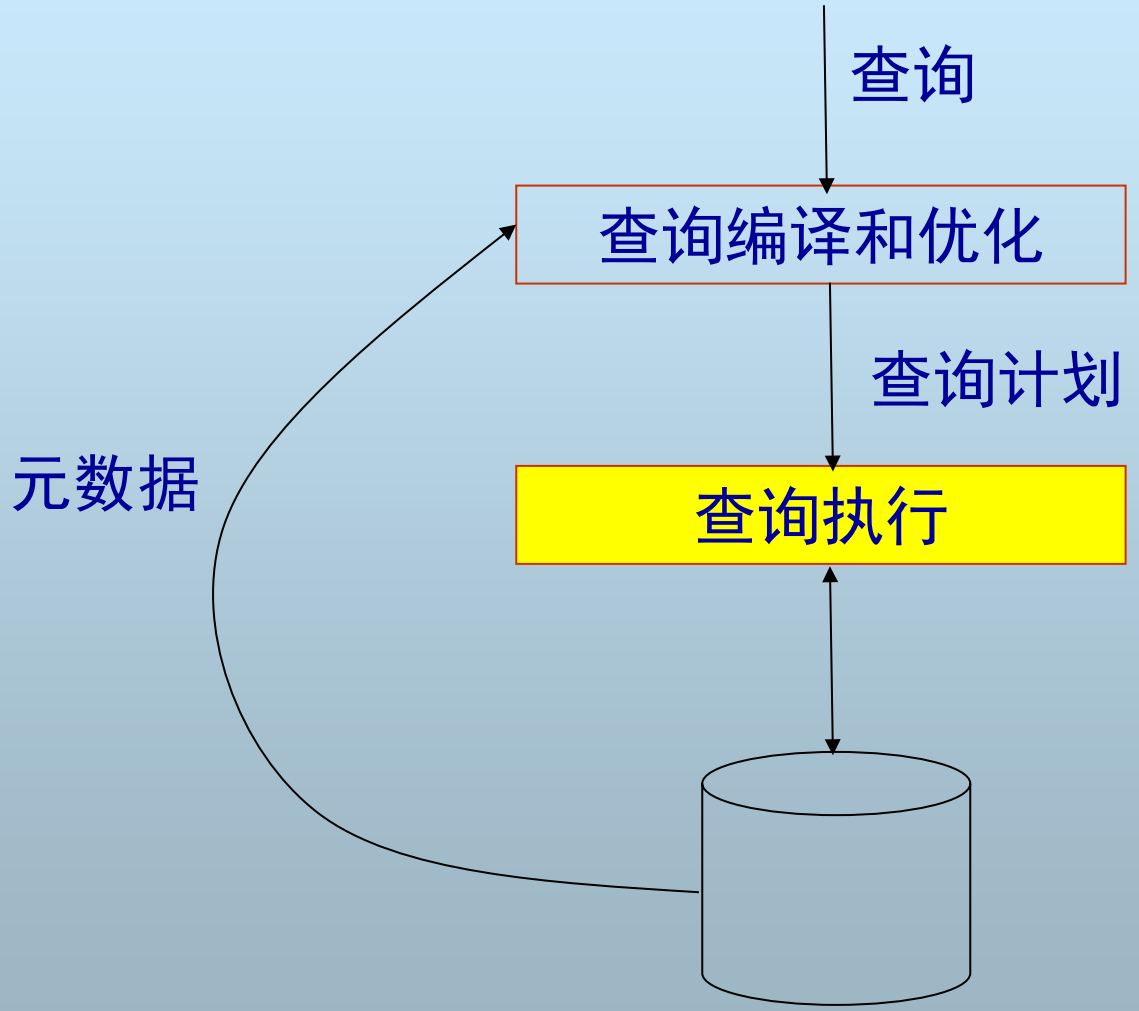
Join Algorithms

金培权（jpq@ustc.edu.cn）

查询处理概述



主要内容

- 物理查询计划操作符
- 连接操作的实现算法
  - 嵌套循环连接
  - 归并连接
  - 索引连接
  - 散列连接
- 连接算法的I/O代价估计
- 连接顺序选择

一、物理查询计划操作符

- 逻辑操作符的物理操作符
  - 逻辑操作符的特定实现
- 其它物理操作符
  - 表扫描：TableScan
  - 排序扫描：SortScan
  - 索引扫描：IndexScan

一、物理查询计划操作符

- 物理操作符的执行算法
  - 一趟算法
  - 两趟算法
  - 多趟算法

  按数据的读取方式

  - 基于排序的算法
  - 基于散列的算法
  - 基于索引的算法

  按所基于的底层算法

二、连接操作(Join)的实现算法

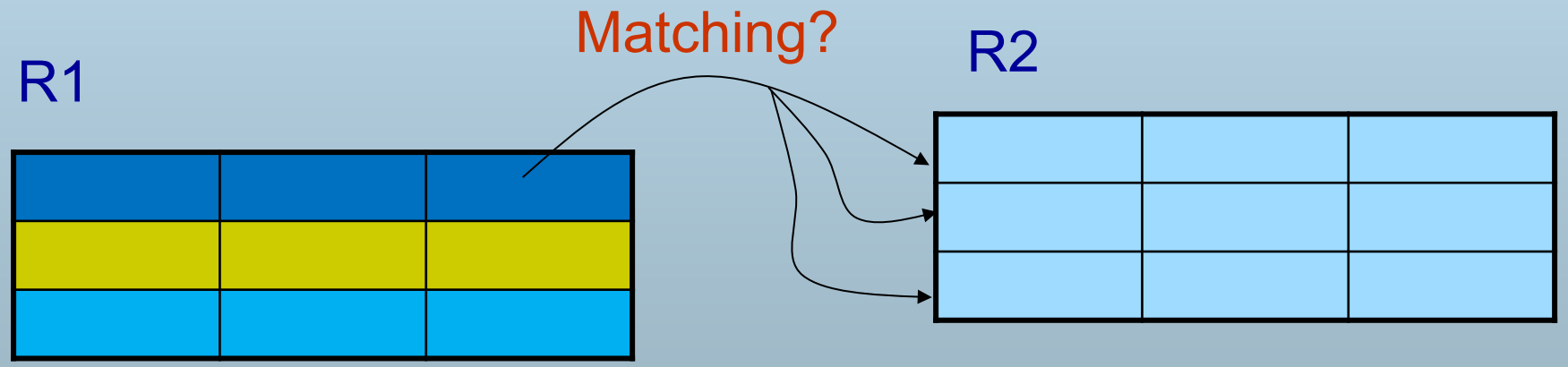
- R1(A,C)⋈R2(C,D)
  - 嵌套循环连接
    (Nested loops join or Iteration join)
  - 归并连接 (Merge join or Sort join)
  - 索引连接 (Index join)
  - 散列连接 (Hash join)

1、嵌套循环连接

For each r ∈ R1 Do

For each s ∈ R2 do

If r.C = s.C Then output r, s pair



2、归并连接

```
(1) if R1 and R2 not sorted, sort them
(2) i ← 1; j ← 1;
    While (i ≤ T(R1)) ∧ (j ≤ T(R2)) do  {
        if R1[ i ].C = R2[ j ].C then
                OutputTuples;
        else if R1[ i ].C > R2[ j ].C then
                j ← j+1;
        else if R1[ i ].C < R2[ j ].C then
                i ← i+1;
    }
```

2、归并连接

```
Procedure OutputTuples
  While (R1[ i ].C = R2[ j ].C) ∧ (i ≤ T(R1)) do {
      jj ← j;
      while (R1[ i ].C = R2[ jj ].C) ∧ (jj ≤ T(R2)) do {
          output pair R1[ i ], R2[ jj ];
          jj ← jj+1;
      }
      i ← i+1;
  }
```

2、归并连接

Example

i	R1[i].C	R2[j].C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

3、索引连接

```
For each r ∈ R1 do {
    X ← index (R2, C, r.C)
    For each s ∈ X do
        Output r,s pair
}
```

Assume R2.C index

Note: X ← index(rel, attr, value)

then X = set of rel tuples with attr = value

4、散列连接

- C上的散列函数 h, range 0 → k
- Buckets for R1: G0, G1, ... Gk
- Buckets for R2: H0, H1, ... Hk

Algorithm

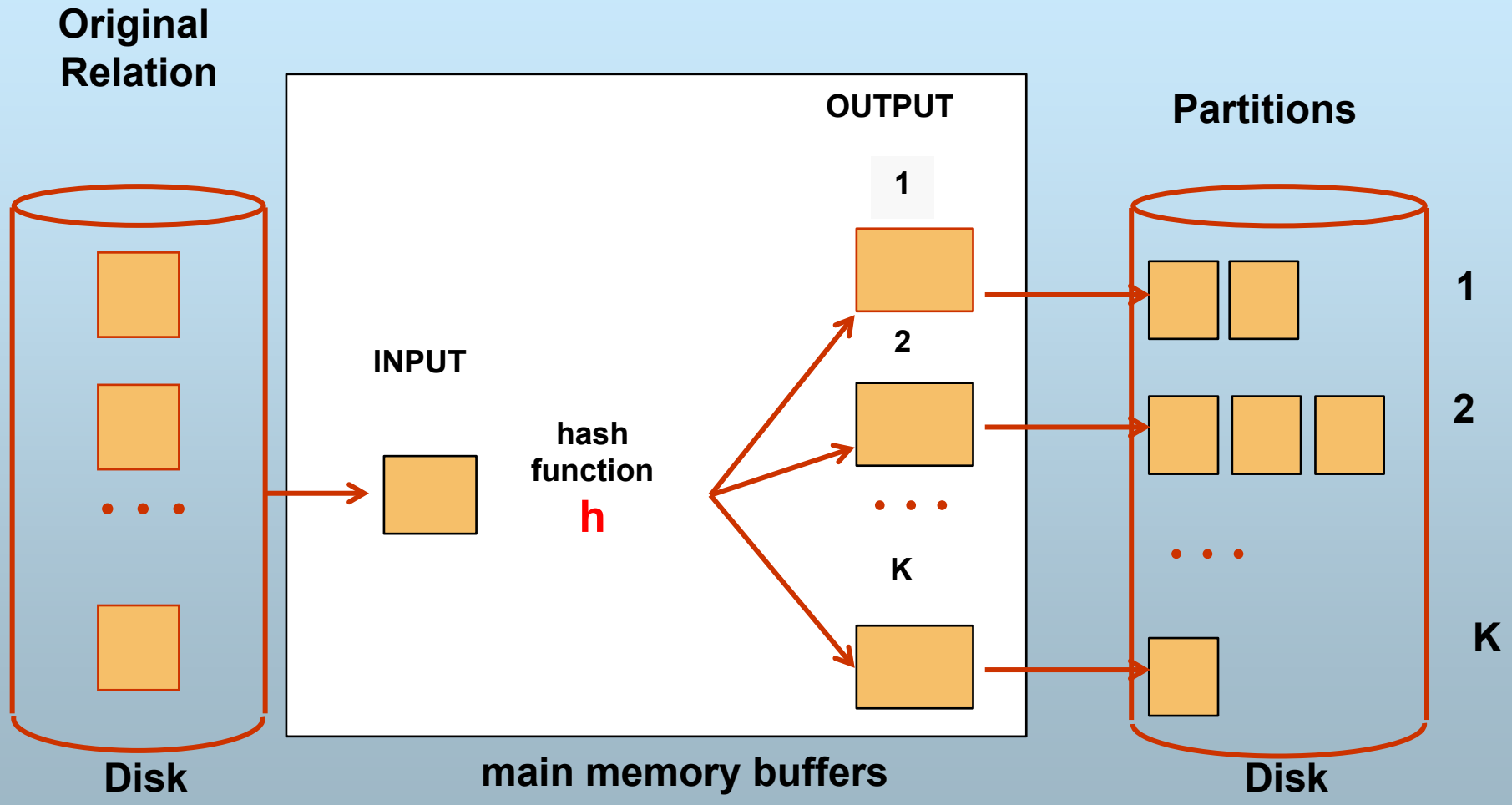
(1) Hash R1 tuples into G buckets

(2) Hash R2 tuples into H buckets

(3) For i = 0 to k do

match tuples in Gi, Hi buckets

4、散列连接


Original
Relation
OUTPUT
Partitions
1
INPUT
2
hash
function
h
K
1
2
K
Disk
main memory buffers
Disk

4、散列连接

Simple example　　hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R1	R2
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

三、连接算法的代价分析

- 影响连接算法代价(I/O)的因素
  - 关系的元组是否在磁盘块中连续存放？(contiguous?)
  - 关系是否按连接属性有序？ (ordered?)
  - 连接属性上是否存在索引？ (indexed?)

1、嵌套循环连接代价分析

- Case1：not contiguous
- Case2：contiguous

1、嵌套循环连接代价分析

Example 1：**not contiguous**

- 设 **T(R1) = 10,000　　T(R2) = 5,000**

  S(R1) = S(R2) =1/10 block --元组大小

  MEM = 101 blocks

Cost: For each R1 tuple:

[Read tuple + Read R2]

Total =10,000 [1+5000]=50,010,000 IOs

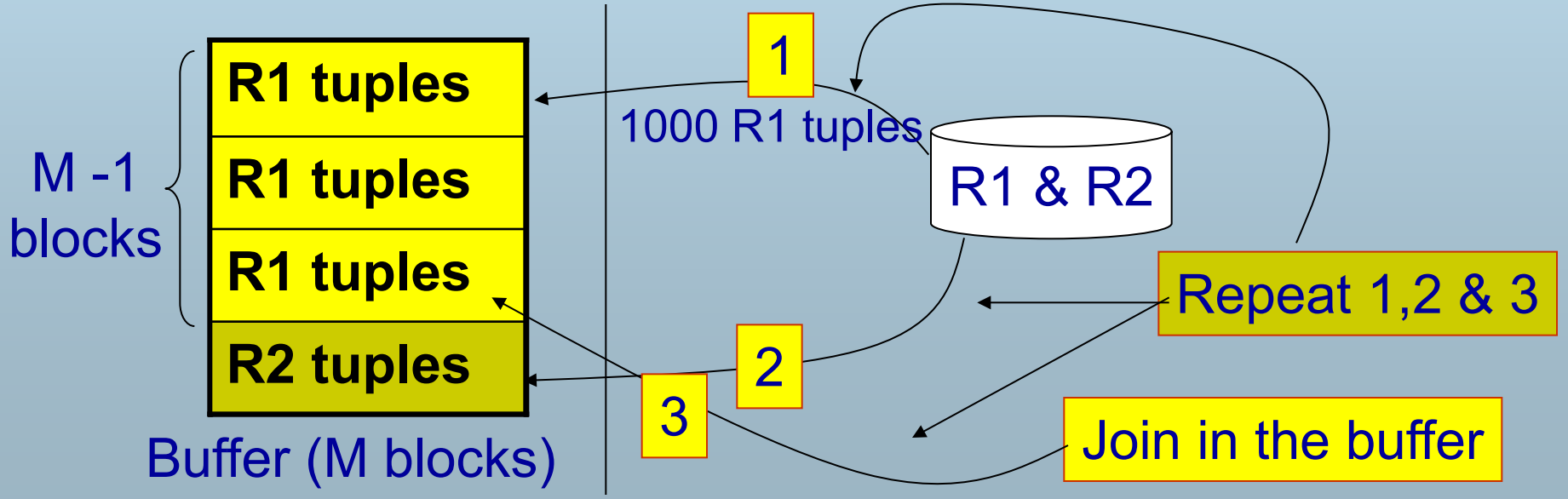
1、嵌套循环连接代价分析

改进的执行策略

(1) Read 100 blocks of R1

(2) Read all of R2 (using 1 block) + join

(3) Repeat until done



1、嵌套循环连接代价分析

改进的执行策略

Cost: For each loop:

Read R1：1000 IOs (1000 tuples)

Read R2：5000 IOs (5000 tuples)

Total：6000 IOs

$$\text{Total} = \frac{10{,}000}{1{,}000} \times 6000 = 60{,}000 \text{ IOs}$$

Better than previous one!

1、嵌套循环连接代价分析

- **Can we further improve it?**

Reverse Join order! Since R1 ⋈ R2 ⇔ R2 ⋈ R1

(1) Read 100 blocks of R2

(2) Read all of R1 (using 1 block) + join

(3) Repeat until done

$$\text{Total} = \frac{5000}{1000} \times (1000 + 10{,}000)$$

$$= 5 \times 11{,}000 = 55{,}000 \text{ IOs}$$

much better!

1、嵌套循环连接代价分析

Example 2：**contiguous**

R2 ⋈ R1

Cost

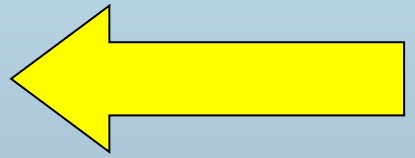
For each loop:

Read R2:	100 IOs
Read R1:	1000 IOs
	1,100

Total= 5 loops x 1,100 = 5,500 IOs

Where are we?

- 物理查询计划操作符
- 连接操作的实现算法
- 连接算法的I/O代价估计
  - 嵌套循环连接代价分析
  - 归并连接代价分析
  - 索引连接代价分析
  - 散列连接代价分析



2、归并连接代价分析

沿用前面的例子

- T(R1) = 10,000　　T(R2) = 5,000

  S(R1) = S(R2) =1/10 block --元组大小

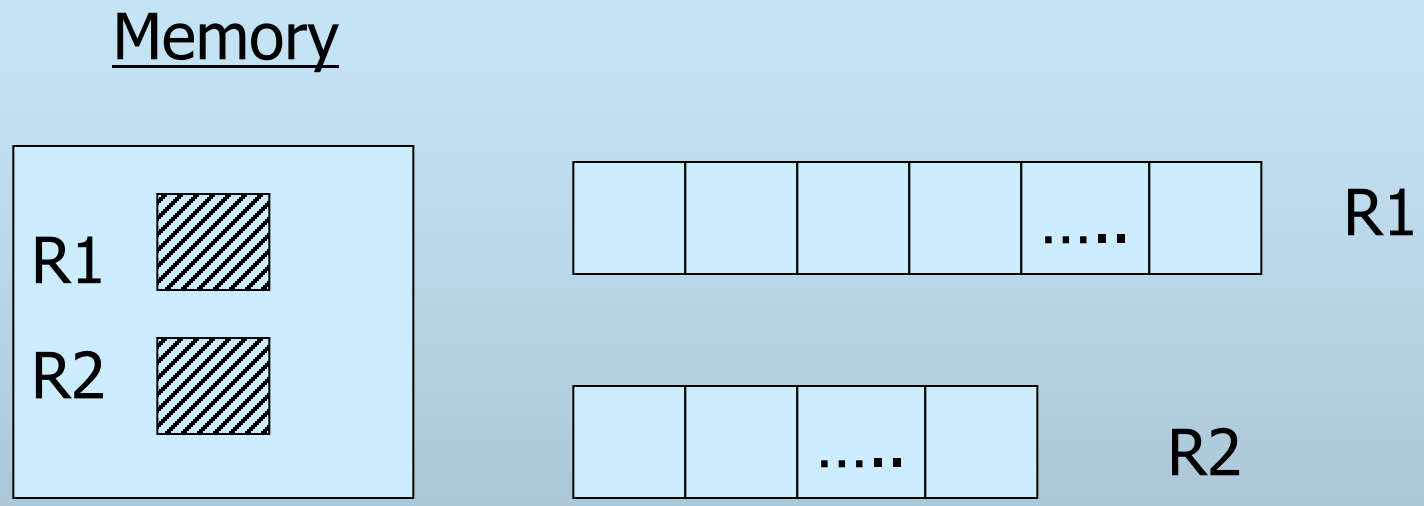
  MEM=101 blocks

2、归并连接代价分析

- **Still need to consider**
  - **Contiguous?**
  - **Ordered?**

2、归并连接代价分析

Example 3：**contiguous and ordered**



Total cost: Read R1 cost + read R2 cost

= 1000 + 500 = 1,500 IOs

2、归并连接代价分析

Example 4：**contiguous but not ordered**

- **Need to sort R1 and R2 first**

2、归并连接代价分析

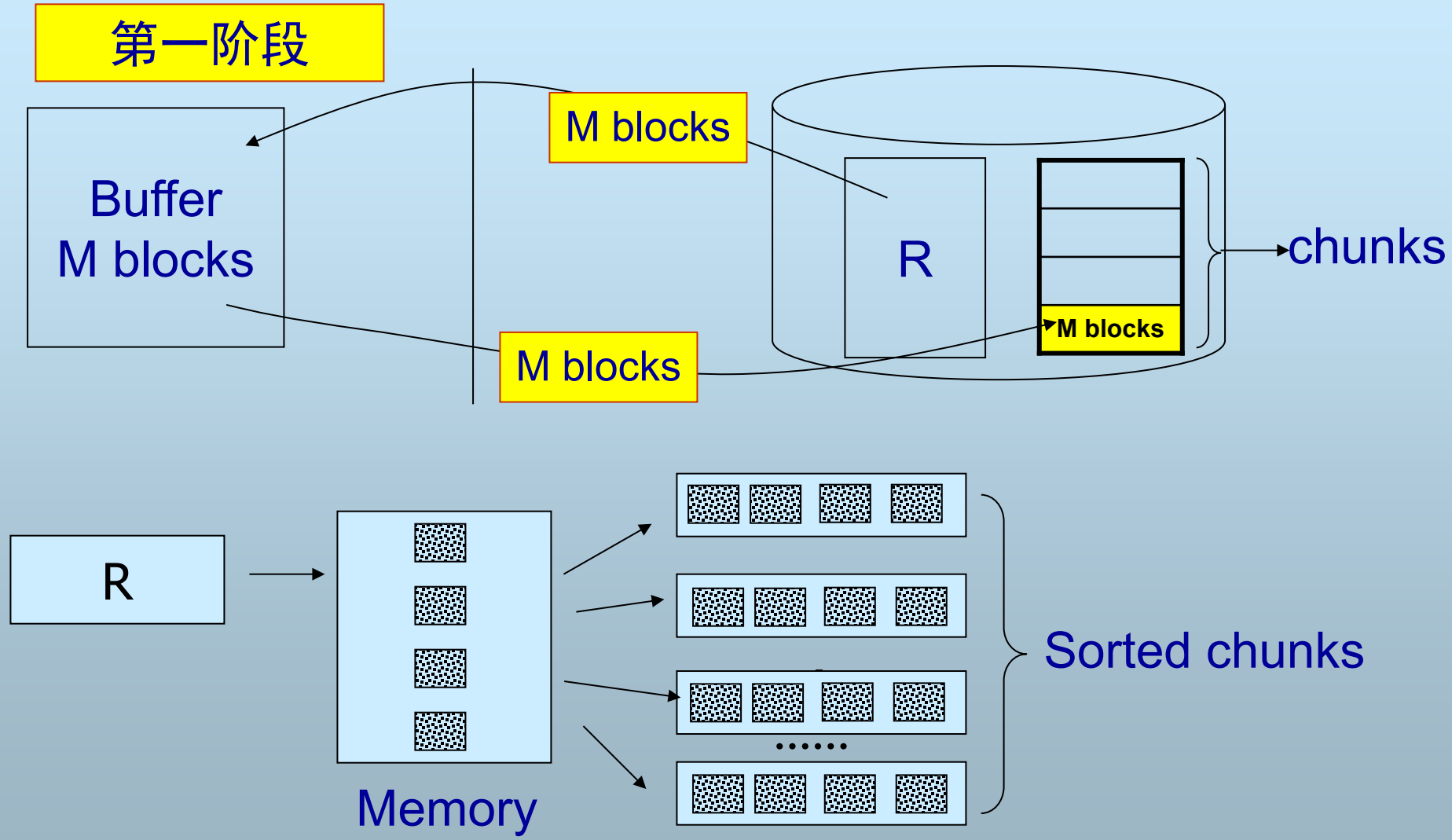
Example 4：**contiguous but not ordered**

一种排序方法：两阶段多路归并排序 (Merge Sort)

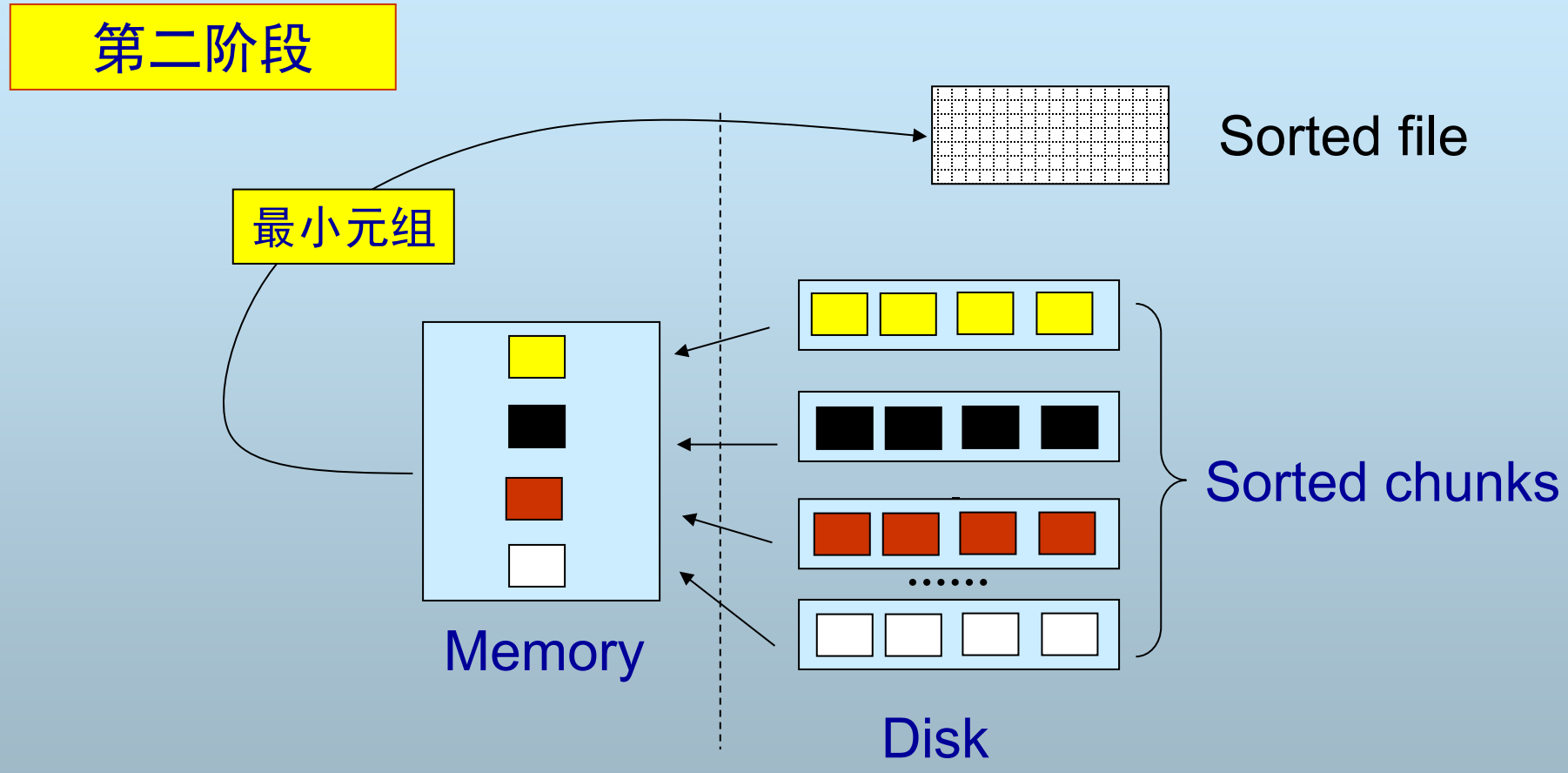
(i) For each 100 blocks of R:
- Read into memory
- Sort in memory
- Write to disk as a chunk

(ii) Read all chunks + merge + write out

2、归并连接代价分析


第一阶段
Buffer
M blocks
M blocks
M blocks
R
M blocks
chunks
R
Memory
Sorted chunks

2、归并连接代价分析



2、归并连接代价分析

Cost: Sort

Each tuple is read, written (first phase)
read, written (second phase)
So each tuple costs 4 IOs.

Sort cost R1: 4 x 1,000 = 4,000
Sort cost R2: 4 x 500 = 2,000

Total: 6,000 IOs

2、归并连接代价分析

Example 4：**contiguous but not ordered**

Cost: Merge join

Sort cost: 6,000
Join cost: 1,500
Total: 7,500 IOs = 5* 1,500 // 每个元组5次IO

But nested loop join only costs 5,500
So merge join does not pay off.

2、归并连接代价分析

Example 4：**contiguous but not ordered**

But if R1 = 10,000 blocks
R2 = 5,000 blocks

Iterate: $\frac{5000}{100}$ x (100+10,000) = 50 x 10,100
= 505,000 IOs

Merge join: 5*(10,000+5,000) = 75,000 IOs

Merge Join (with sort) WINS!

2、归并连接代价分析

Example 4：**contiguous but not ordered**

- **Cost : Nested loop join vs. Merge Join**
  - **Nested loop join**

$$Cost = \frac{B(R2)}{M-1}(M-1+B(R1)) = B(R2) + \frac{B(R1)B(R2)}{M-1}$$

  - **Merge Join**

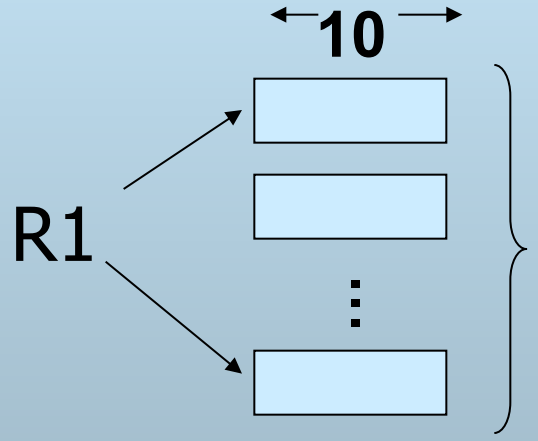
$$Cost = 5(B(R1)+B(R2))$$

嵌套循环连接是固有的二次算法，而归并连接是一次算法，当关系较小时，嵌套循环连接可能优于归并连接，但当关系较大时，归并连接更优。

2、归并连接代价分析

- 两阶段多路归并排序对Memory的要求

E.g: B(R1)=1000 and M=10



100 chunks ⇒ to merge, need
100 memory blocks!

2、归并连接代价分析

- 两阶段多路归并排序对Memory的要求

Say M=k, B(R)=x
chunks = (x/k) size of chunk = k

chunks 不能大于可用的Buffer block数

so... $(x/k) \leq k$

or $k^2 \geq x$ or $k \geq \sqrt{x}$

Buffer block数的平方必须大于等于排序关系R的块数B(R)

2、归并连接代价分析

- 在前面的例子中

$$R1 \text{ is } 1000 \text{ blocks}, \quad k \geq 31.62$$
$$R2 \text{ is } 500 \text{ blocks}, \quad k \geq 22.36$$

- 至少需有32个Buffer blocks才能执行归并连接

2、归并连接代价分析

- 归并连接算法改进 (for contiguous but not ordered)
  - 将第二阶段的排序和 join 合并进行

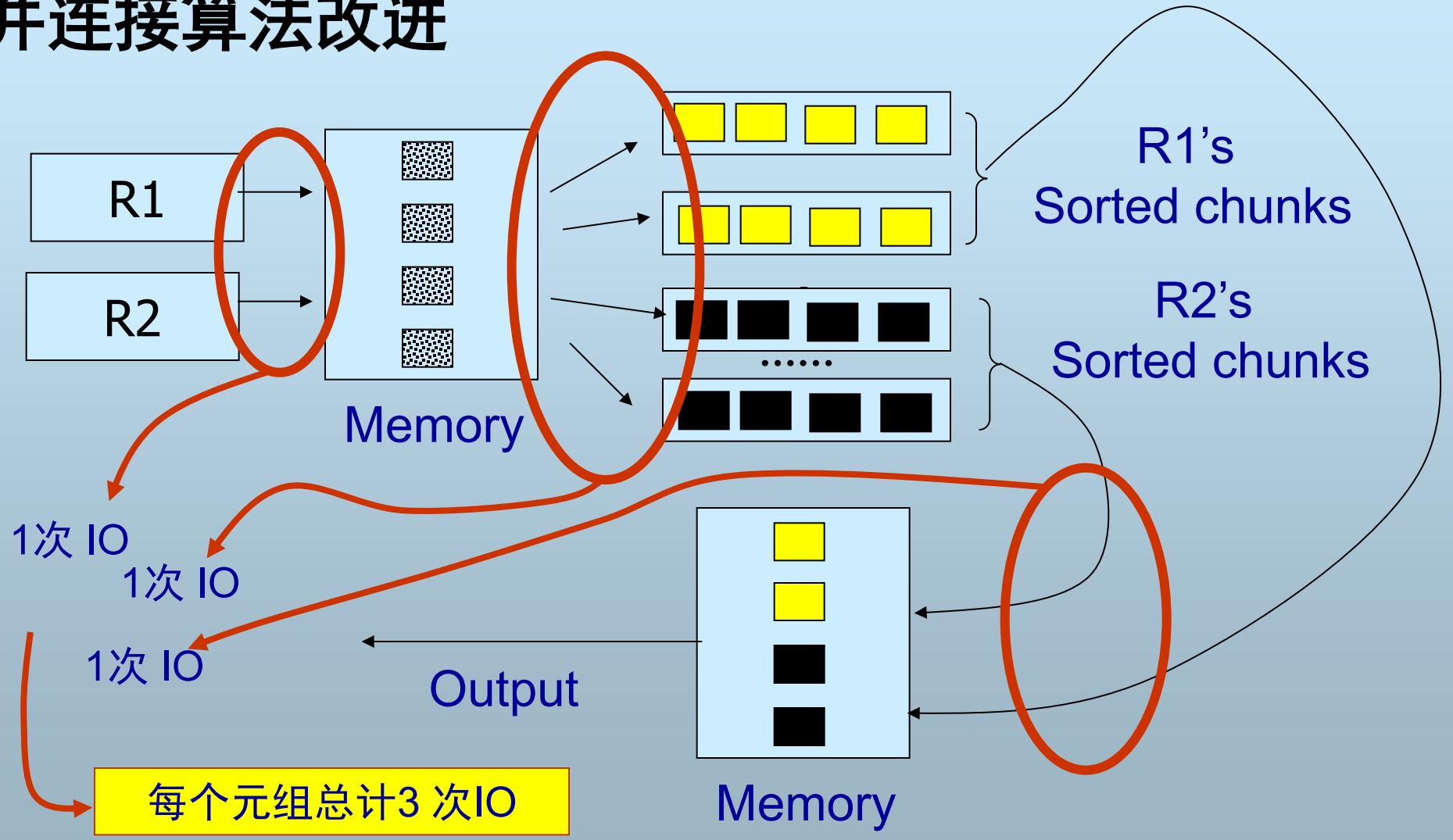
(1) Read R1 and R2 into sorted chunks (each has M blocks)

(2) Read first blocks of both R1's chunks and R2's into buffer

(3) Join in the memory

2、归并连接代价分析

- 归并连接算法改进



2、归并连接代价分析

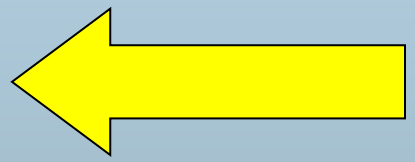
- 归并连接算法改进

- Cost＝3（B(R1)+B(R2)）
  ＝3×1,500＝4,500

- What are required？
  - R1’s #chunks + R2’s #chunks ≤ M

Where we are?

- 物理查询计划操作符
- 连接操作的实现算法
- 连接算法的I/O代价估计
  - 嵌套循环连接代价分析
  - 归并连接代价分析
  - 索引连接代价分析
  - 散列连接代价分析



3、索引连接算法代价分析

- R1(A,C) ⋈ R2(C,D)
  - Assume R1.C index exists
  - Assume R1.C index fits in memory
  - Assume R2 contiguous, unordered

3、索引连接算法代价分析

Algorithm

for each R2 tuple:
- probe index on R1.C (1)
- if match, read R1 tuple (2)

Cost

T(R1)=10,000, T(R2) = 5,000

(0) Read R2 tuples => 500 IOs
(1) Probe index =>No IOs

(2) Read matching R1 tuples => ?

3、索引连接算法代价分析

Matching tuples 选择的基数p估计

1. 若R1.C是主键, R2.C是外键,则 每个R2 tuple在R1中, 选择基数 p =1
2. 若V(R1,C)=5,000, T(R1) = 10,000, 则 每个R2 tuple在R1中的选择基数 p = T(R1)/V(R1,C)=2

3、索引连接算法代价分析

Index join 总代价估计

Cost= B(R2) + T(R2) * p

1. **Cost = 500 + 5000*1 = 5,500**
2. **Cost = 500 + 5000*2 = 10,500**

3、索引连接算法代价分析

- 如果R1.C上的Index不能全部放在内存?
  - Suppose R1.C index is 200 blocks

(1)把第二级索引块(假设只有1块)和另外98块第一级索引块放在Memory中

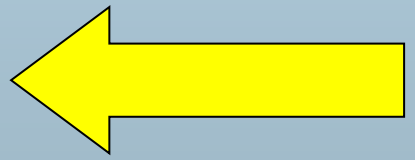
(2)Cost to probe index
=(0 IOs)*(98/200)+(1 IOs)*(102/200)
≈ 0.5 IOs

3、索引连接算法代价分析

- 如果R1.C上的Index不能全部放在内存?

- Cost = B(R2) + T(R2) *(Probe index cost + read tuples)
  - Cost = 500 + 5000 * (0.5 +1) = 8,000
  - Cost = 500 + 5000 * (0.5 + 2) = 13,000

Where are we?

- 物理查询计划操作符
- 连接操作的实现算法
- 连接算法的I/O代价估计
  - 嵌套循环连接代价分析
  - 归并连接代价分析
  - 索引连接代价分析
  - 散列连接代价分析 

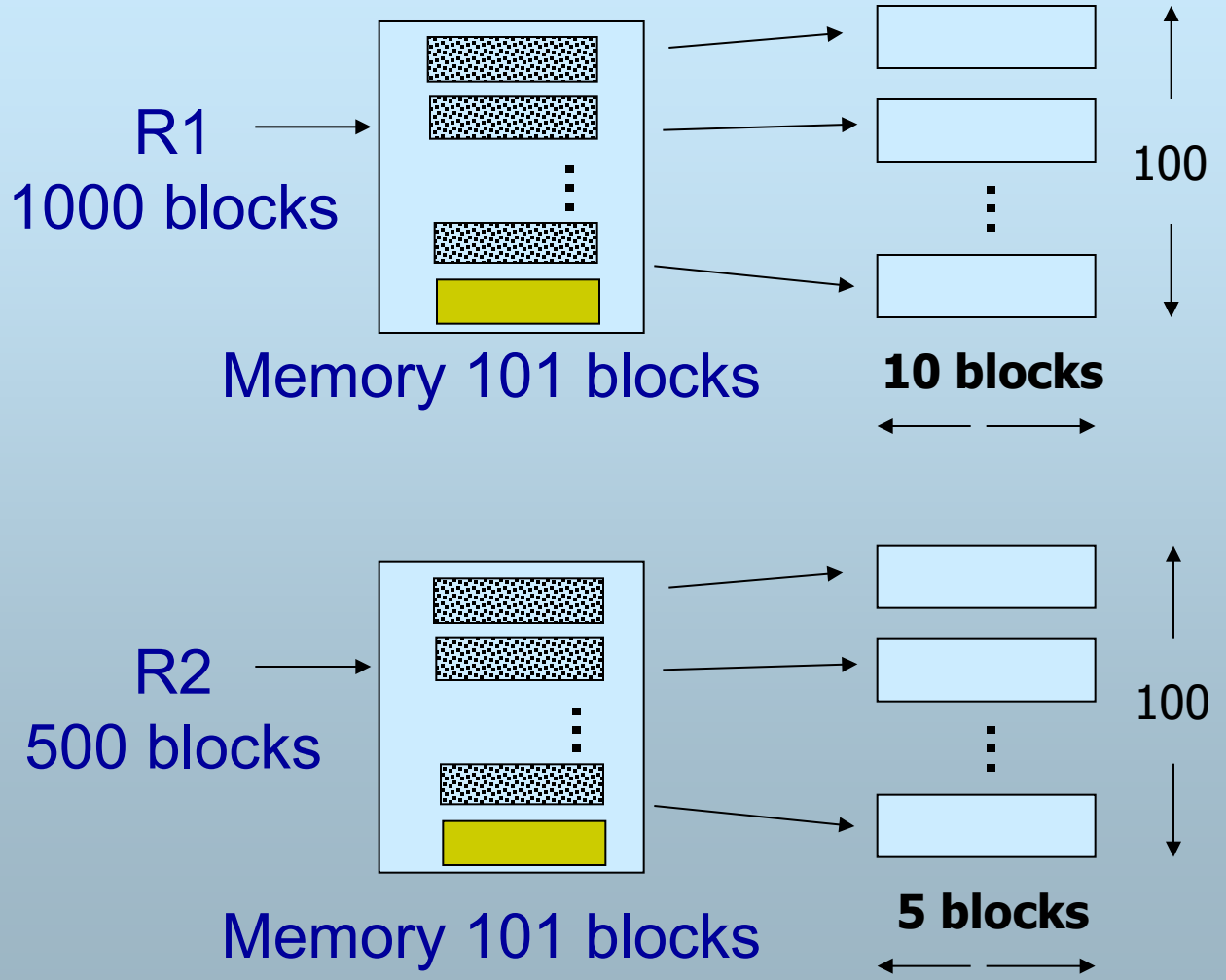
4、散列连接算法代价分析

- **Say R1, R2 contiguous but not ordered**
- **Say 100 hash buckets**

(1)Read R1, Hash, Write into buckets
(2)Read R2, Hash, Write into buckets
(3)Repeat
① Read one bucket of R2 (say B(R2)≤ B(R1))
② Read corresponding R1 bucket
③ Join in the memory

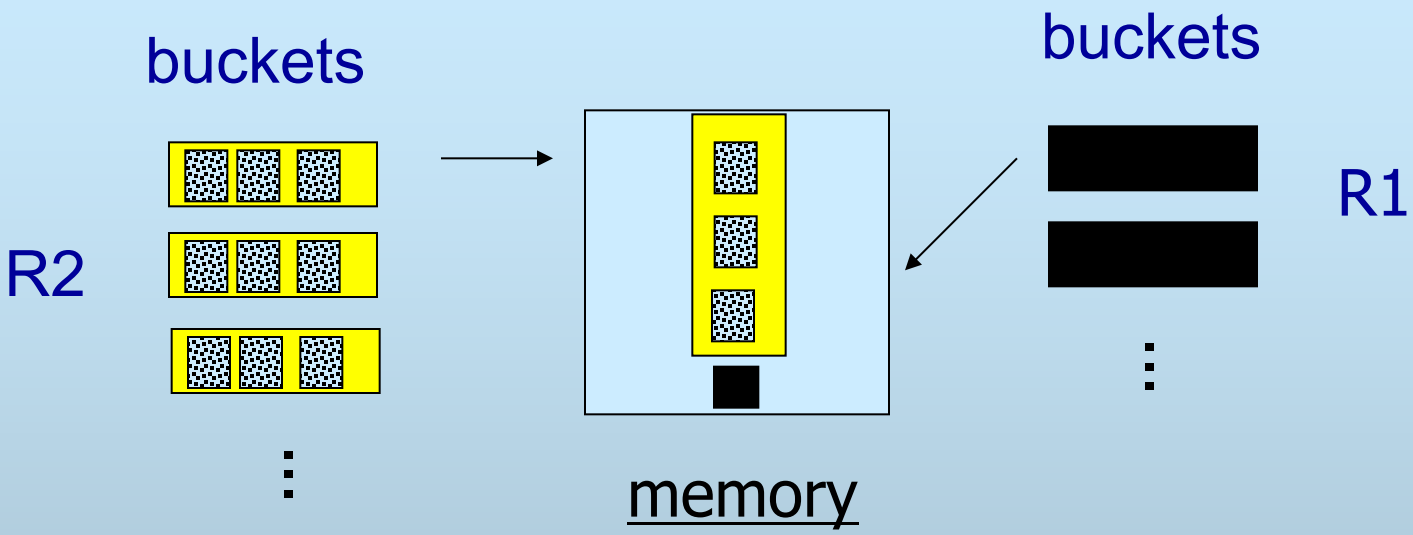
Note：一块一块地读入R1 bucket中的块，并Join。但这不影响IO代价

4、散列连接算法代价分析



Note：
划分为M－1个桶，每一块对应一个桶，最后一块用于读入R1的一块，计算其中每个元组的h，并将元组复制到相应的块中。

4、散列连接算法代价分析



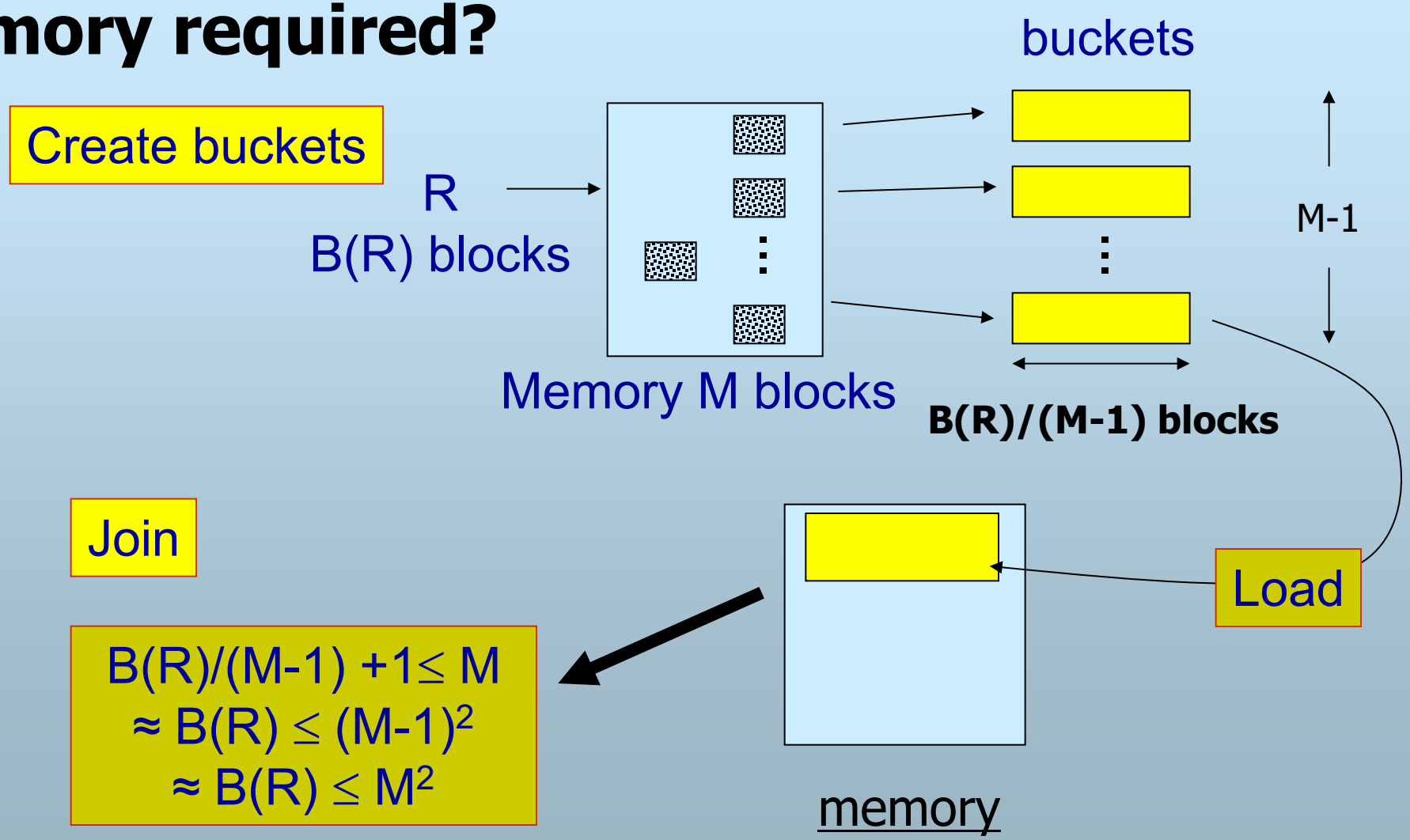
4、散列连接算法代价分析

- **Cost: For each block**
  - **Create buckets**
    - **R1: Read + Write**
    - **R2: Read + Write**
  - **Join**
    - **R1: Read**
    - **R2: Read**

Total : 3 * (B(R1) + B(R2)) = 4,500

4、散列连接算法代价分析

- **Memory required?**



4、散列连接算法代价分析

- **Memory required?**
  - **For R1⋈R2**
  - **Min(B(R1), B(R2)) ≤ M^2**

5、连接算法总结

算法 [1]	Cost	M
Nested Loop Join	B(R2)+B(R1)B(R2)/M	≥ 2
Merge Join	5(B(R1)+B(R2))	$\sqrt{B(R1)}$
Merge Join (improved)	3(B(R1)+B(R2))	$\sqrt{B(R1)+B(R2)}$
Index Join	B(R2)+T(R1)T(R2)/V(R1,C)	LB(R1.C) [2]
Hash Join	3(B(R1)+B(R2))	$\sqrt{B(R2)}$

1: suppose B(R2) ≤ B(R1)
2: suppose index fits in memory

5、连接算法总结

- Nested loop ok for "small" relations (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, hash join usually best
- Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use merge join
- If index exists, it could be useful

  (depends on the size of expected results)

四、连接顺序的选择

- 回顾
  - 影响查询计划I/O代价的因素
    - 实现查询计划的逻辑操作符
    - 中间结果的大小
    - 实现逻辑操作符的物理操作符
    - 物理操作符之间的参数传递方式
    - 相似操作的顺序 
      - 例如，多关系的连接顺序

1、连接的左右变元

- R1(A,C)⋈R2(C,D)：假设连接时总是先将左变元R1读入内存，然后一次一块地读入R2做连接
  - 嵌套循环连接
    - 较小的关系作为左变元
  - 归并连接
    - 排序之后较小的关系作为左变元首先读入内存
  - 索引连接
    - 有索引的关系作为右变元
  - 散列连接
    - 较小的关系作为左变元，散列后将其读入内存

1、连接的左右变元

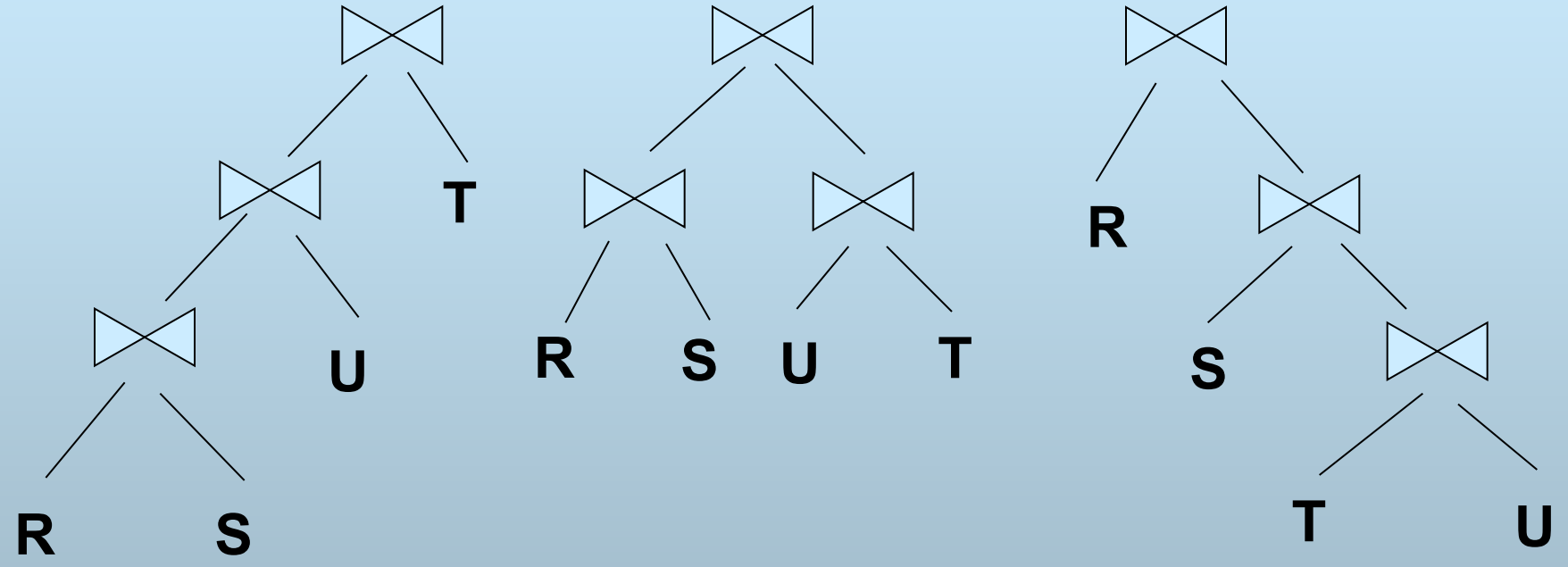
- R1(A,C) ⋈ R2(C,D)
  - R1：建立关系（Build Relation）
    - 连接的左变元，连接时首先将其读入内存（1次或多次I/O）
  - R2：试探关系（Probe Relation）
    - 连接的右变元，连接时读入Build Relation后，一次一块地读入Probe Relation进行连接

2、连接树（Join Tree）

- 仅涉及连接的逻辑查询计划树
- 对于给定的R1，R2，……，Rn，存在多种连接顺序，对应不同的连接树

2、连接树（Join Tree）

- 涉及4个关系连接的3种连接树



左深连接树
Left-deep Join Tree

紧密树
Bushy Tree

右深连接树
Right-deep Join Tree

2、连接树（Join Tree）

- 我们在连接顺序选择时一般只考虑Left-deep Join Tree
  - 理由1：给定n个关系的left-deep join tree数目相对较小，因此在计划枚举时容易搜索
  - 理由2：基于左深树的连接比其他非左深树的连接更高效

2、连接树（Join Tree）

- 给定n个关系的一种树形状，可能的连接顺序为n！
- 给定n个关系的树形状数 *T(n)*

$$T(1)=1$$

$$T(n)=\sum_{i=1}^{n-1} T(i)T(n-i)$$

T(i): i个关系组成的左子树形状数
T(n-i): n-i个关系组成的右子树形状数

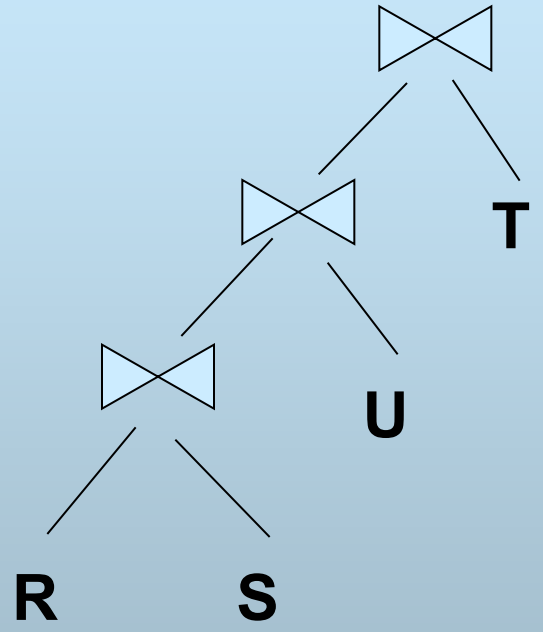
- 给定n个关系的连接树数目＝*T(n)* * *n*！

2、连接树（Join Tree）

- 例：6个关系的连接树数= T(6) * 6! =30240
  - T(1) =1, T(2) =1, T(3) =2, T(4) =5, T(5) =14, T(6) = 42
  - 6! =720
  - 左深树数＝6！＝720

2、连接树（Join Tree）

- 左深树更高效(减少内存要求)



首先要读入左变元
R ⋈ S ⋈ U

为此要读入 R ⋈ S，
再读U连接

为此要读入 R,
再读S连接

In Memeory

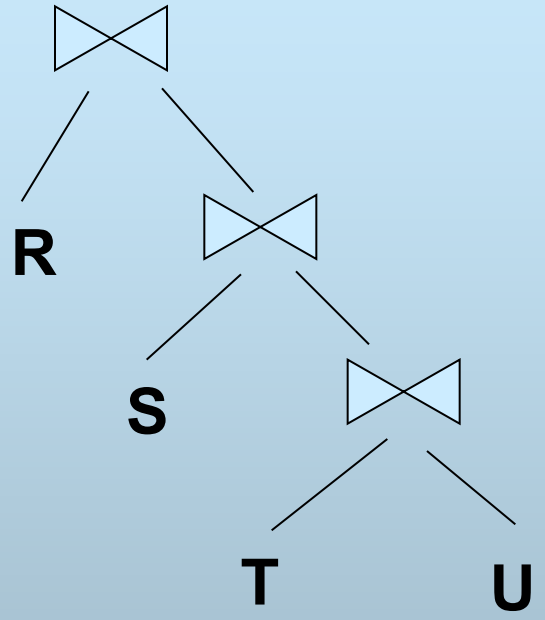
B(R⋈ S ⋈U)+
B(R⋈S ⋈U ⋈ T)

B(R⋈S)+
B(R⋈S ⋈U)

B(R)+B(R⋈S)

若 左变元可以一次读入内存, 则连接需要在内存中存储至多两个临时关系

2、连接树（Join Tree）



首先要读入左变元 R

然后要计算 S ⋈ T ⋈ U
为此要读入S

为此要计算 T ⋈ U
为此要读入T

In Memeory

B(R)

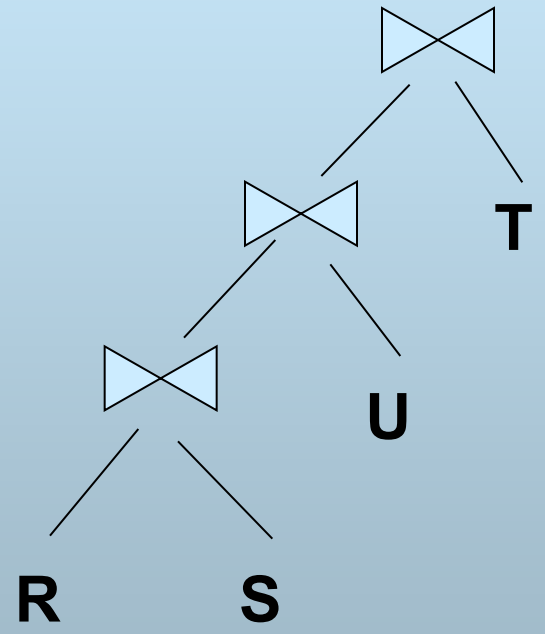
B(R)+B(S)

B(R)+B(S)+B(T)

若 左变元可以一次读入内存, 则n个关系的右深树连接需要在内存中存储n-1个关系

2、连接树（Join Tree）

- 若用嵌套循环连接算法,左深树可避免重复构建中间关系



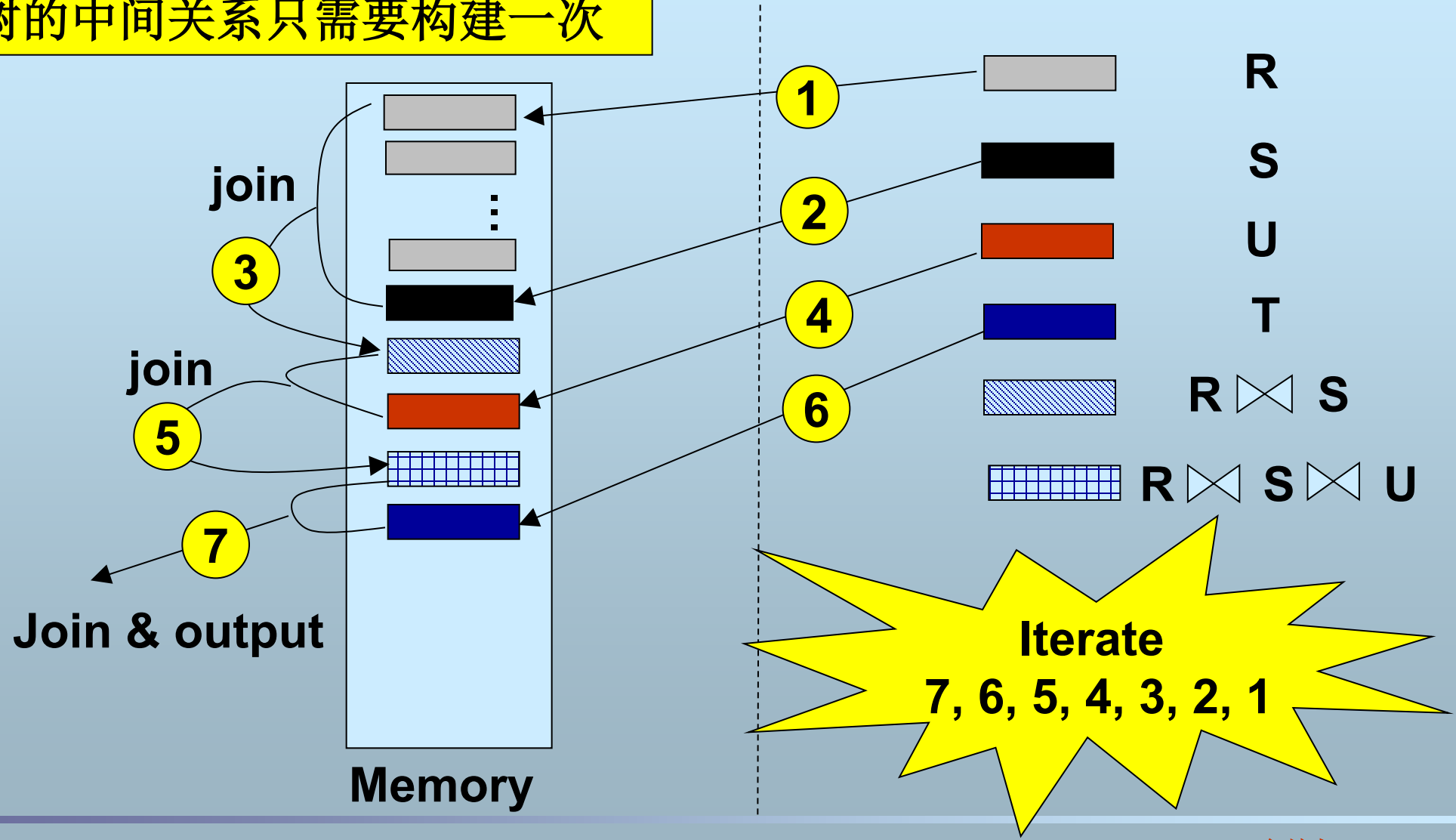
左变元 $R \bowtie S \bowtie U$ 的每一块, 都需要循环读T进行连接

为此要循环读入 $R \bowtie S$，再循环读U连接

为此要循环读入 R，再读S连接

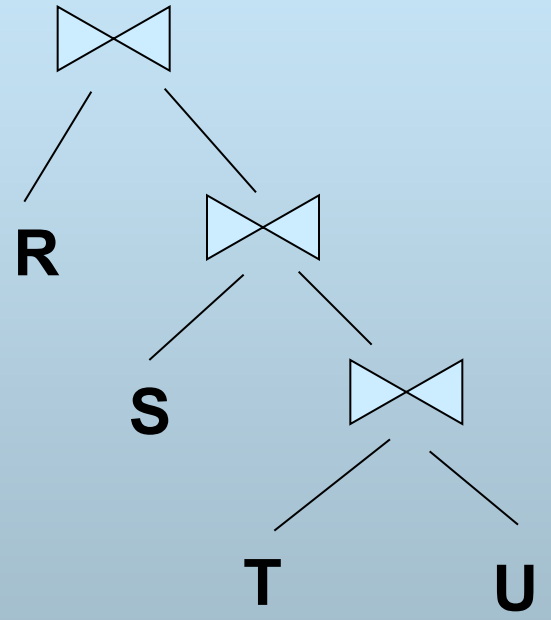
2、连接树（Join Tree）

左深树的中间关系只需要构建一次



2、连接树（Join Tree）

- 若用右深树, 需重复构建中间关系



每读左变元 R 的一块, 都要构建 $S \bowtie T \bowtie U$ 并将其与R的读入块连接;

若将 $S \bowtie T \bowtie U$ 存储在磁盘,需要使用额外的I/O

若将其存储在内存, 则占用过多的缓冲区

3、动态规划法选择连接顺序

- **Dynamic Programming**
  - 构造并存储问题子集的代价表，在求解上一级问题时可以参照下一级问题（子集）的代价，从而避免重复计算子集代价，提高问题求解的效率

3、动态规划法选择连接顺序

- 对于n个关系的连接，构造一个或多个关系的子集表，每个子集 $\mathcal{R}$（1或m个关系）包括
  - T($\mathcal{R}$): $\mathcal{R}$ 的大小估计值
  - f($\mathcal{R}$): $\mathcal{R}$ 的最小代价（我们用中间关系大小的和作为代价计算依据）
  - $\mathcal{R}$ 的最佳计划（连接顺序）

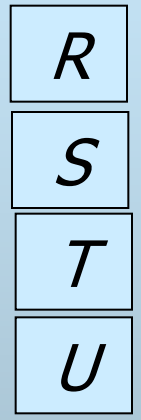
状态转移方程：f(n)=f(n-1)+T(n-1)

3、动态规划法选择连接顺序

- 问题：求解n个关系的最佳连接顺序
- 步骤
  - 填写涉及1个关系和2个关系的子集表
  - 填写3个关系的子集表，此时可以参考2个关系的子集表
  - 继续填写4个关系的子集表，此时可以参考2个关系和3个关系的子集表
  - 直到填充完n个关系的子集表为止

3、动态规划法选择连接顺序

- 例：求解 $R(a,b) \bowtie S(b,c) \bowtie T(c,d) \bowtie U(d,a)$ 的最佳连接顺序



T(*R*) = 1000 V(R,a)=100 V(R,b)=200
T(*S*) = 1000 V(S,b)=100 V(S,c)=500
T(*T*) = 1000 V(T,c)=20 V(T,d)=50
T(U) = 1000 V(U,d)=1000 V(U,a)=50

3、动态规划法选择连接顺序

- 1个关系的子集表

	{R}	{S}	{T}	{U}
大小	1000	1000	1000	1000
最小代价	0	0	0	0
最佳计划	R	S	T	U

3、动态规划法选择连接顺序

- 2个关系的子集表

	{R,S}	{R,T}	{R,U}	{S,T}	{S,U}	{T,U}
大小T_{n-1}	5000	10^6	10000	2000	10^6	1000
最小代价f_{n-1}	0	0	0	0	0	0
最佳计划	R ⋈ S	R ⋈ T	R ⋈ U	S ⋈ T	S ⋈ U	T ⋈ U

3、动态规划法选择连接顺序

- 3个关系的子集表
  - 共{R,S,T}，{R,S,U}，{R,T,U}，{S,T,U}四种
  - 对于每一种子集，我们只考虑左深树，因此只考虑其中2个关系的子集作为左变元的情况

3、动态规划法选择连接顺序

- 3个关系的子集表
  - 例如 {R,S,T}, 由前面的子表得知， {R,S,T}左子树的最佳计划为

    R ⋈ S,　R ⋈ T,　S ⋈ T
  - 分别构造三种情况下的左深树
    - 估计大小及代价，选择最小代价的计划

最佳！

(R ⋈ S) ⋈ T　　(R ⋈ T) ⋈ S　　(S ⋈ T) ⋈ R

代价＝T(R ⋈ S)　　代价＝T(R ⋈ T)　　代价＝T(S ⋈ T)

＝5000　　＝1000000　　＝2000

由2个关系的子表得到

3、动态规划法选择连接顺序

- 3个关系的子集表

	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
大小T_{n-1}	10000	50000	10000	2000
最小代价f_{n-1}	2000	5000	1000	1000
最佳计划	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

3、动态规划法选择连接顺序

- 4个关系的子集表
  - 由于n=4，所以只有{R,S,T,U}一种情况，该子集表的结果就是我们求解的答案
  - 我们只考虑左深树，因此只考虑其中3个关系的子集作为左变元的情况

3、动态规划法选择连接顺序

$f(n)=f(n-1)+T(n-1)$

$(S \bowtie T) \bowtie R \Rightarrow ((S \bowtie T) \bowtie R) \bowtie U$

代价=T($(S \bowtie T) \bowtie R$)
+ ($(S \bowtie T) \bowtie R$)连接的最小代价

12000

$(R \bowtie S) \bowtie U \Rightarrow ((R \bowtie S) \bowtie U) \bowtie T$ 代价 55000

$(T \bowtie U) \bowtie R \Rightarrow ((T \bowtie U) \bowtie R) \bowtie S$ 代价 11000

$(T \bowtie U) \bowtie S \Rightarrow ((T \bowtie U) \bowtie S) \bowtie R$ 代价 3000

	{R,S,T}	{R,S,U}	{R,T,U}	{S,T,U}
大小T_{n-1}	10000	50000	10000	2000
最小代价f_{n-1}	2000	5000	1000	1000

本章小结

- 物理查询计划操作符
- 连接操作的实现算法
  - 嵌套循环连接
  - 归并连接
  - 索引连接
  - 散列连接
- 连接算法的I/O代价估计
- 连接顺序的选择
- Other operators？ -- see textbook