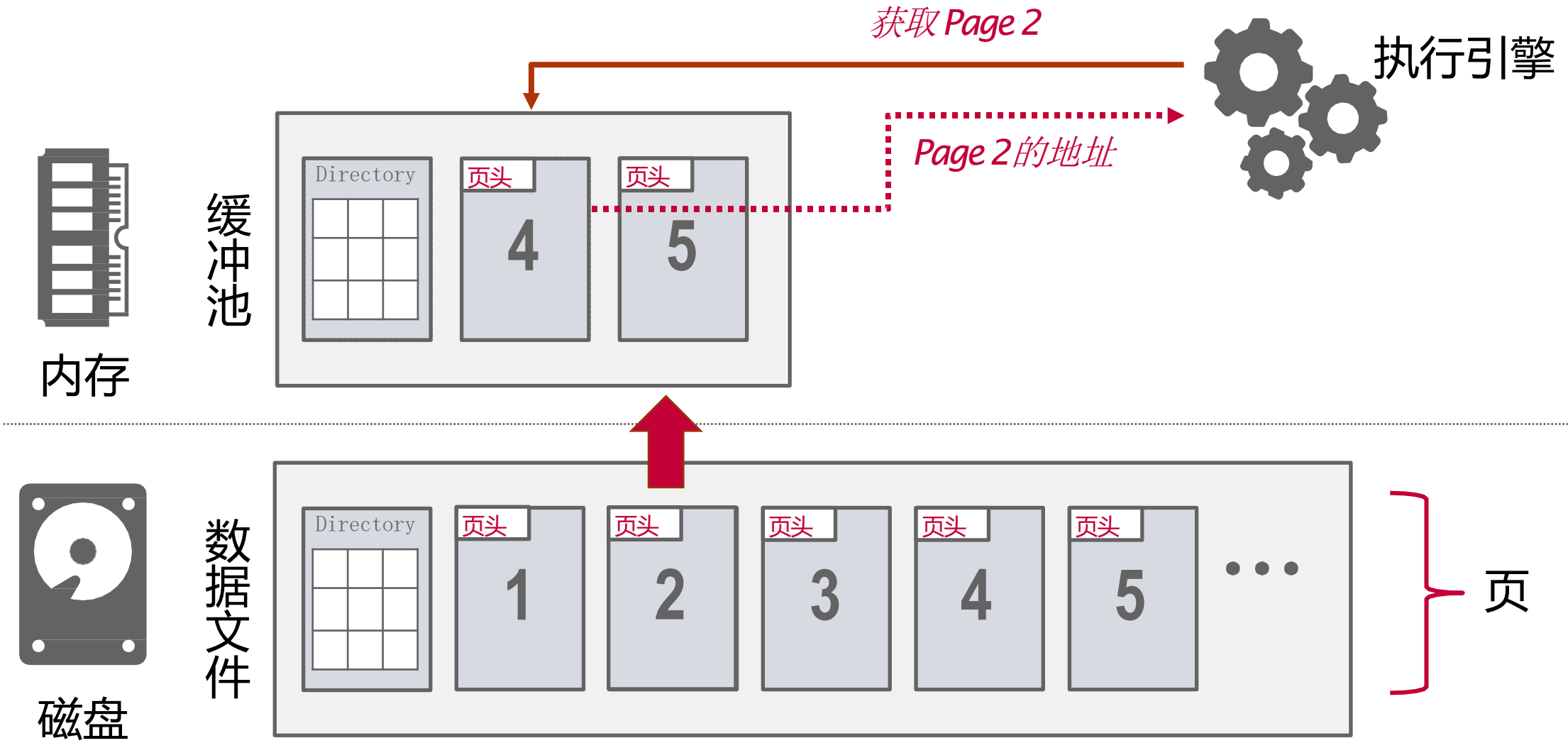
1 缓冲池的工作原理


获取 Page 2
执行引擎
Page 2 的地址
内存
缓冲池
Directory
页头
2
磁盘
数据文件
Directory
页头
1
页头
2
页头
3
页头
4
页头
5
页

1 缓冲池的工作原理


获取 Page 2
执行引擎
Page 2的地址
内存
缓冲池
Directory
磁盘
数据文件
Directory
页头
1
页头
2
页头
3
页头
4
页头
5
页

1 缓冲池的工作原理

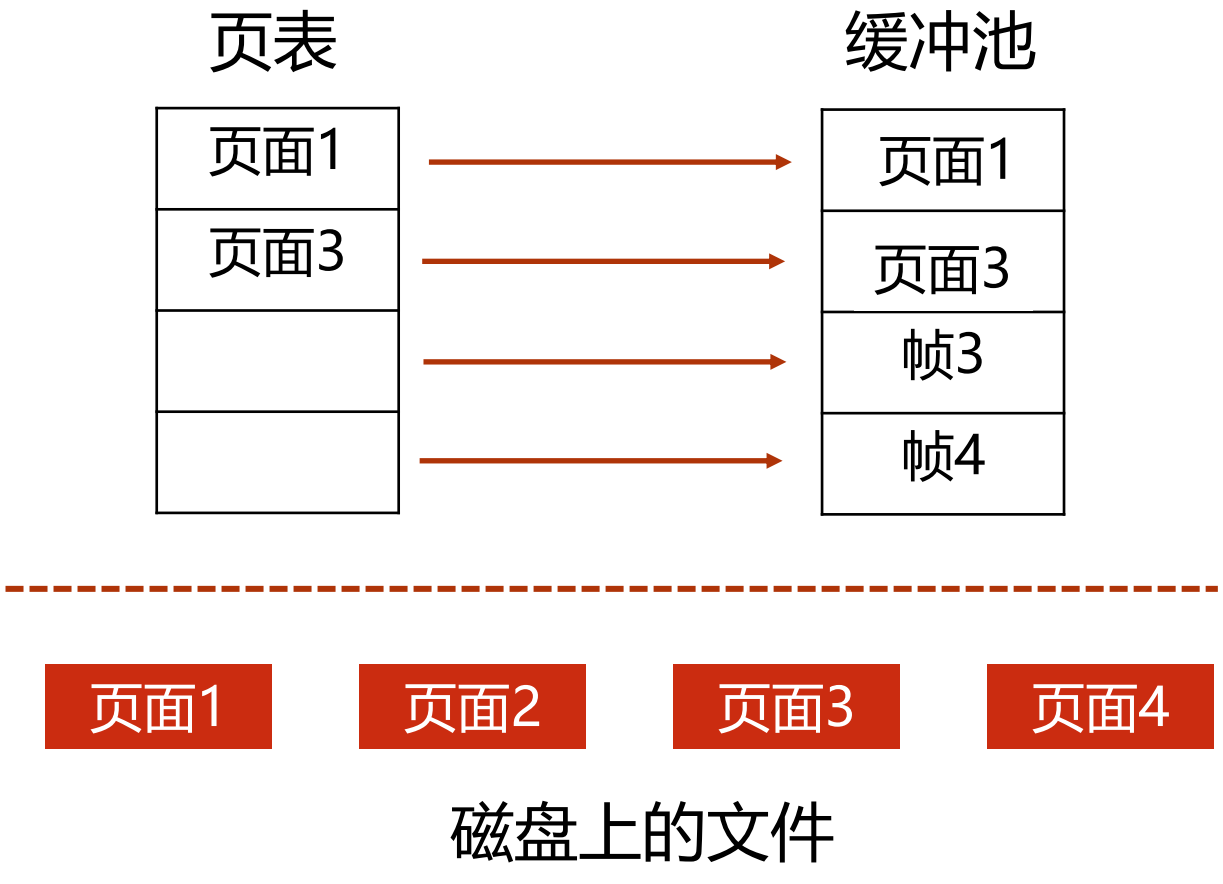

获取 Page 2
执行引擎
Page 2的地址
内存
缓冲池
Directory
页头
4
页头
5
磁盘
数据文件
Directory
页头
1
页头
2
页头
3
页头
4
页头
5
页

缓冲池的作用

- 减少磁盘I/O，提升页面读写效率。
  - 读操作时，如果所需页面已经加载到缓冲池，就不需要从磁盘读了。
    - 利用“局部性原理”，即如果某个数据被访问，它的近邻数据大概率也会被访问。因此预先把这些数据加载到缓冲池中，就可以减少磁盘IO。
  - 写操作时，如果多次修改了同一页面，只需要写一次磁盘。
    - 对于已经修改的页面也不要立即写回磁盘，因为这个页面中刚刚修改的元组的下一个元组很有可能马上也会被修改。不妨在稍后某个时间点，比如检查点，再写回磁盘。

2 缓冲池结构

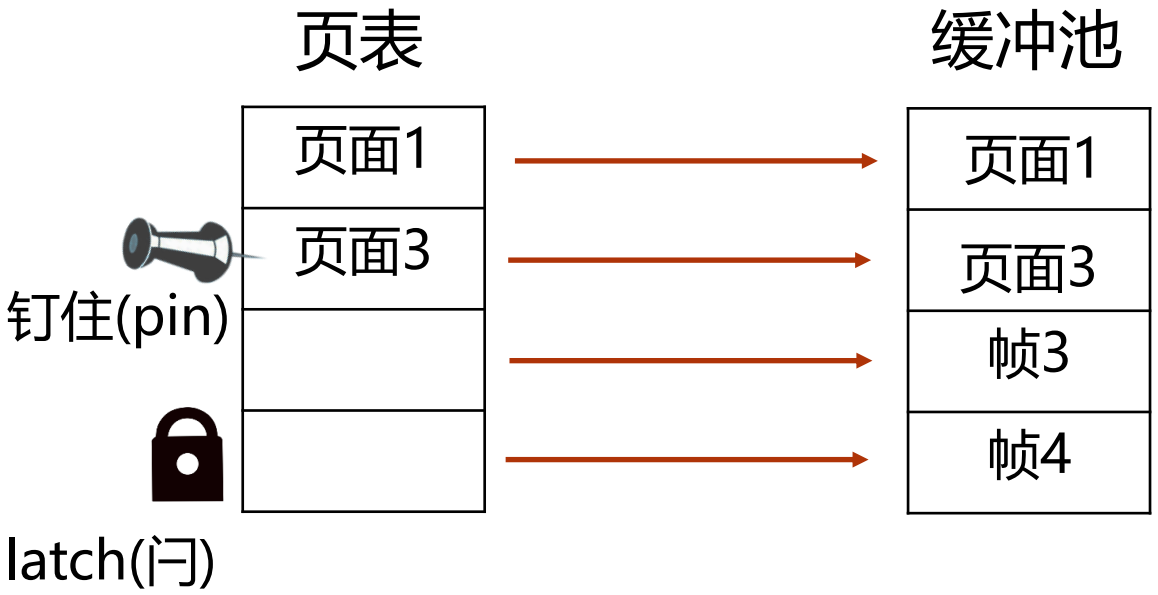
- 存储从磁盘获取页面的内存区域。
- 内存空间被组织为一个数组每个数组项被称为一个帧一个帧正好能放置一个页面。
- 维护缓冲池元数据的数据结构-页表。



缓冲池页表

维护页的元数据：

- 帧和页面的映射关系(位置信息)
- 脏标记
- 引用计数（钉住/pin）
- 闩(latch)



脏页的处理

在淘汰页面时，对于脏页可以有两种情况：

- 快速：优先淘汰非脏页面（可能将未来不需要的脏页留在缓冲池）；
- 慢速：将脏页写回磁盘后再将其淘汰（这将降低替换页面的速度）。

另一种处理方法是**后台写(Backgroud Writing)**：

DBMS定期遍历页表并将脏页写入磁盘，并将脏页标记复位，避免在淘汰页面时才执行页面写出操作。

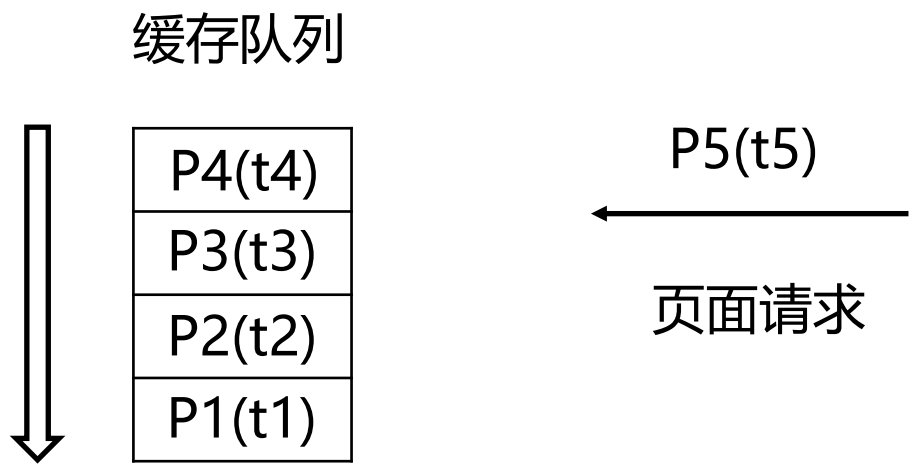
特别需要注意的是，写脏页前，必须保证先写日志。

3 缓冲池替换算法

LRU(Least Recently Used)

- 维护页面最后一次被访问的时间戳
- 总是选择淘汰时间戳最早的页面。

假设：t1<t2<t3<t4<t5，缓冲池SIZE=4

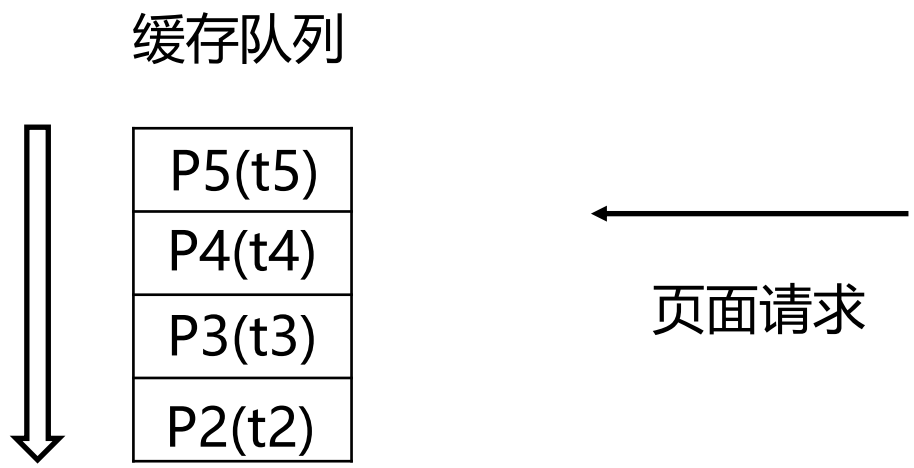


3 缓冲池替换算法

LRU(Least Recently Used)

- 维护页面最后一次被访问的时间戳
- 总是选择淘汰时间戳最早的页面。

假设：t1<t2<t3<t4<t5，缓冲池SIZE=4



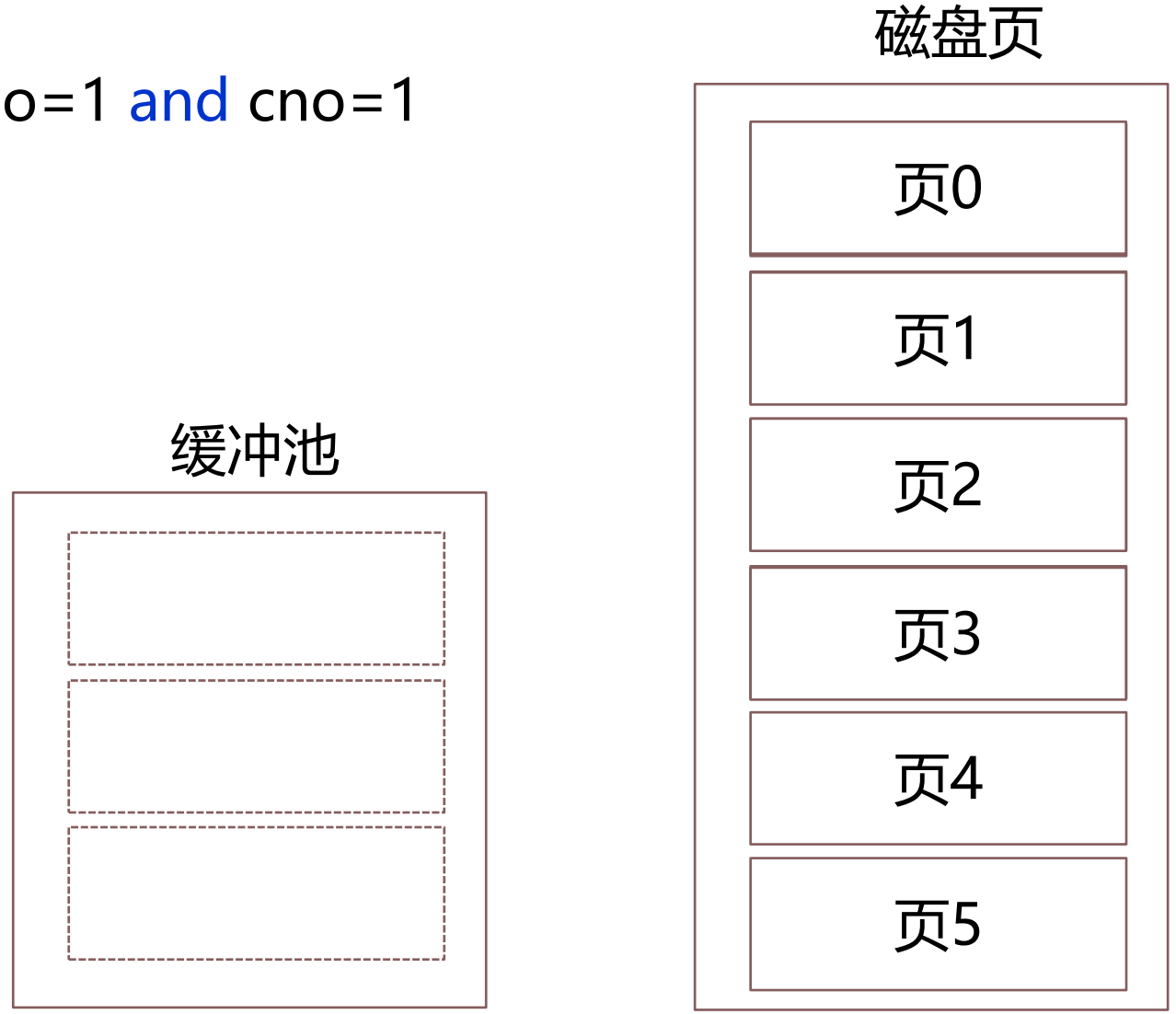
LRU算法的缺点

容易受到顺序洪泛（sequential flooding）问题的影响

- 因一次顺序扫描需将表的所有页面读入缓存，导致缓存污染问题。
- 缓存污染：在缓存机制中，会存在把不常用的数据读取到缓存中的现象，这种现象称为缓存污染。

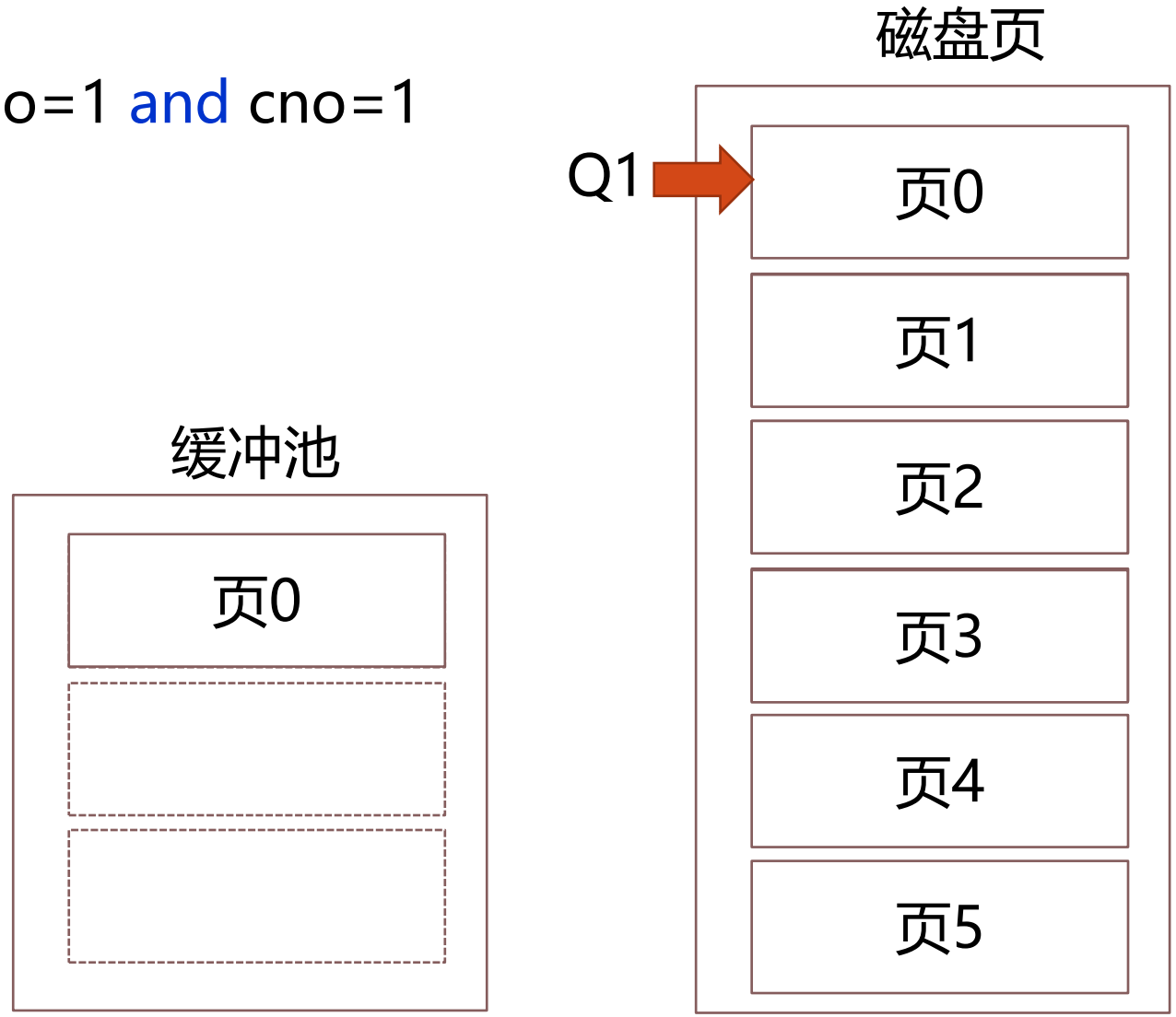
顺序洪泛的例子

Q1　Select * from SC where sno=1 and cno=1



顺序洪泛的例子

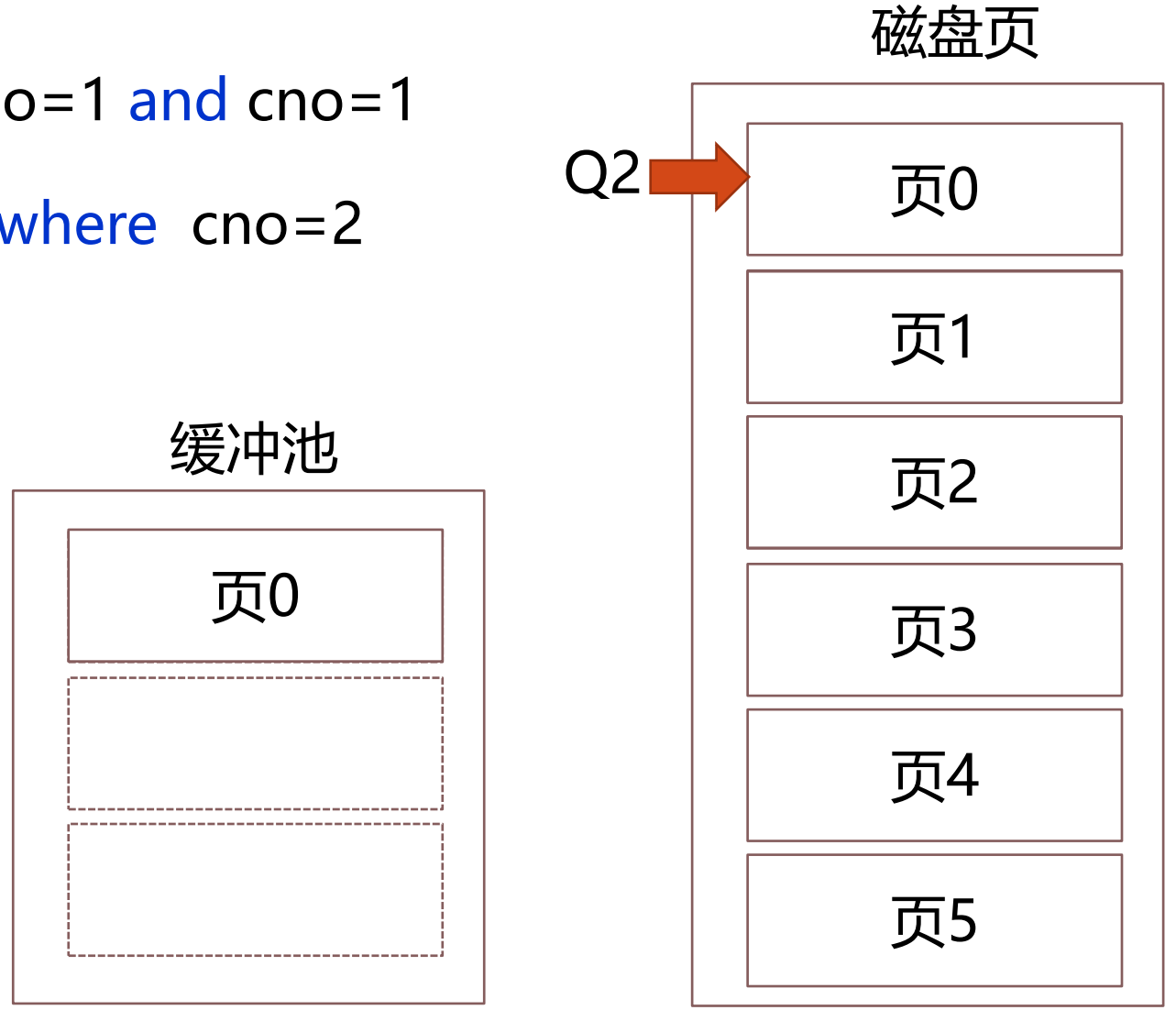
Q1　Select * from SC where sno=1 and cno=1



顺序洪泛的例子

Q1 Select * from SC where sno=1 and cno=1

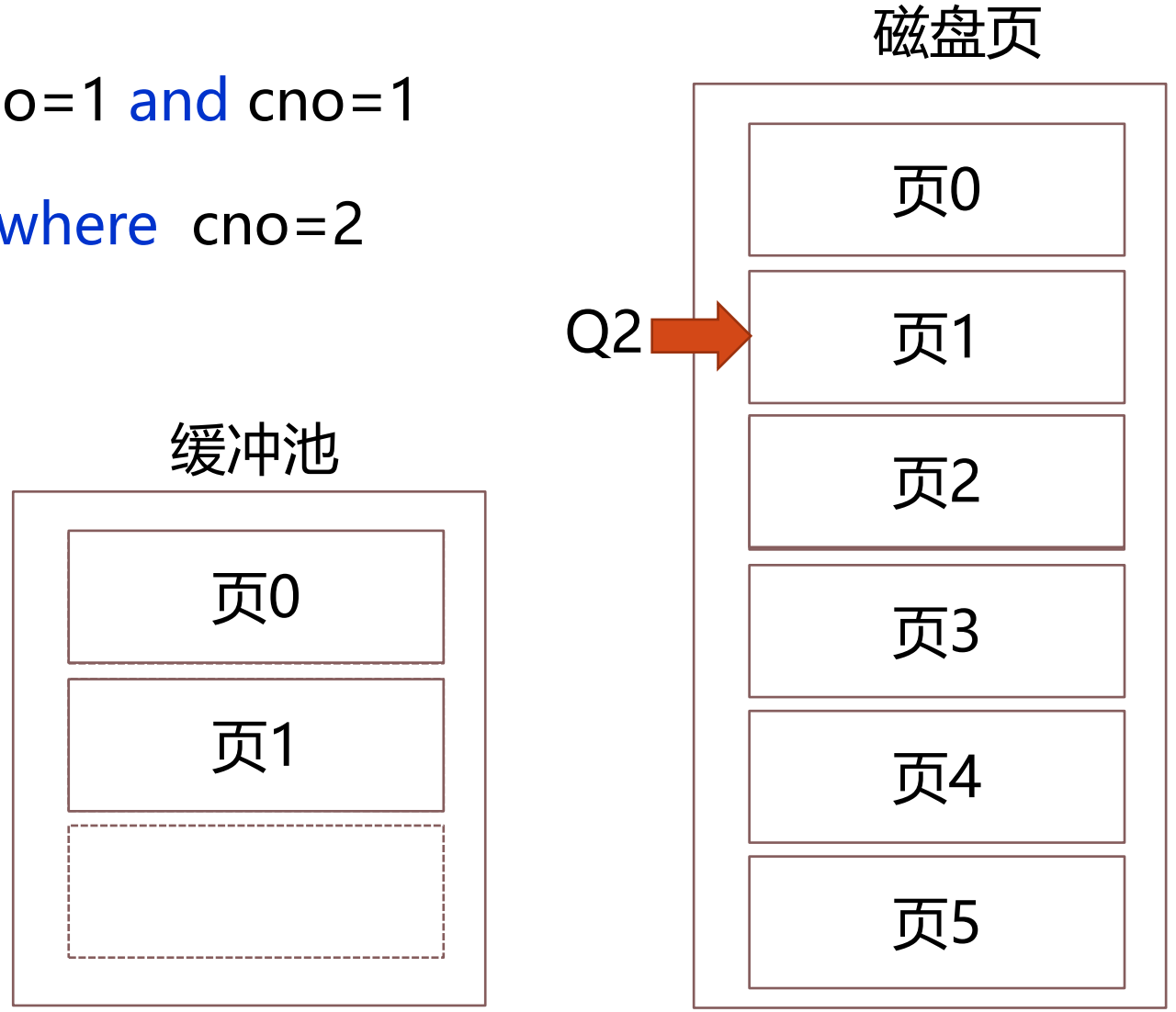
Q2 Select avg(score) from SC where cno=2



顺序洪泛的例子

Q1 Select * from SC where sno=1 and cno=1

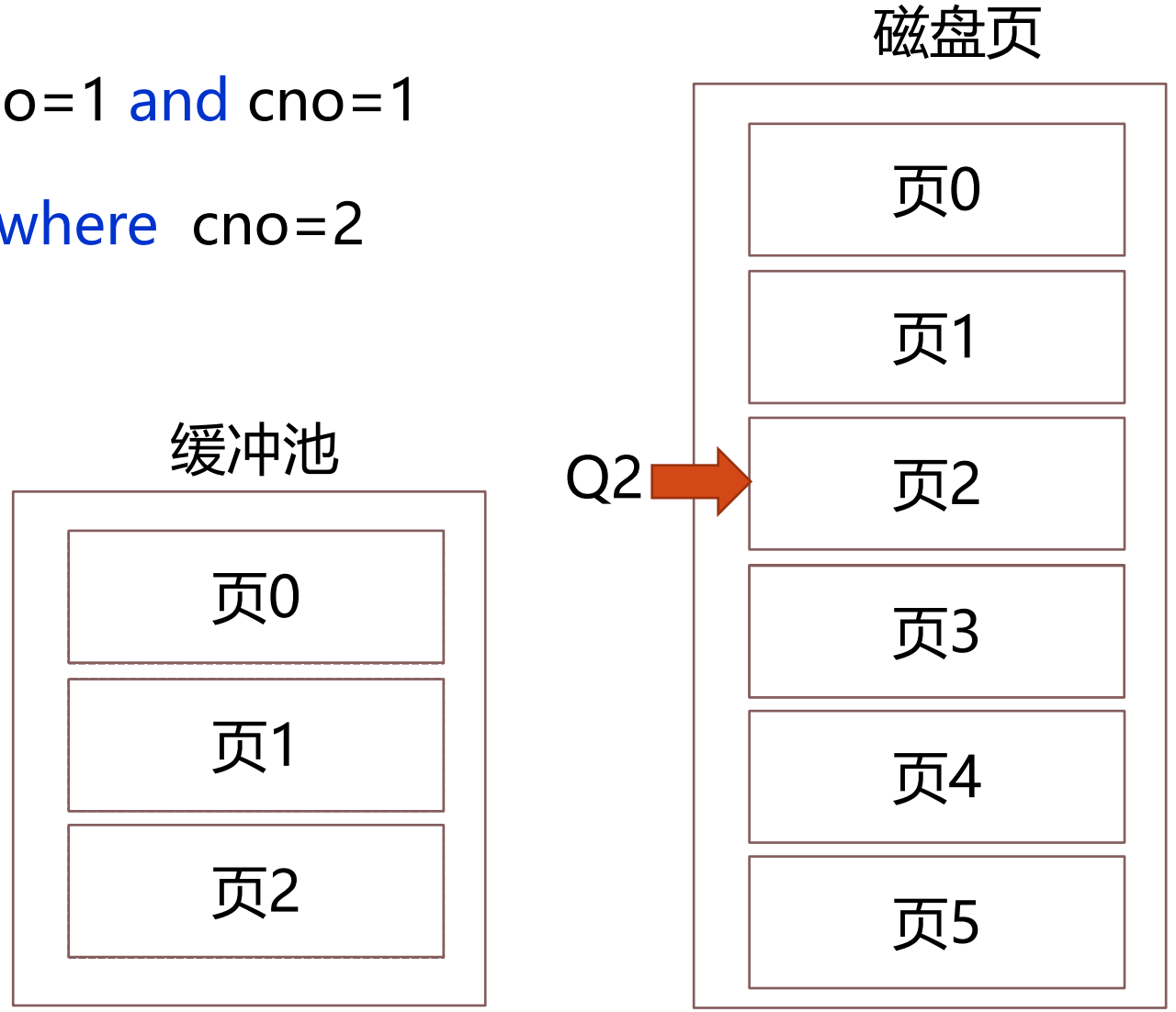
Q2 Select avg(score) from SC where cno=2



顺序洪泛的例子

Q1 Select * from SC where sno=1 and cno=1

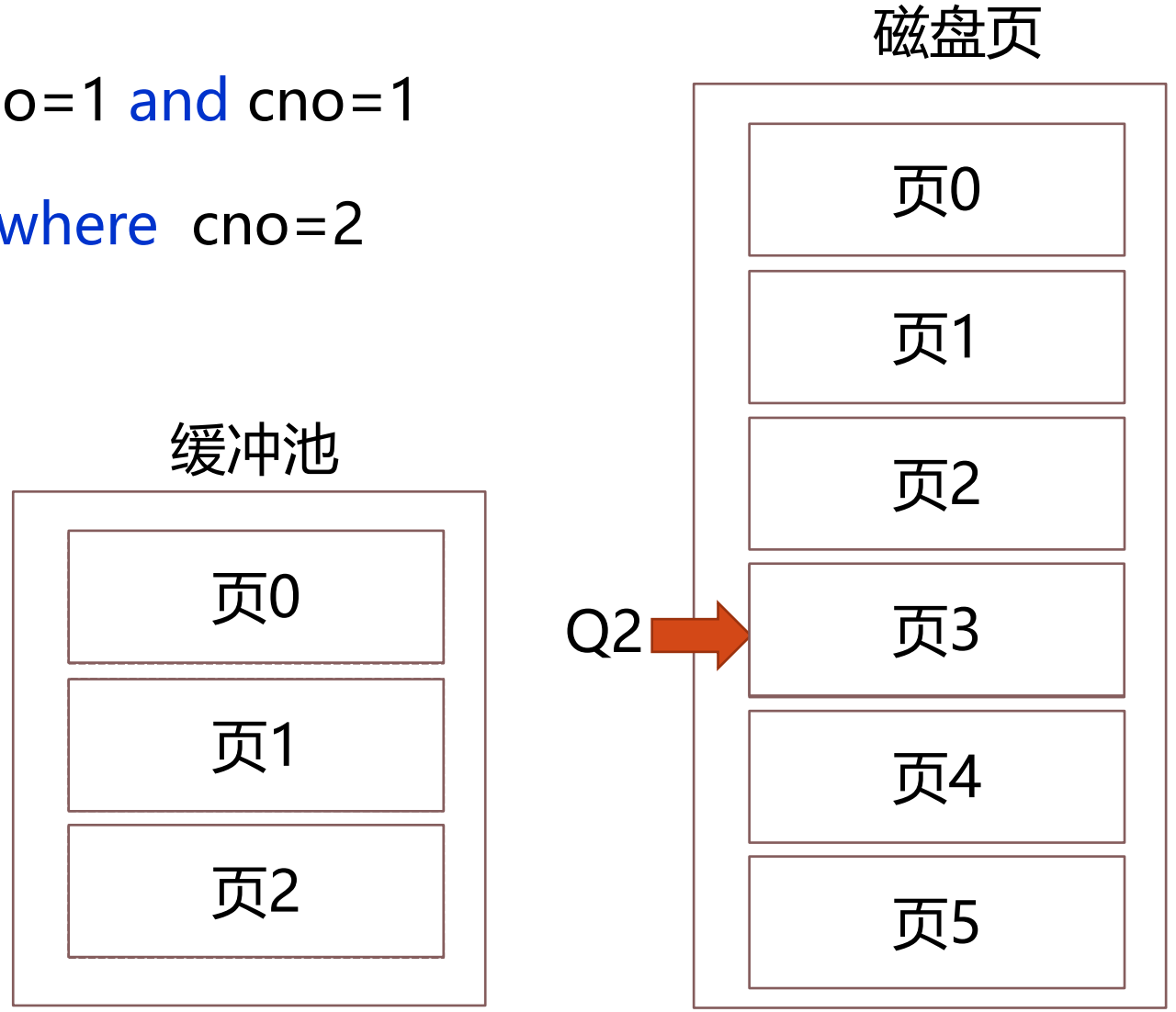
Q2 Select avg(score) from SC where cno=2



顺序洪泛的例子

Q1 Select * from SC where sno=1 and cno=1

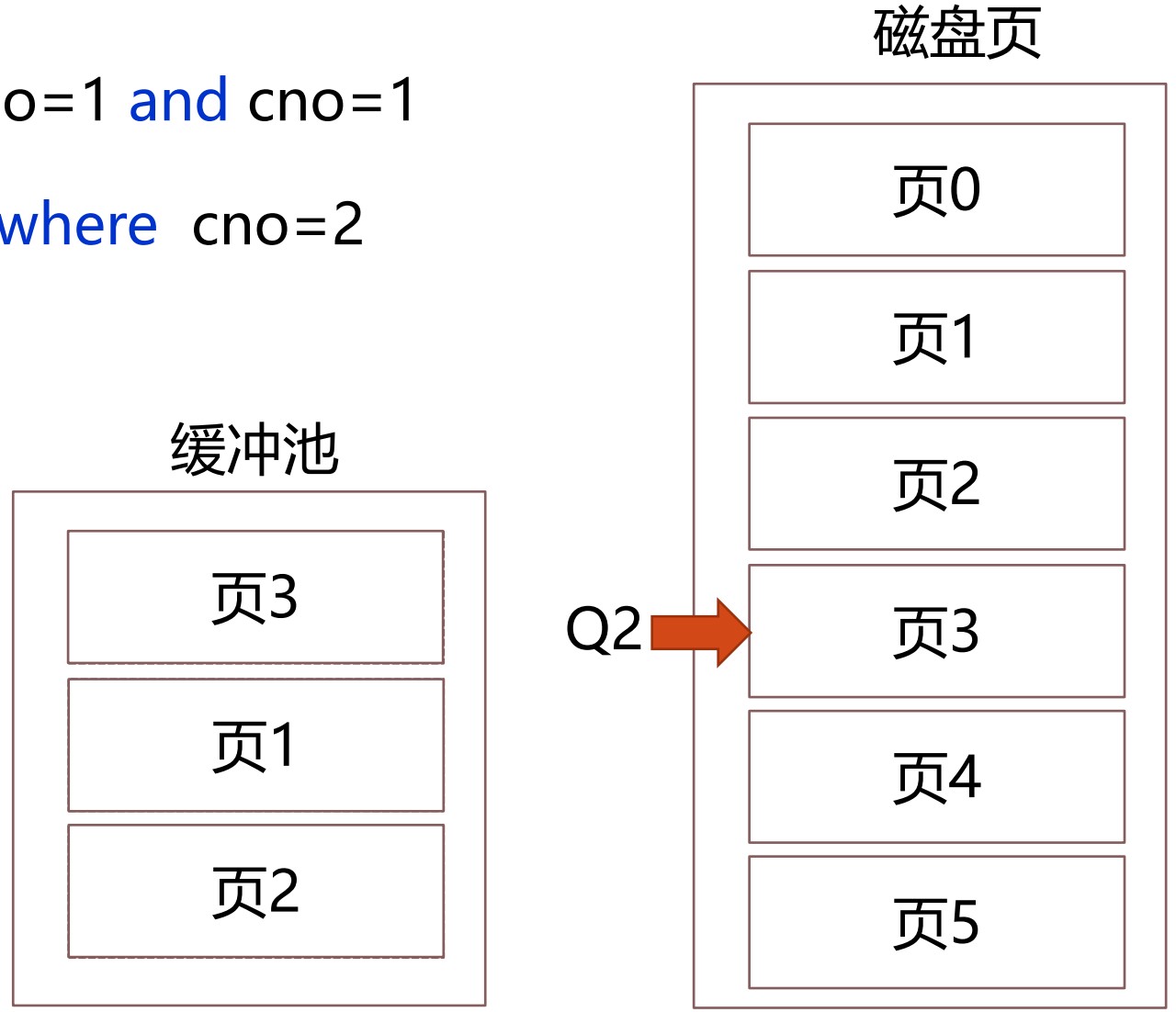
Q2 Select avg(score) from SC where cno=2



顺序洪泛的例子

Q1 Select * from SC where sno=1 and cno=1

Q2 Select avg(score) from SC where cno=2



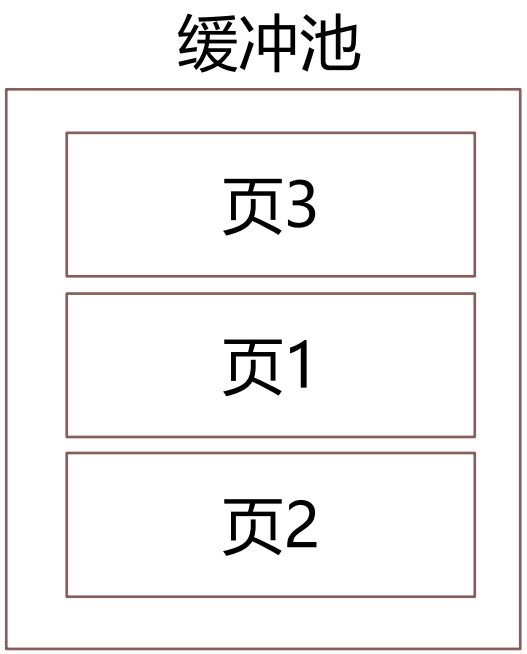
顺序洪泛的例子

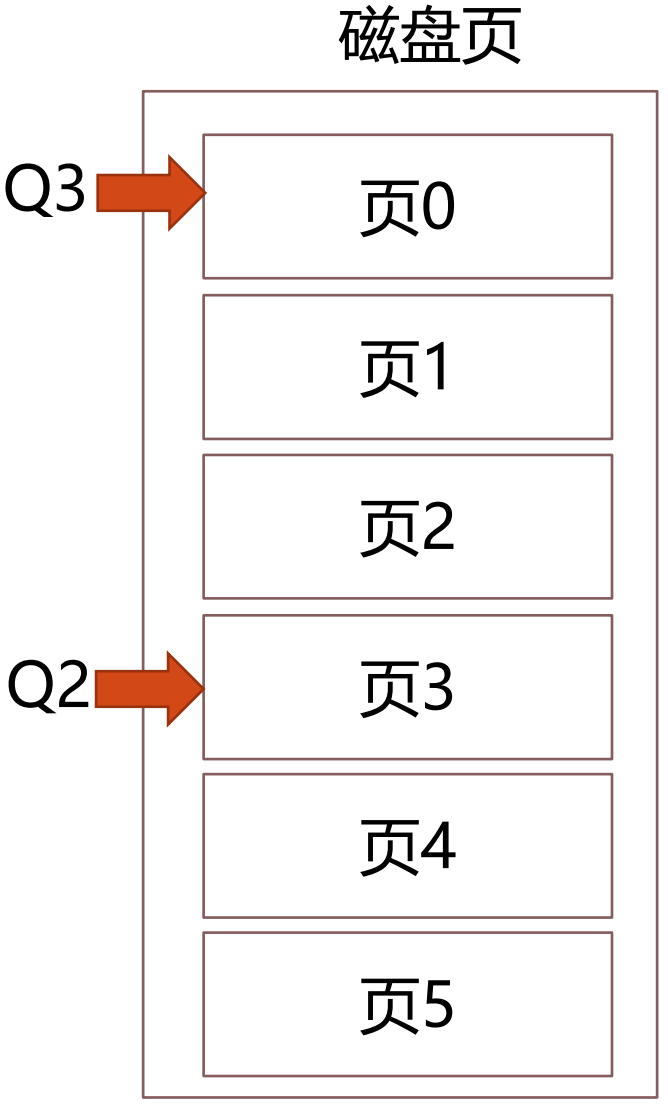
Q1 Select * from SC where sno=1 and cno=1

Q2 Select avg(score) from SC where cno=2

Q3 Select * from SC where sno=1 and cno=1

由于Q2的顺序扫描操作，导致真正会被再次访问的第0页被淘汰。





LRU-K算法

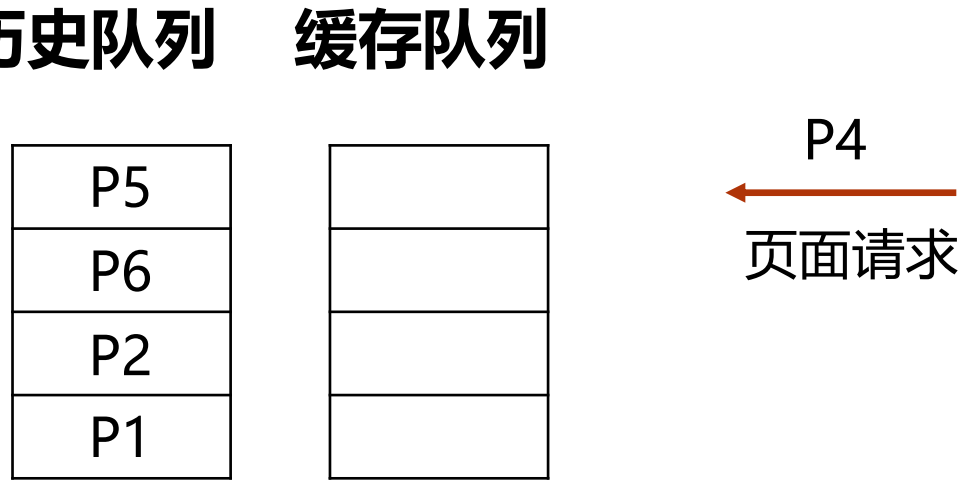
https://dl.acm.org/doi/epdf/10.1145/170036.170081

核心思路是将最近使用过1次的判断标准扩展为最近使用过K次。

LRU-K算法每次驱逐最近第k次访问距今时间最久，或者说k距离最大的页面，少于k个访问记录的页面，其k距离被赋予正无穷大。当缓冲池中有多个页面的 k 距离为正无穷大时，可采用普通的LRU算法淘汰时间戳最早的页面。

LRU-K算法

假设K=2，SIZE=4



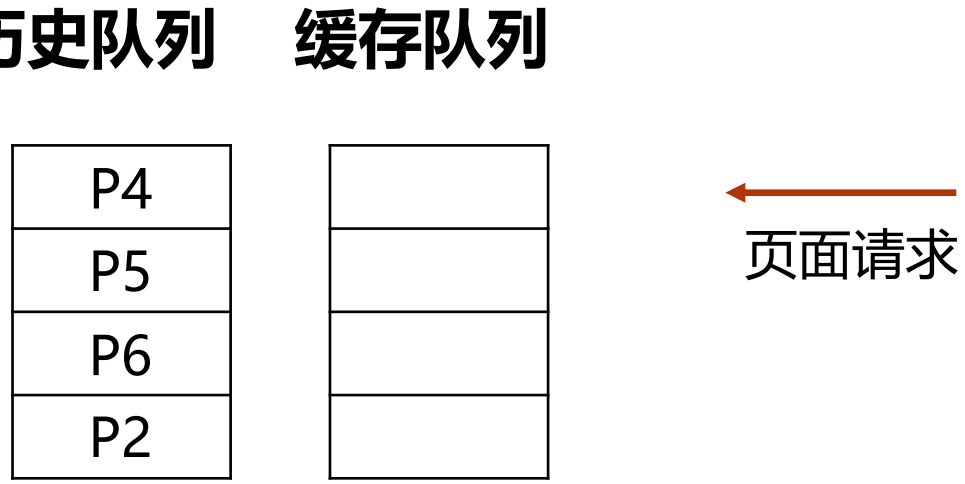
访问顺序：

P1 P2 P6 P5

- 页面第一次被访问，添加到历史队列中。若队列满，依LRU淘汰。
- 第二次访问，移至缓存队列，若缓存队列满，依LRU-K淘汰。

LRU-K算法

假设K=2，SIZE=4



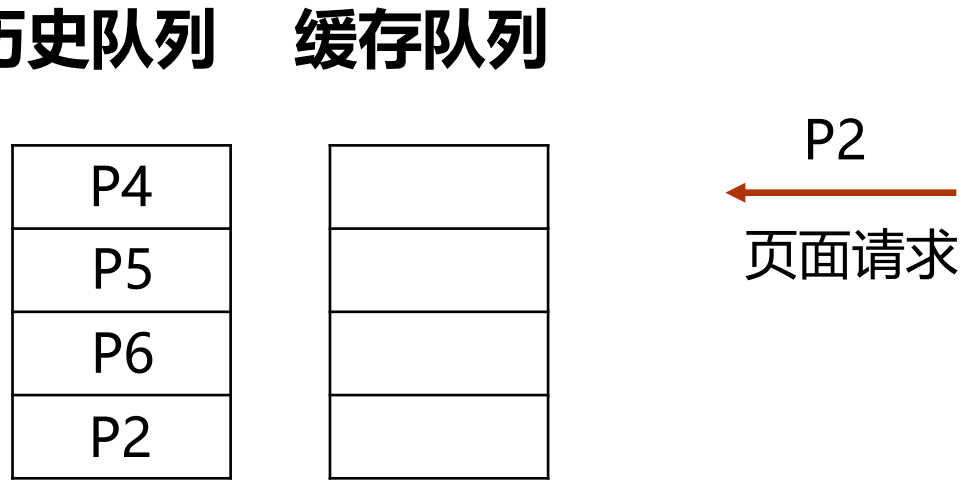
访问顺序：

P1 P2 P6 P5 P4

- 页面第一次被访问，添加到历史队列中。若队列满，依LRU淘汰。
- 第二次访问，移至缓存队列，若缓存队列满，依LRU-K淘汰。

LRU-K算法

假设K=2，SIZE=4



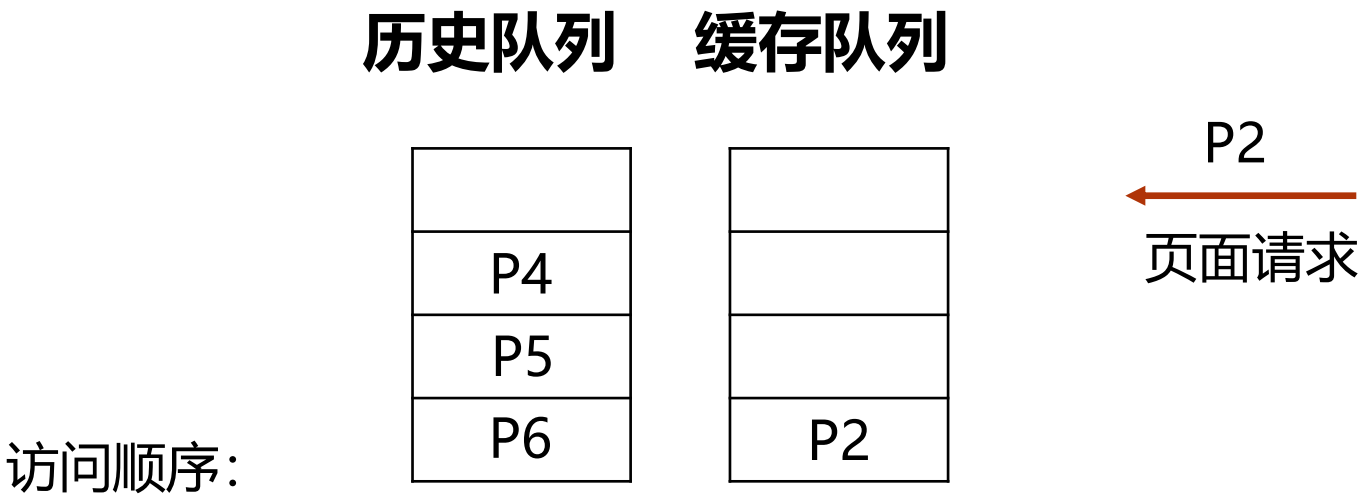
访问顺序：

P1 P2 P6 P5 P4

- 当历史队列中的某个页面第K次被访问时，该页面从历史队列中移出，并存放至缓存队列。

LRU-K算法

假设K=2，SIZE=4



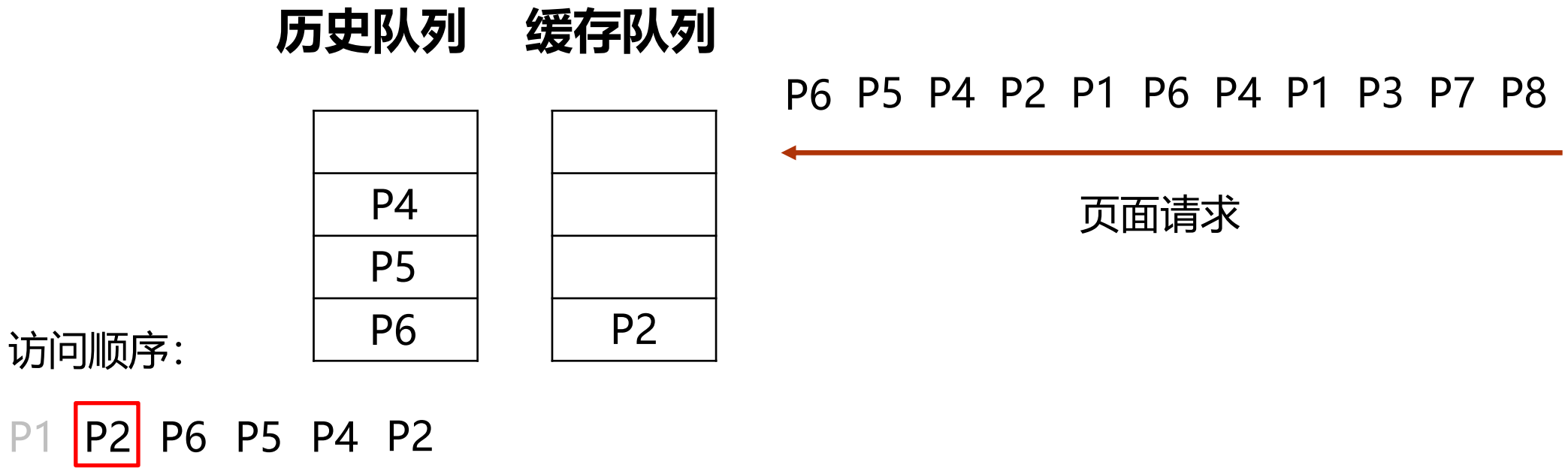
访问顺序：

P1 P2 P6 P5 P4 P2

- 当历史队列中的某个页面第K次被访问时，该页面从历史队列中移出，并存放至缓存队列。

LRU-K算法

假设K=2，SIZE=4



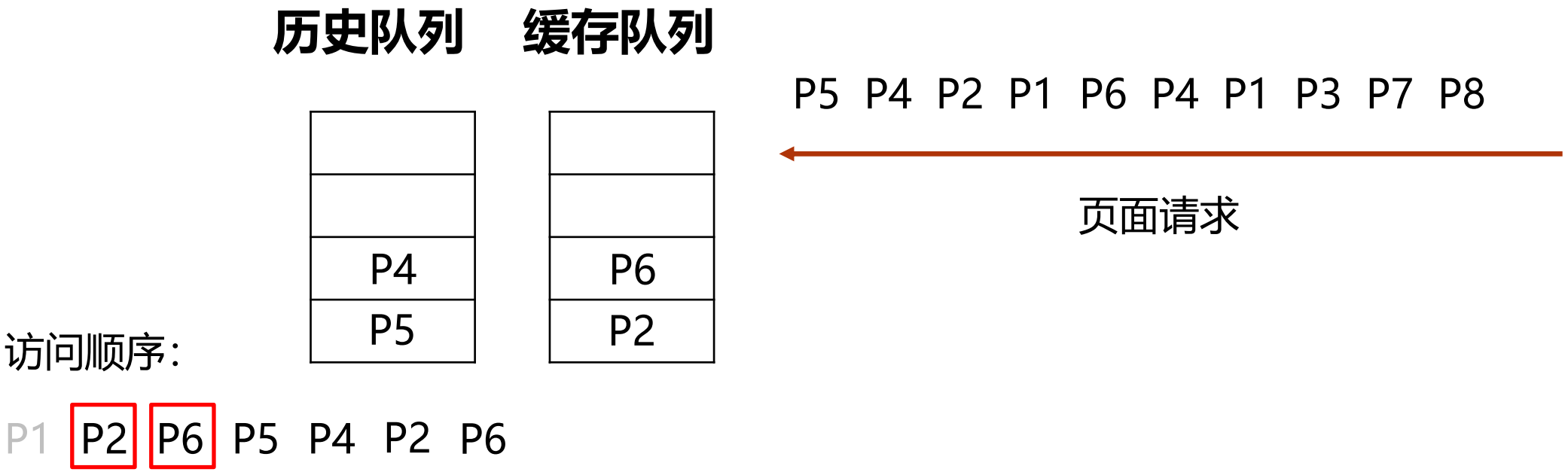
访问顺序：

P1 P2 P6 P5 P4 P2

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



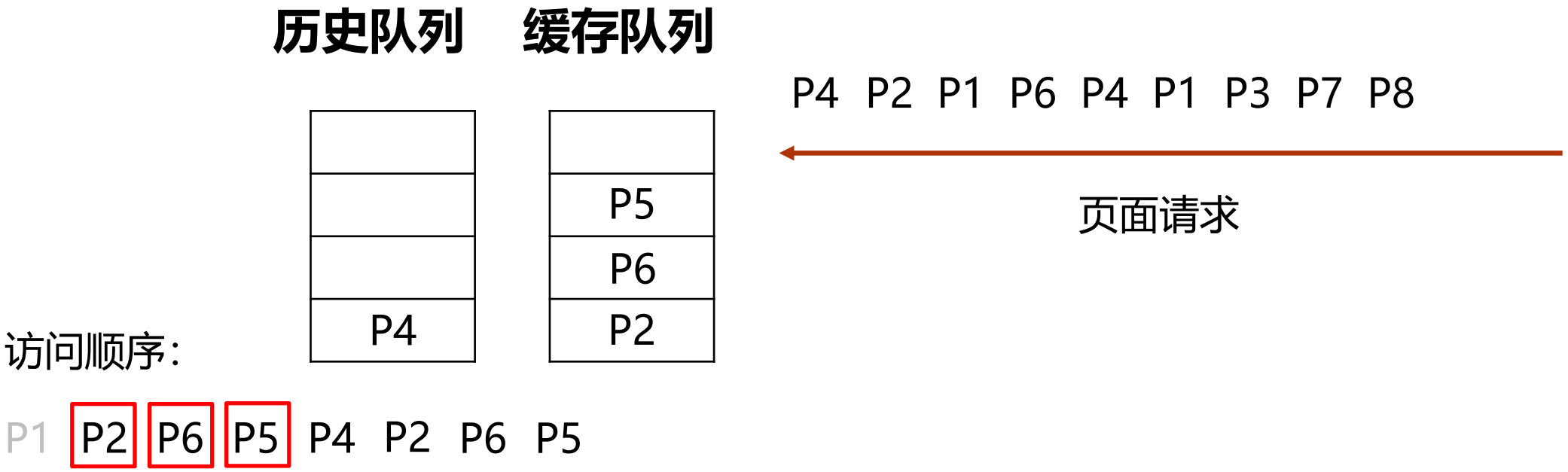
访问顺序：

P1 P2 P6 P5 P4 P2 P6

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



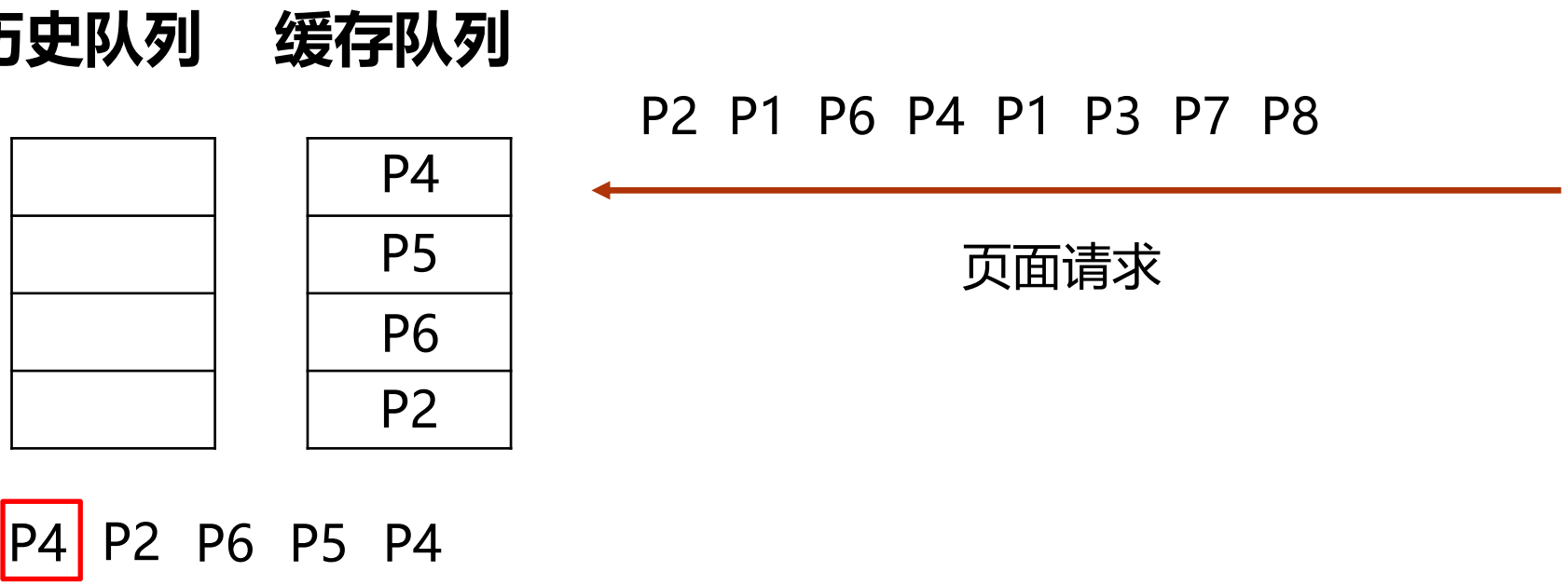
访问顺序：

P1 P2 P6 P5 P4 P2 P6 P5

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



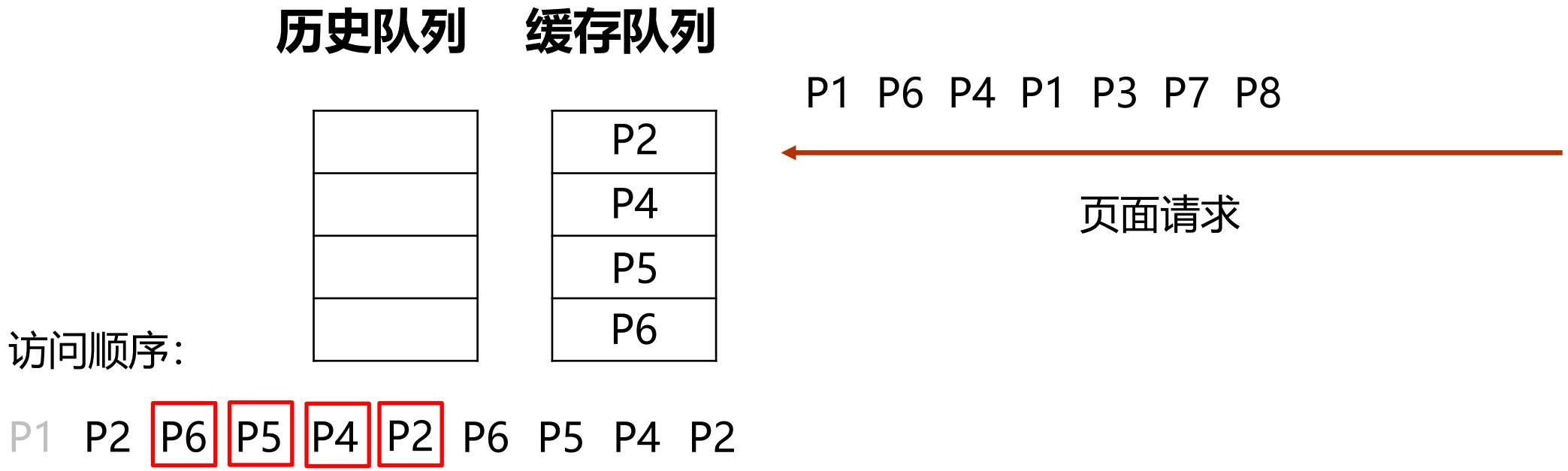
访问顺序：

P1 P2 P6 P5 P4 P2 P6 P5 P4

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

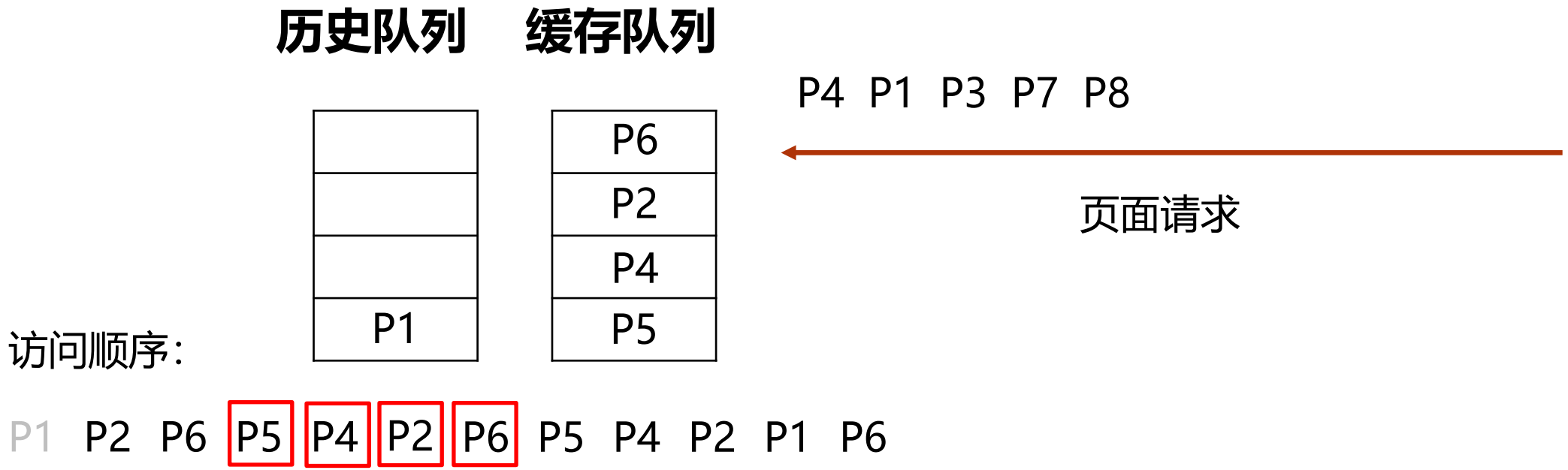
假设K=2，SIZE=4



- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



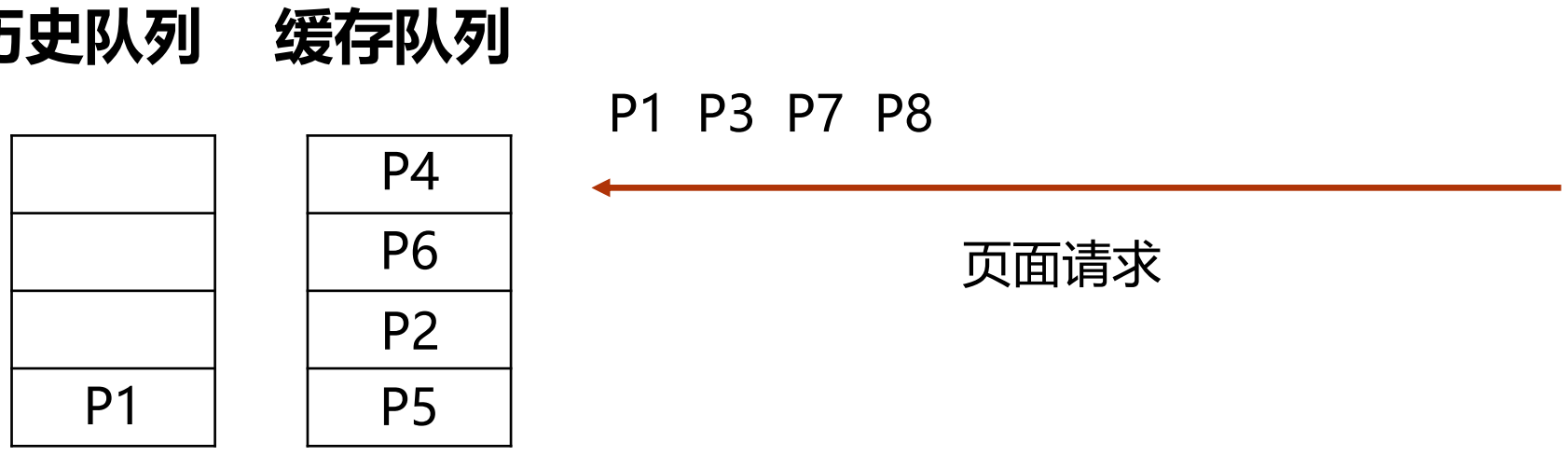
访问顺序：

P1 P2 P6 P5 P4 P2 P6 P5 P4 P2 P1 P6

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



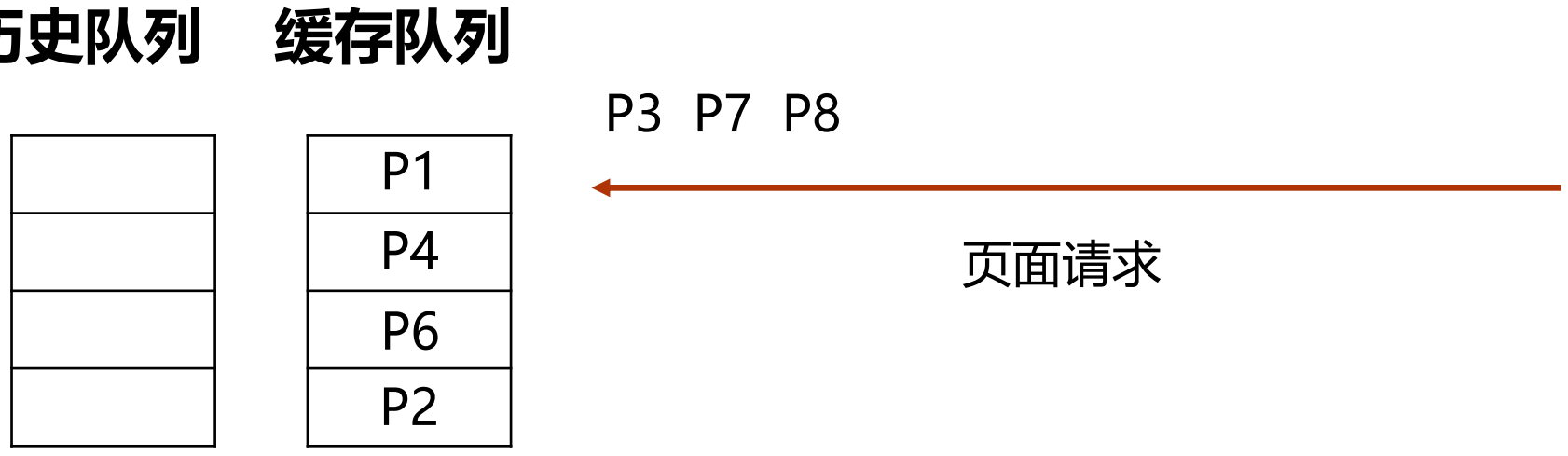
访问顺序：

P1 P2 P6 P5 P4 P2 P6 P5 P4 P2 P1 P6 P4

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

LRU-K算法

假设K=2，SIZE=4



访问顺序：

P1 P2 P6 P5 P4 P2 P6 P5 P4 P2 P1 P6 P4 P1

- 缓存队列中的页面再次被访问时，依倒数第K次访问的时间排序，更新缓存队列中该页面的位置。

查询处理

本节主要内容

1 查询计划

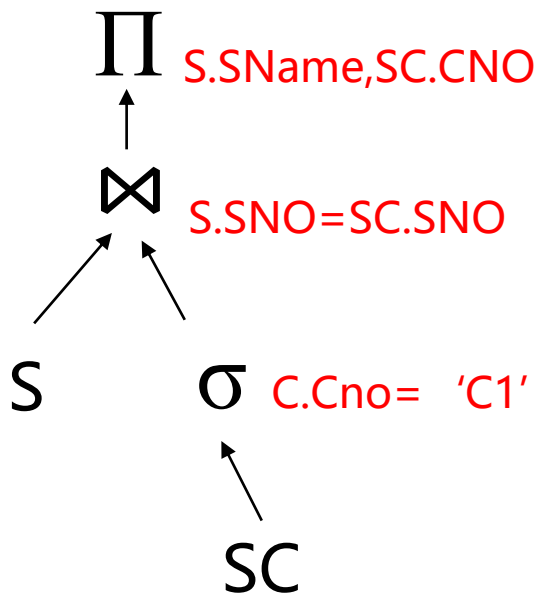
2 查询处理模型

3 数据存取方法

1 查询计划

- 操作算子以树的形式进行组织
- 数据流从叶子结点流向根节点
- 根节点的输出是查询的结果

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```



2 查询处理模型

处理模型: 如何执行一个查询计划

不同的任务负载选择不同的处理模型

- 迭代模型（Iterator Model ）
- 物化模型（Materialization Model）
- 向量/批量模型（Vectorized / Batch Model）

拉取式：迭代/物化/向量模型

- **每个算子都实现3个函数：**
  - **Open()**

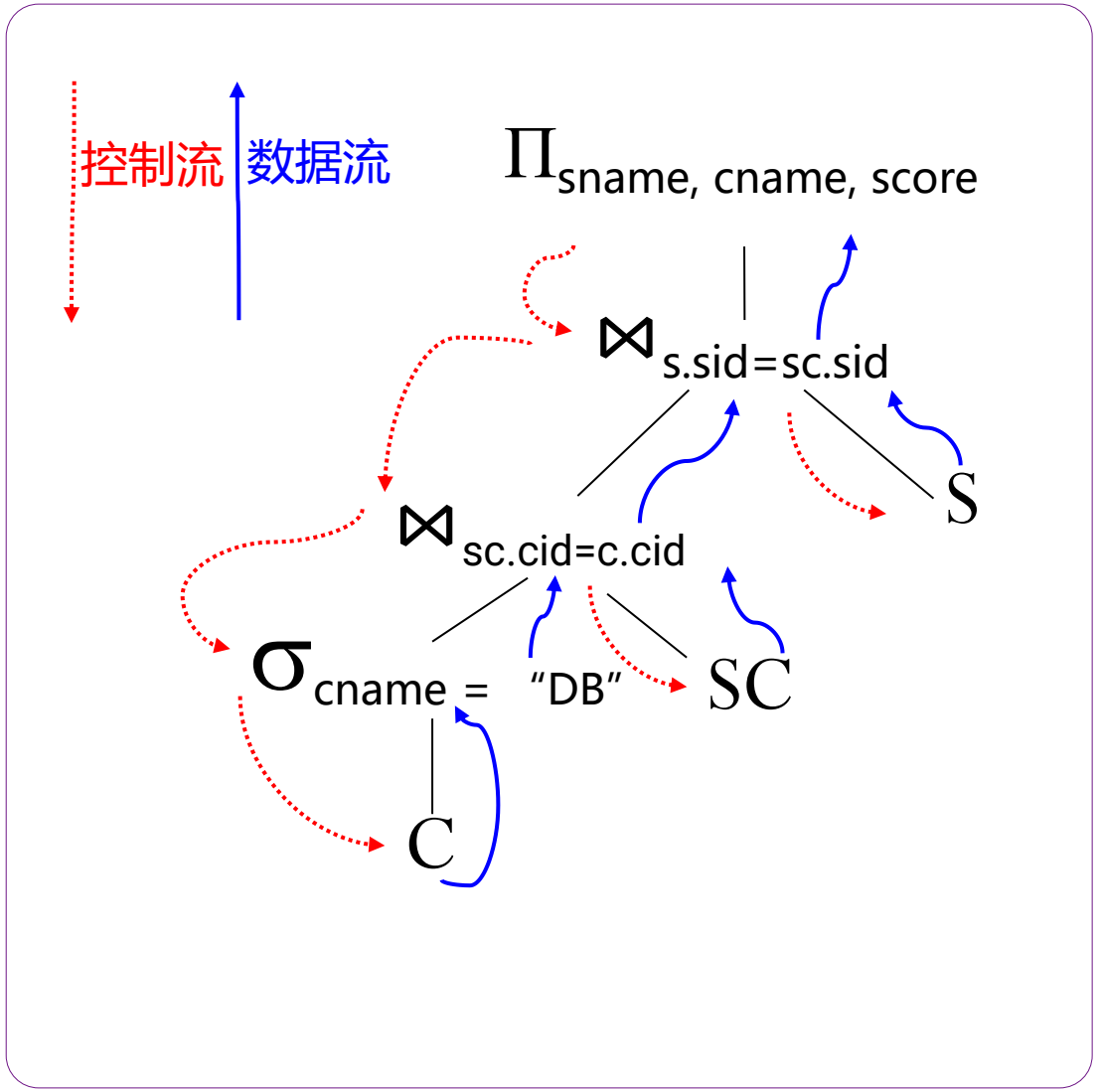
    初始化；申请空间和设置连接、选择条件的参数
  - **Next()**

    获取下一条/一批记录；验证是否满足条件，若满足返回父节点；否则中止
  - **Close()**

    清理
  - 从根结点开始执行，自上而下
  - 调用算子的Next()函数从子结点“拉取”数据

拉取式：迭代/物化/向量模型

- 迭代模型
  - 一次一元组
- 物化模型
  - 一次所有元组
- 向量化模型
  - 一次一批元组

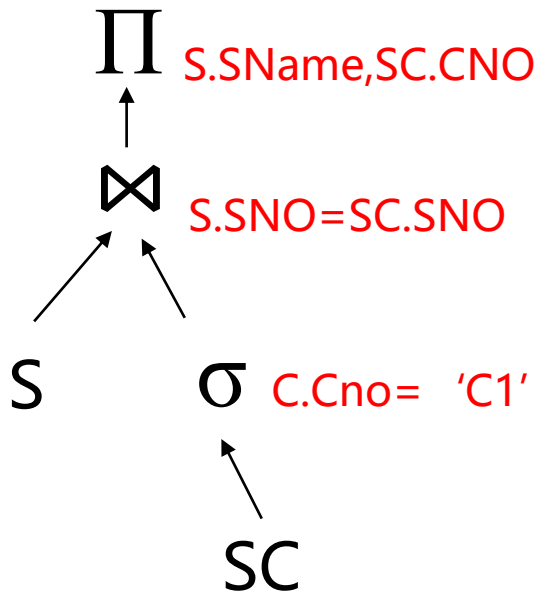


2.1 迭代模型

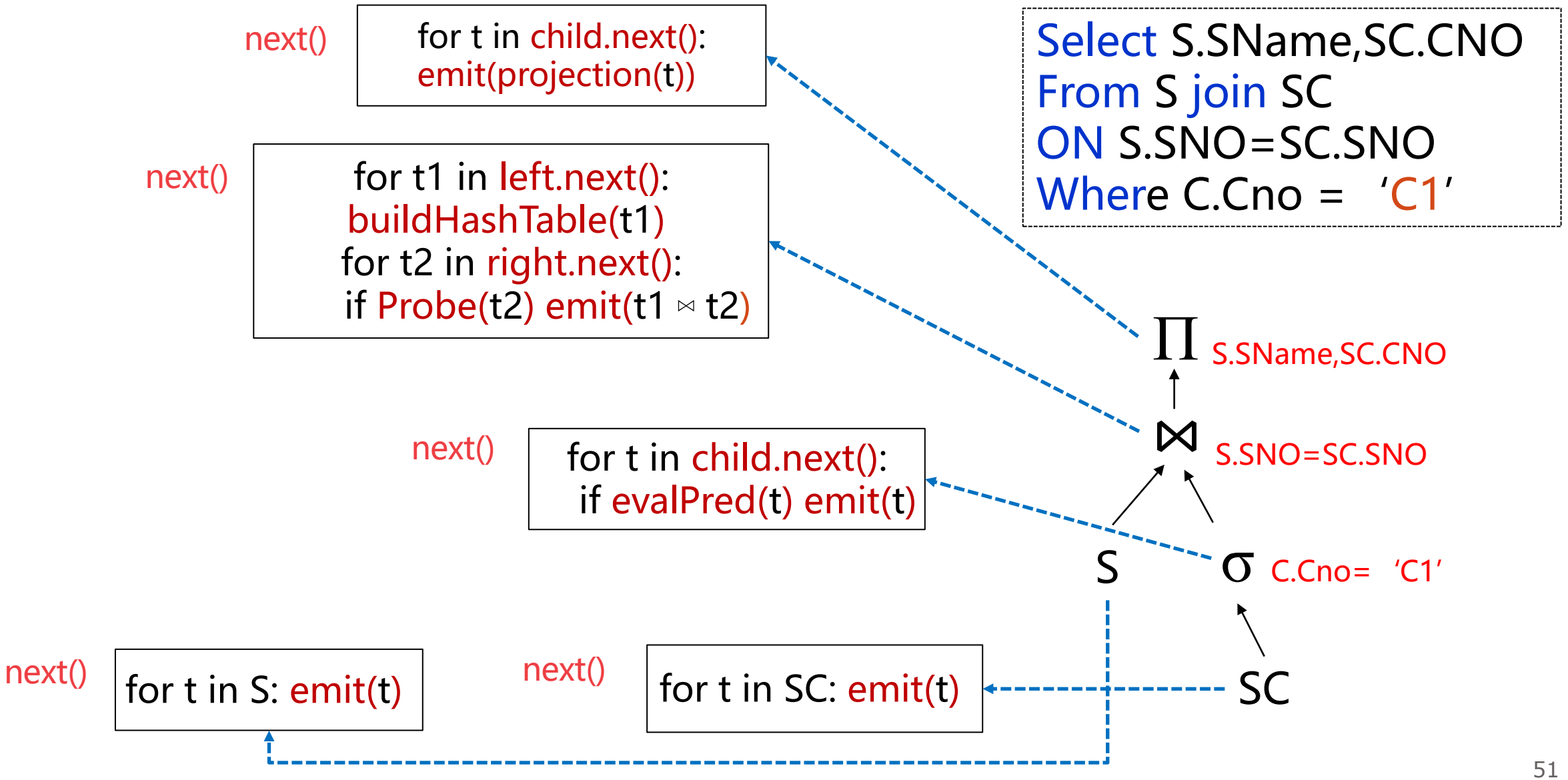
- 也称作火山模型/流水线模型
  (Volcano / Pipeline Model)

- Next(): 返回一个元组，或者一个空值标记
- 循环调用子节点的Next()函数
- 将数据取到内存后一次做足够多的操作，以减少页面替换。

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```



2.1 迭代模型


next()
for t in child.next():
emit(projection(t))
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
next()
for t1 in left.next():
buildHashTable(t1)
for t2 in right.next():
if Probe(t2) emit(t1 ⋈ t2)
Π S.SName,SC.CNO
⋈ S.SNO=SC.SNO
next()
for t in child.next():
if evalPred(t) emit(t)
S
σ C.Cno= 'C1'
SC
next()
for t in S: emit(t)
next()
for t in SC: emit(t)

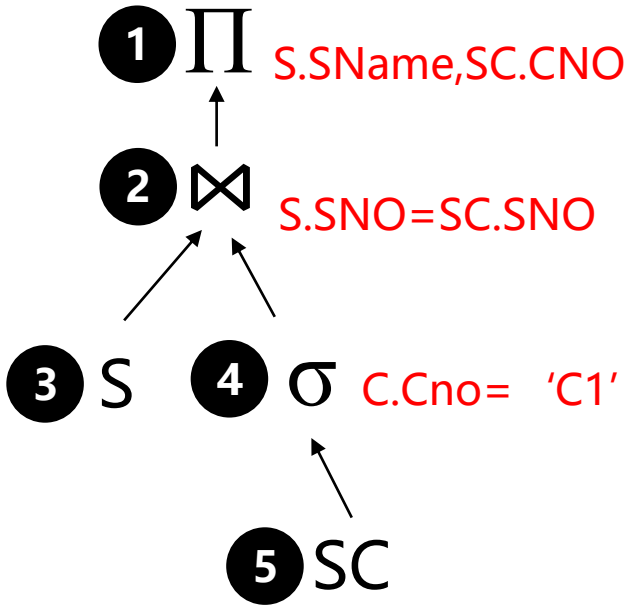
2.1 迭代模型

1
```
for t in child.next():
emit(projection(t))
```

```
for t1 in left.next():
buildHashTable(t1)
for t2 in right.next():
if Probe(t2) emit(t1 ⋈ t2)
```

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```

```
for t in child.next():
 if evalPred(t) emit(t)
```




```
for t in S: emit(t)
```

```
for t in SC: emit(t)
```

2.1 迭代模型

1
```
for t in child.next():
emit(projection(t))
```

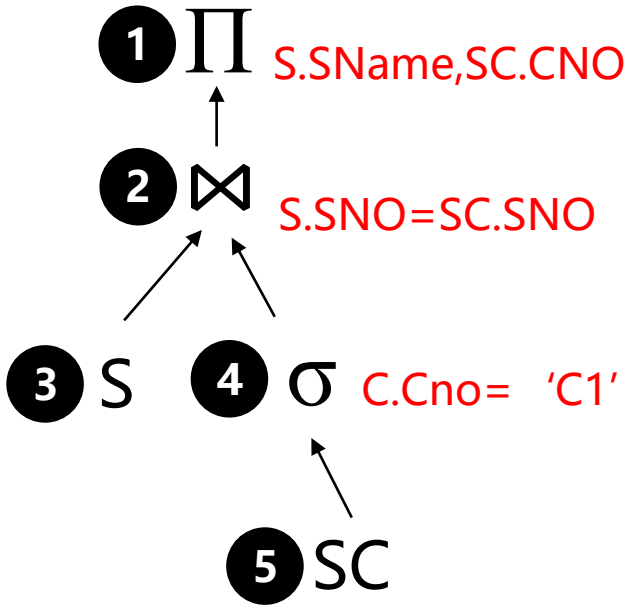
2
```
for t1 in left.next():
buildHashTable(t1)
for t2 in right.next():
if Probe(t2) emit(t1 ⋈ t2)
```

3
```
for t in S: emit(t)
```

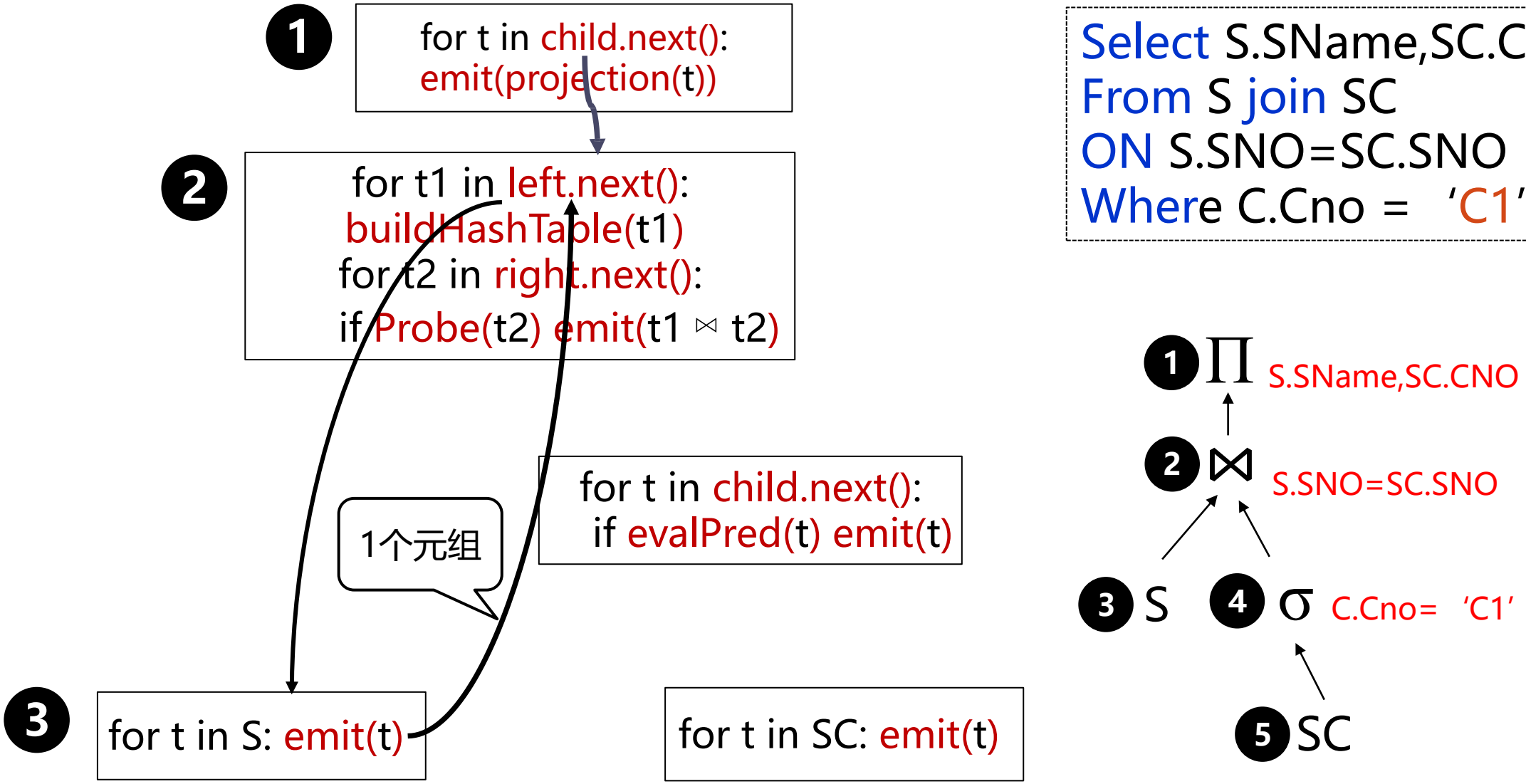
```
for t in child.next():
 if evalPred(t) emit(t)
```

```
for t in SC: emit(t)
```

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno =  'C1'
```

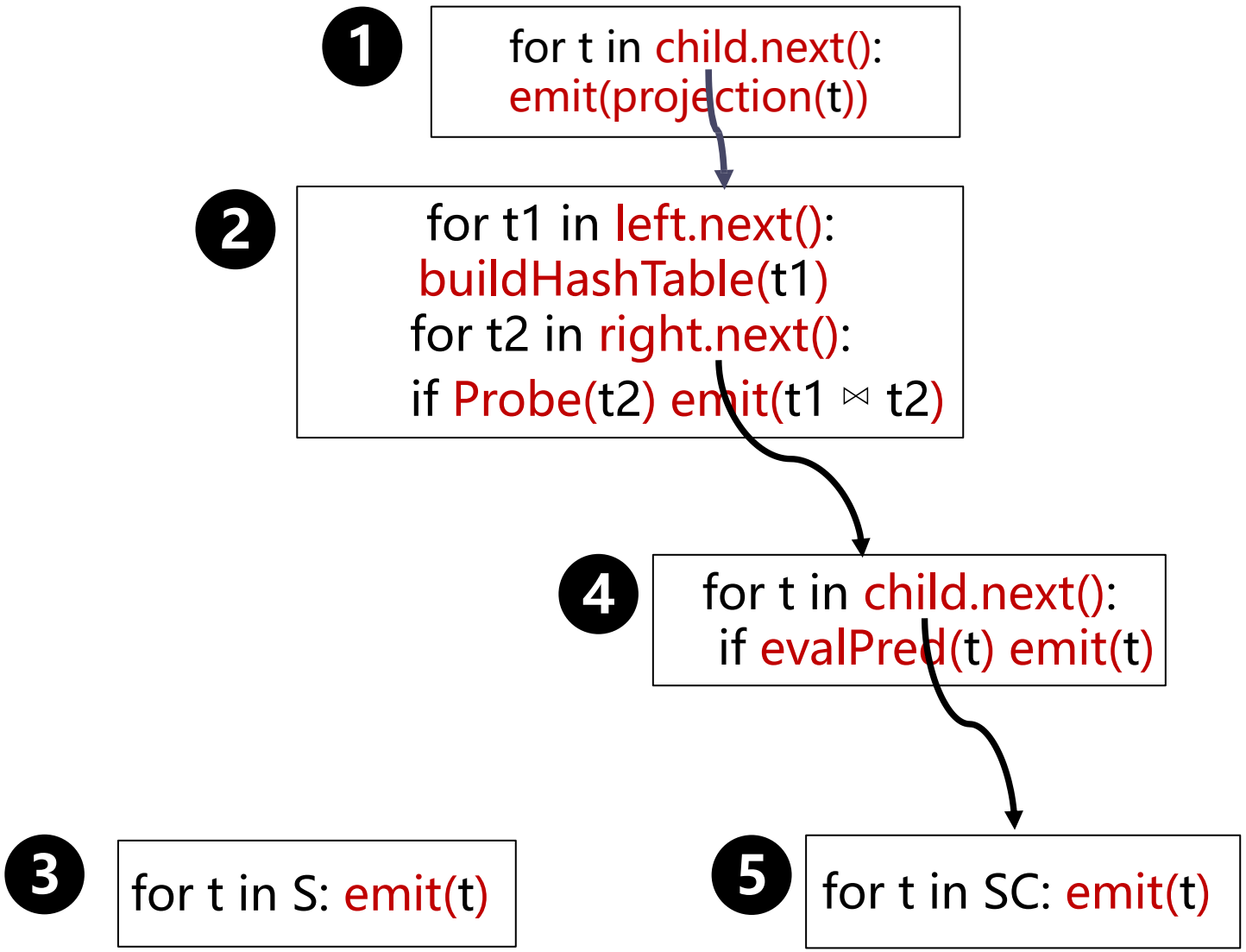


2.1 迭代模型

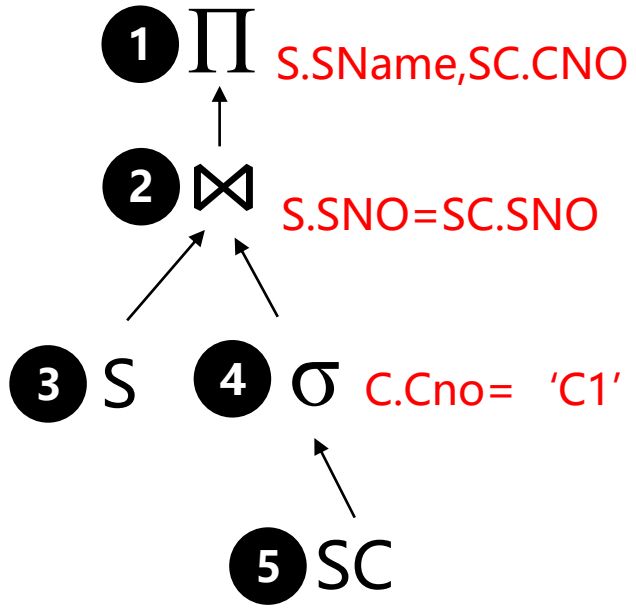



```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno =  'C1'
```

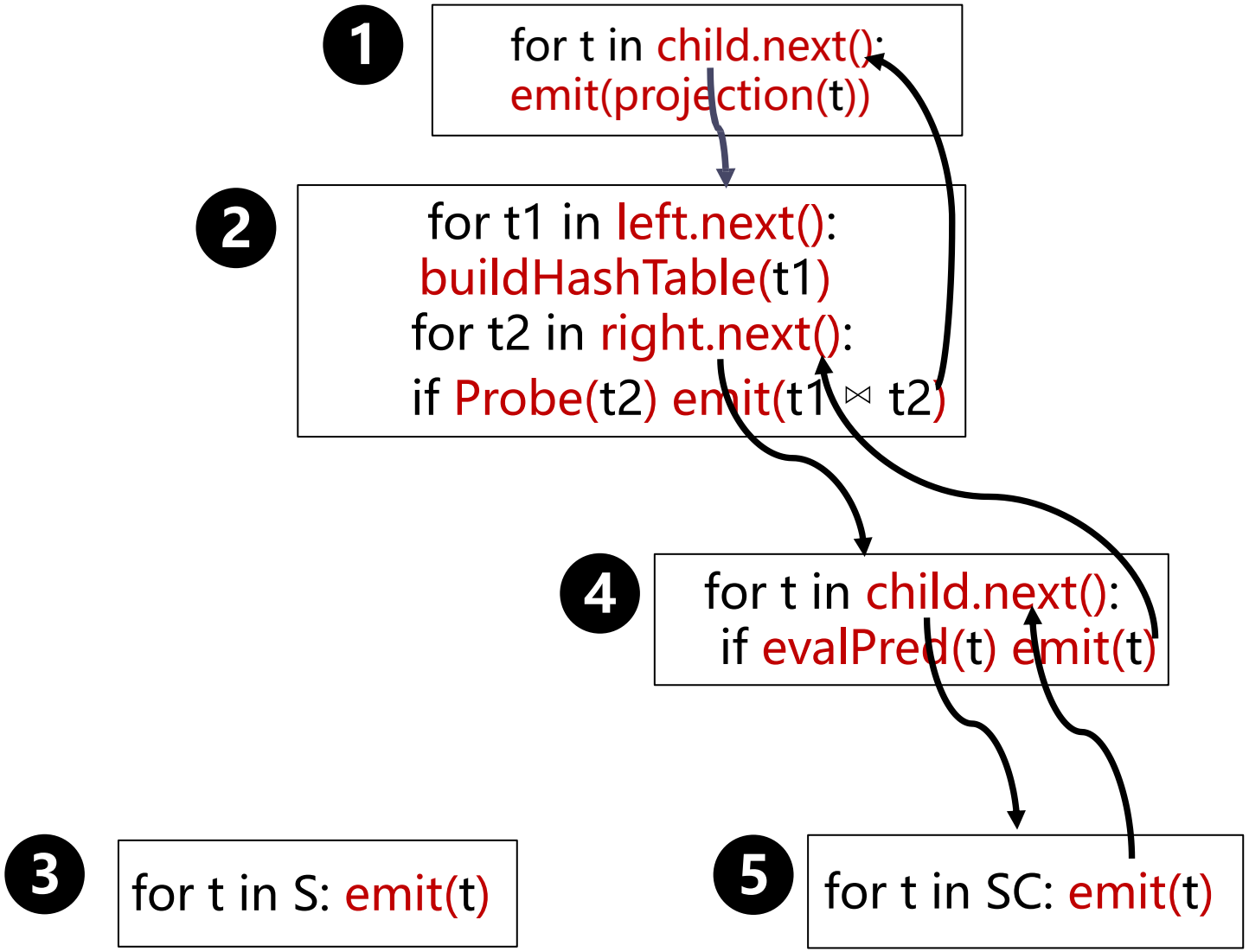
2.1 迭代模型



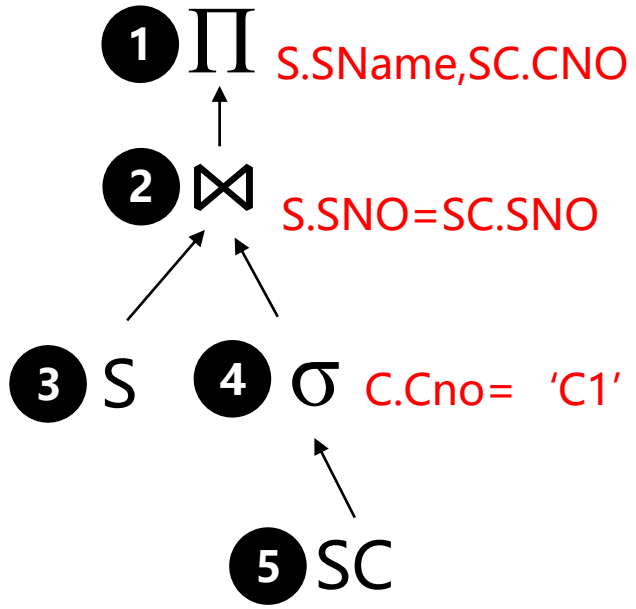

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno =  'C1'
```



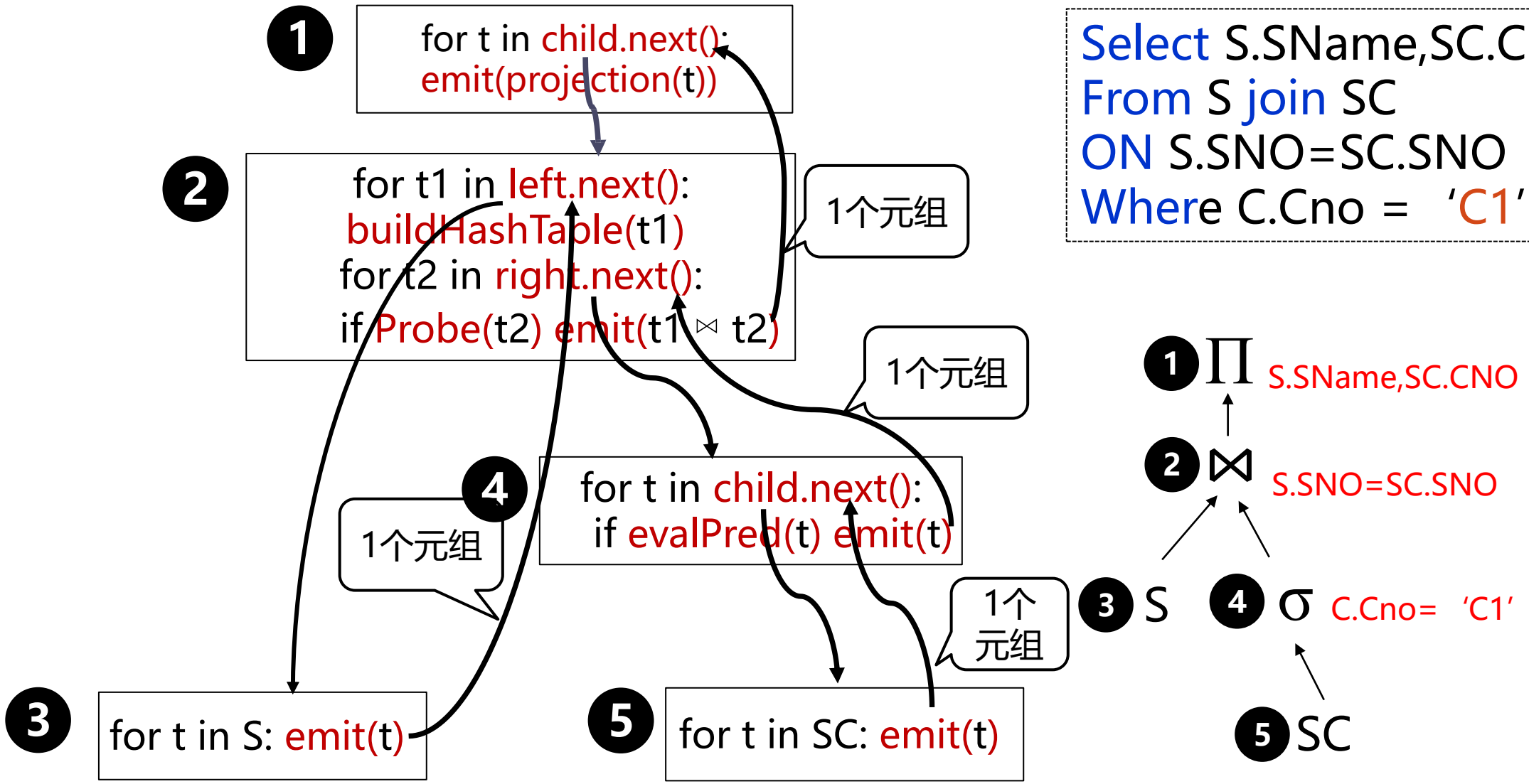
2.1 迭代模型




```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno =  'C1'
```



2.1 迭代模型




```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno =  'C1'
```

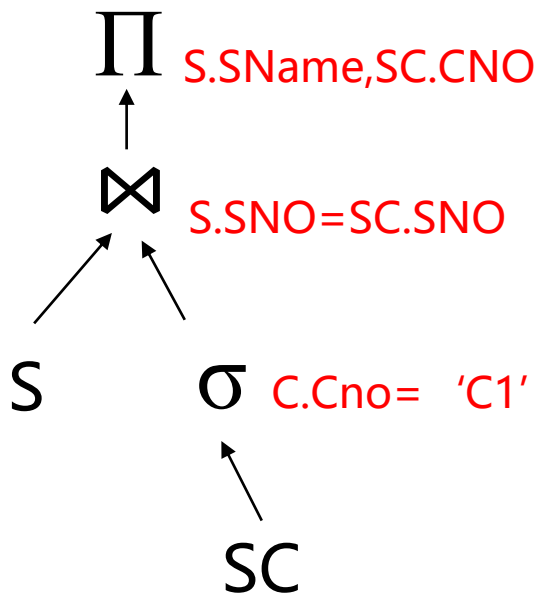
迭代模型的特点

- 几乎所有的DBMS都采用该模型，允许元组流水线。
- 该模型可以很容易的控制输出，如limit操作。
- 有些算子会阻塞数据的流动，直到其子节点发送所有的元组，例如join操作、子查询操作、排序操作。

2.2 物化模型

- 算子一次性获取它所有的输入，当处理完成后再将整体结果返回给它的父节点。
- 一次处理足够多的数据，能传递”hints”来避免扫描过多元组（如limit）。
- 输出可以是整个元组（行存储模式），或者查询所需的属性子集（列存储模式）。

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```



物化模型的例子

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

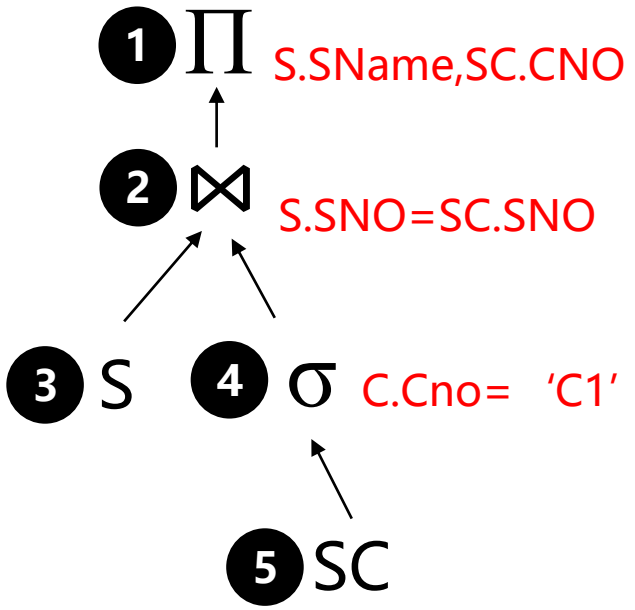
```
out = [ ]
for t1 in left.Output():  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
out = [ ]
for t in S: out.add(t)
return out
```

```
out = [ ]
for t in SC: out.add(t)
return out
```

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```



物化模型的例子

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

```
out = [ ]
for t1 in left.Output():  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

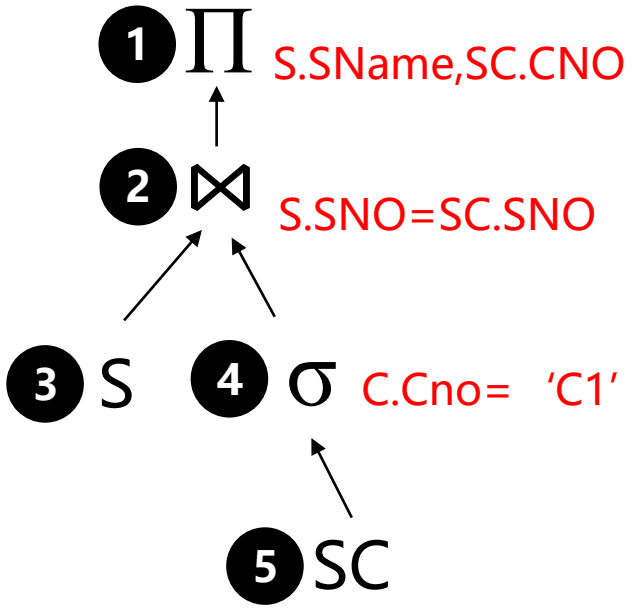
```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

全部元组

```
out = [ ]
for t in S: out.add(t)
return out
```

```
out = [ ]
for t in SC: out.add(t)
return out
```

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```



物化模型的例子

```
1  out = [ ]
   for t in child.Output():
     out.add(projection(t))
   return out
```

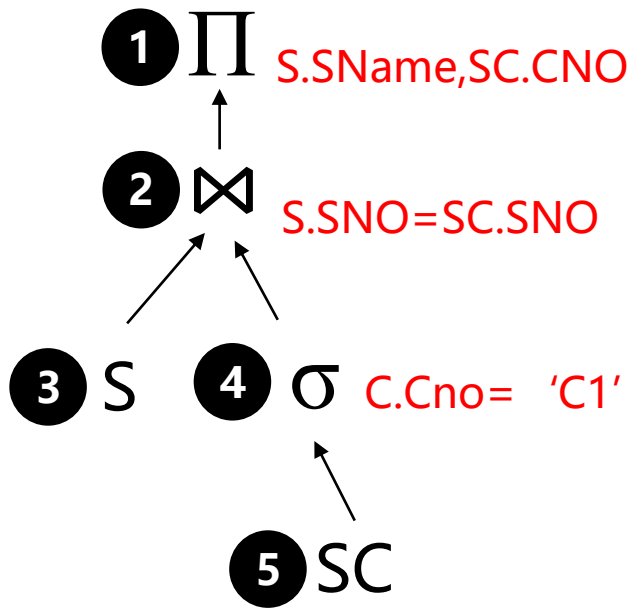
```
2  out = [ ]
   for t1 in left.Output():  buildHashTable(t1)
   for t2 in right.Output():
     if probe(t2): out.add(t1⋈t2)
   return out
```

```
4  out = [ ]
   for t in child.Output():
     if evalPred(t): out.add(t)
   return out
```

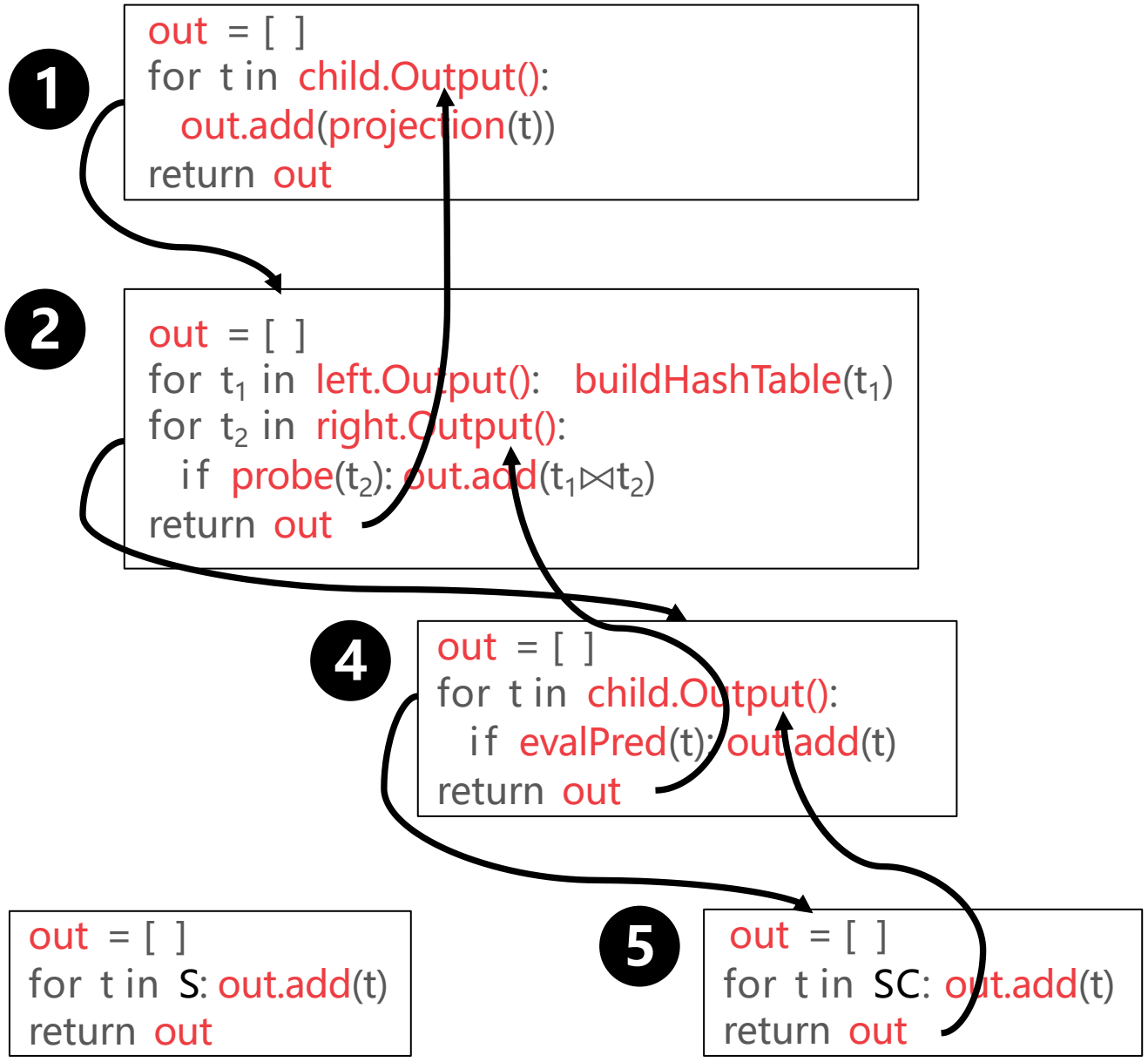
```
3  out = [ ]
   for t in S: out.add(t)
   return out
```

```
5  out = [ ]
   for t in SC: out.add(t)
   return out
```

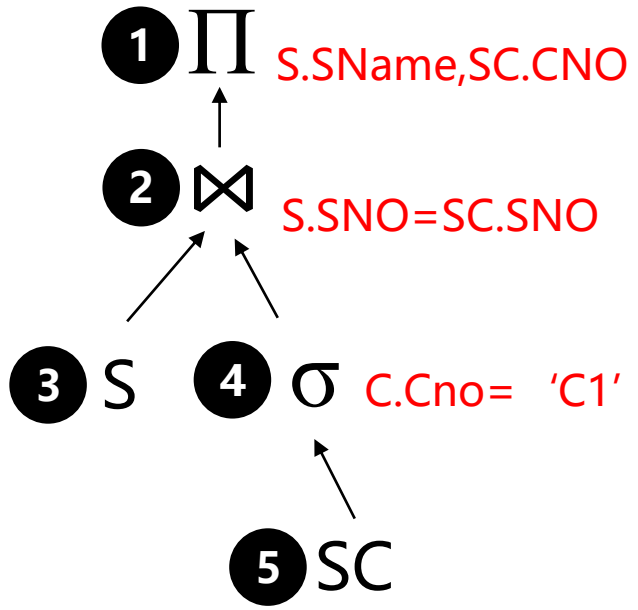
```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```



物化模型的例子




```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```



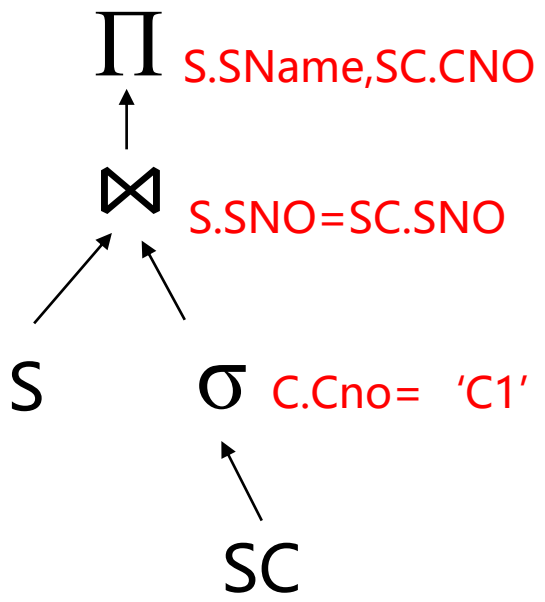
物化模型的特点

- 适合OLTP，一次处理少量数据
- 不适合OLAP，可能查询产生较大的中间结果
- 相对火山模型，较少的函数调用

2.3 向量模型

- 执行框架同火山模型
- 不同之处是每次调用Next函数，返回的是一批（batch）元组而不是一个元组；
- Batch的大小可以预先指定；
- 介于火山模型和物化模型之间；
- 适用于OLAP和OLTP。

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```



向量模型

```
out=[]
for t in child.Next():
    out.add(projection(t))
    if |out|>n: emit(out)
```

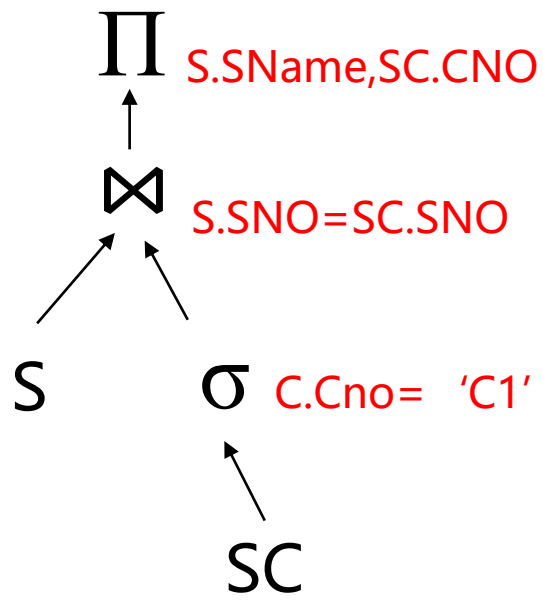
```
out=[]
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
 if Probe(t2) out.add(t1 ⋈ t2)
 if |out|>n: emit(out)
```

```
out=[]
for t in child.next():
 if evalPred(t) out.add(t)
 if |out|>n: emit(out)
```

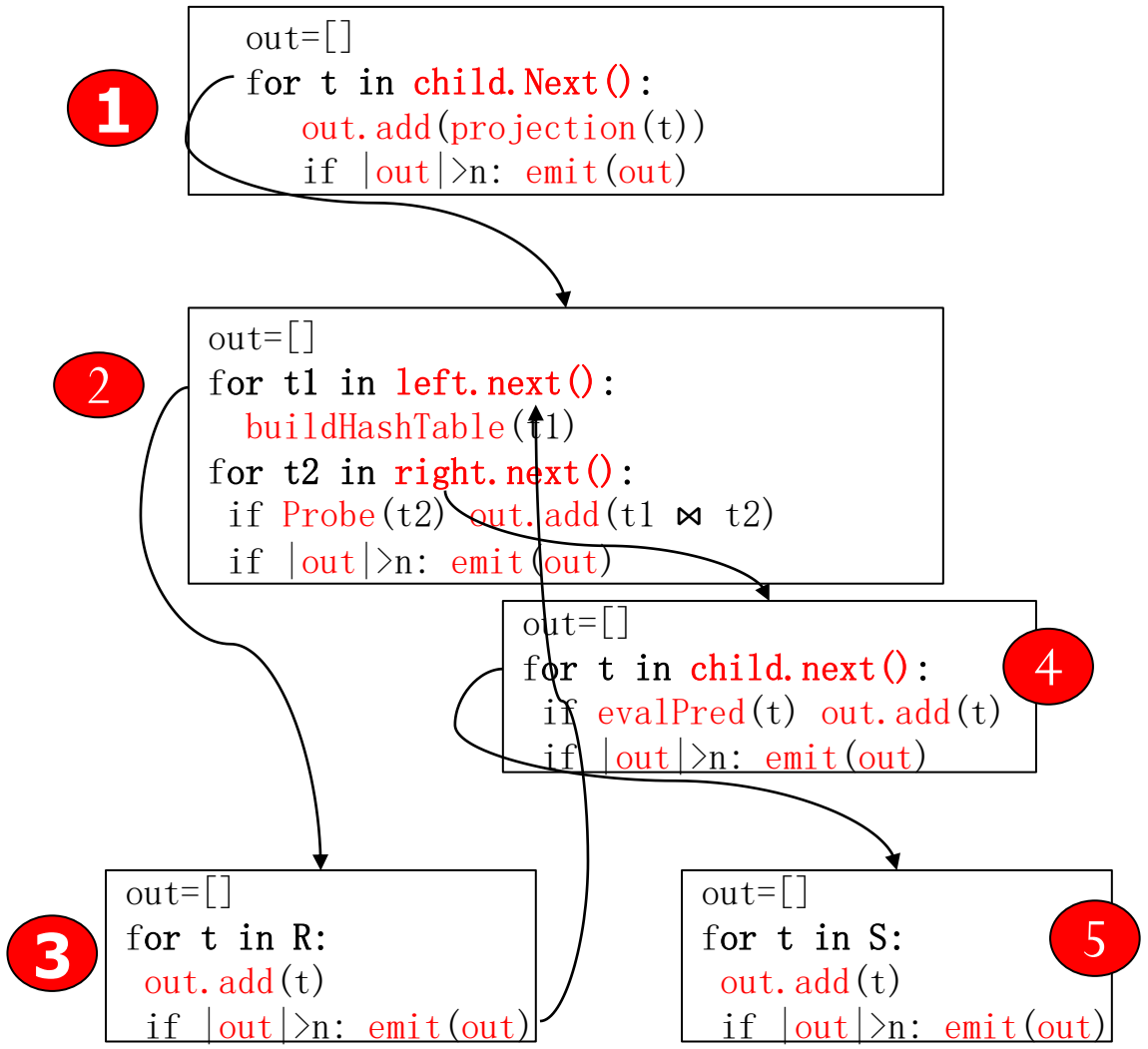
```
out=[]
for t in R:
 out.add(t)
 if |out|>n: emit(out)
```

```
out=[]
for t in S:
 out.add(t)
 if |out|>n: emit(out)
```

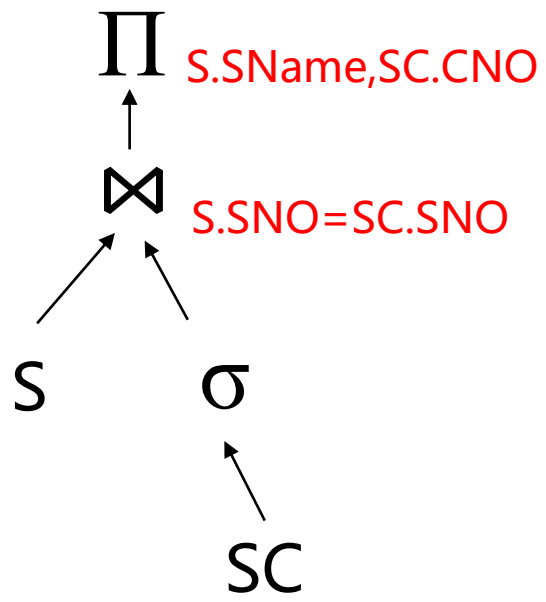
```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = 'C1'
```



向量模型




```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```

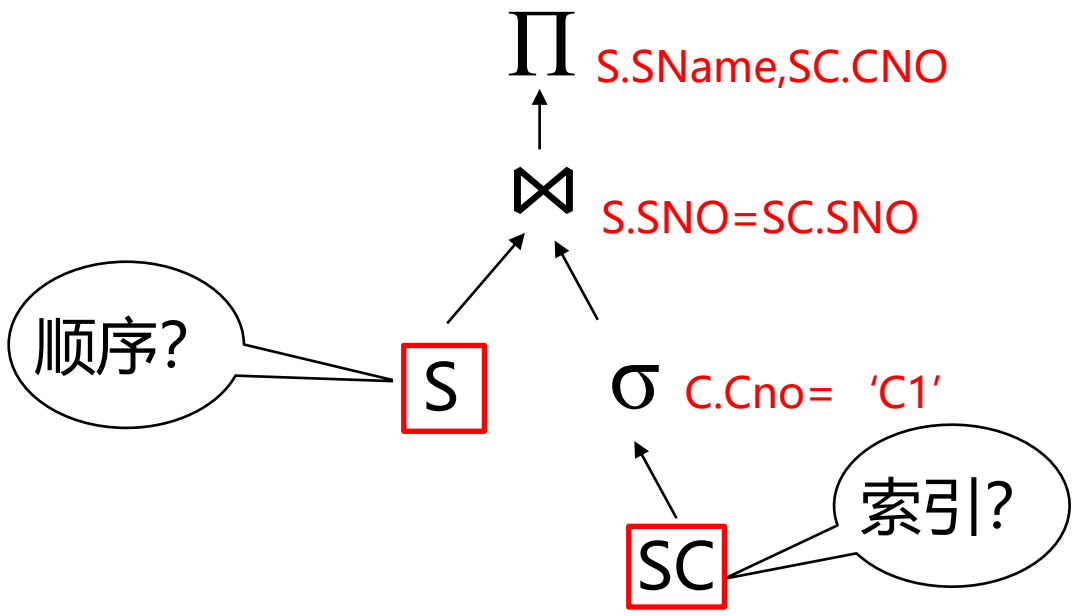


3 数据存取

访问存储在表(table)中的数据有两种基本方法：

- 顺序扫描
- 索引扫描

```
Select S.SName,SC.CNO
From S join SC
ON S.SNO=SC.SNO
Where C.Cno = ‘C1’
```



顺序扫描

对于表的每一个页：

- 加载到BufferPool中
- 对BufferPool的页中符合条件的元组依次进行处理
- DBMS内部维持一个游标指向上次访问的Page/Slot

```
For Page in Table.Pages:
  For t in Page.tuples:
    If evalPred(t):
      Do Somthing
```

索引扫描

基本思想:

- 对于索引的每一个叶结点
  - 如果叶结点的Key值符合条件
  - 根据叶结点的value将key所在的页提取进buffer pool
- 单索引和多索引扫描

```
For   LeafNode  in Nodes:
  For k in LeafNode.keys:
    If evalPred(k):
      Fetch Page
```