CS3223 Lecture 5
Query Evaluation: Projection & Join

Projection: $\pi_{A_1,\cdots,A_m}(R)$

- $\pi_L(R)$ projects columns given by list L from relation R
- Example: **select distinct** age **from** R

Relation R

name	age	weight	height
Alice	17	48	175
Bob	15	60	178
Curly	10	65	171
Larry	12	70	175
Lucy	17	45	170
Moe	10	55	180

$\pi_{\mathbf{age}}(\mathbf{R})$

age
17
15
10
12

- $\pi_L^*(R)$ same as $\pi_L(R)$ but preserves duplicates

$\pi^*_{\mathbf{age}}(\mathbf{R})$

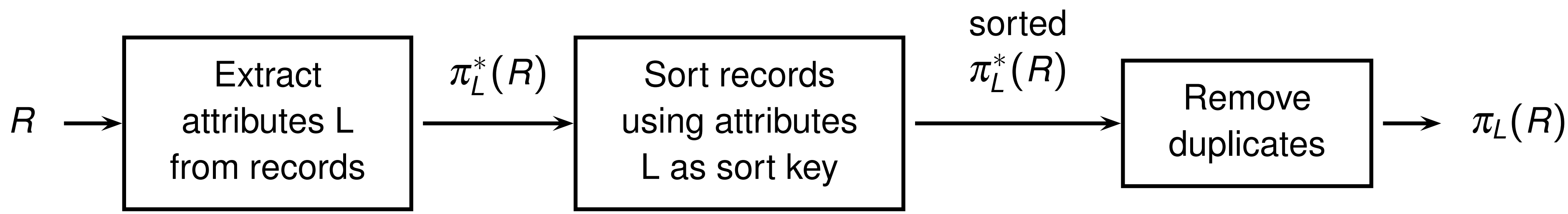
age
17
15
10
12
17
10

Projection Operation, $\pi_{A_1,\cdots,A_m}(R)$

- Projection involves two tasks:
  1. remove unwanted attributes
  2. eliminate any duplicate tuples produced
- Two approaches:
  - Projection based on sorting
  - Projection based on hashing

Sort-based Approach

Consider $\pi_L(R)$ where L denote some sequence of attributes of R



Relation R

name	age	weight	height
Alice	17	48	175
Bob	15	60	178
Curly	10	65	171
Larry	12	70	175
Lucy	17	45	170
Moe	10	55	180

$\pi^*_{\textbf{age}}(\mathbf{R})$

age
17
15
10
12
17
10

Sorted $\pi^*_{\textbf{age}}(\mathbf{R})$

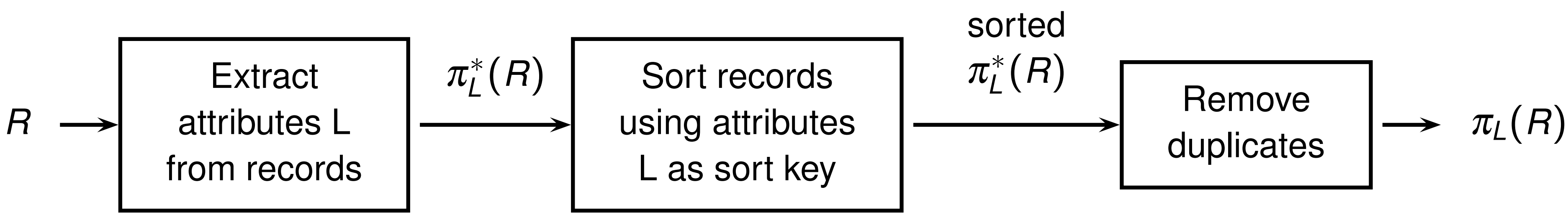
age
10
10
12
15
17
17

$\pi_{\textbf{age}}(\mathbf{R})$

age
10
12
15
17

Sort-based Approach

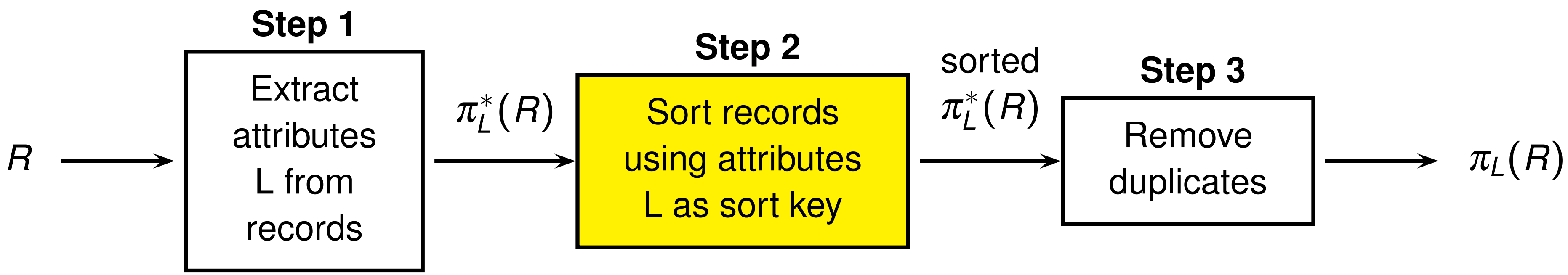
Consider $\pi_L(R)$ where L denote some sequence of attributes of R



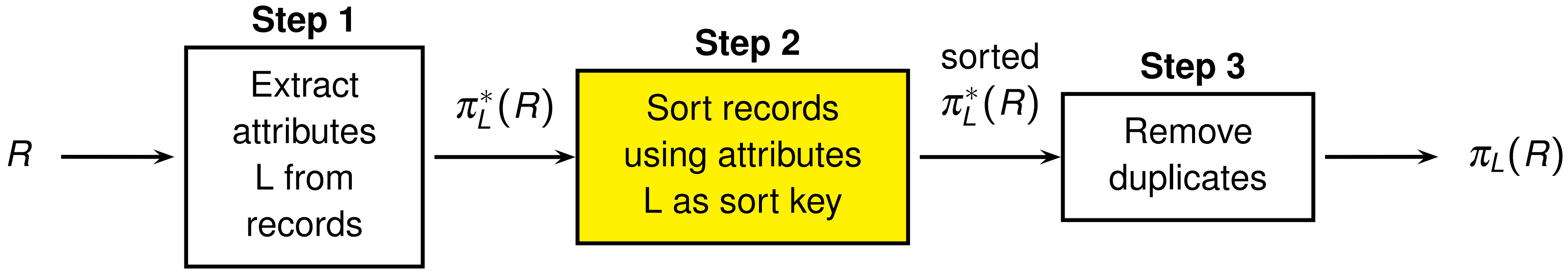
Cost Analysis

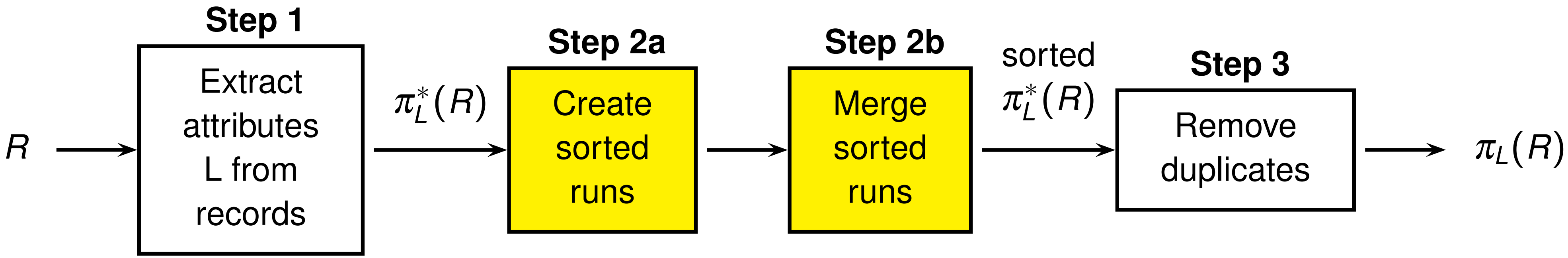
- **Step 1**:
  - Cost to scan records = $|R|$
  - Cost to output temporary result = $|\pi_L^*(R)|$
- **Step 2**:
  - Cost to sort records = $2|\pi_L^*(R)|\ (\log_m(N_0) + 1)$
  - $N_0 =$ number of initial sorted runs, $\quad m =$ merge factor
- **Step 3**:
  - Cost to scan records = $|\pi_L^*(R)|$

Optimized Sort-based Approach

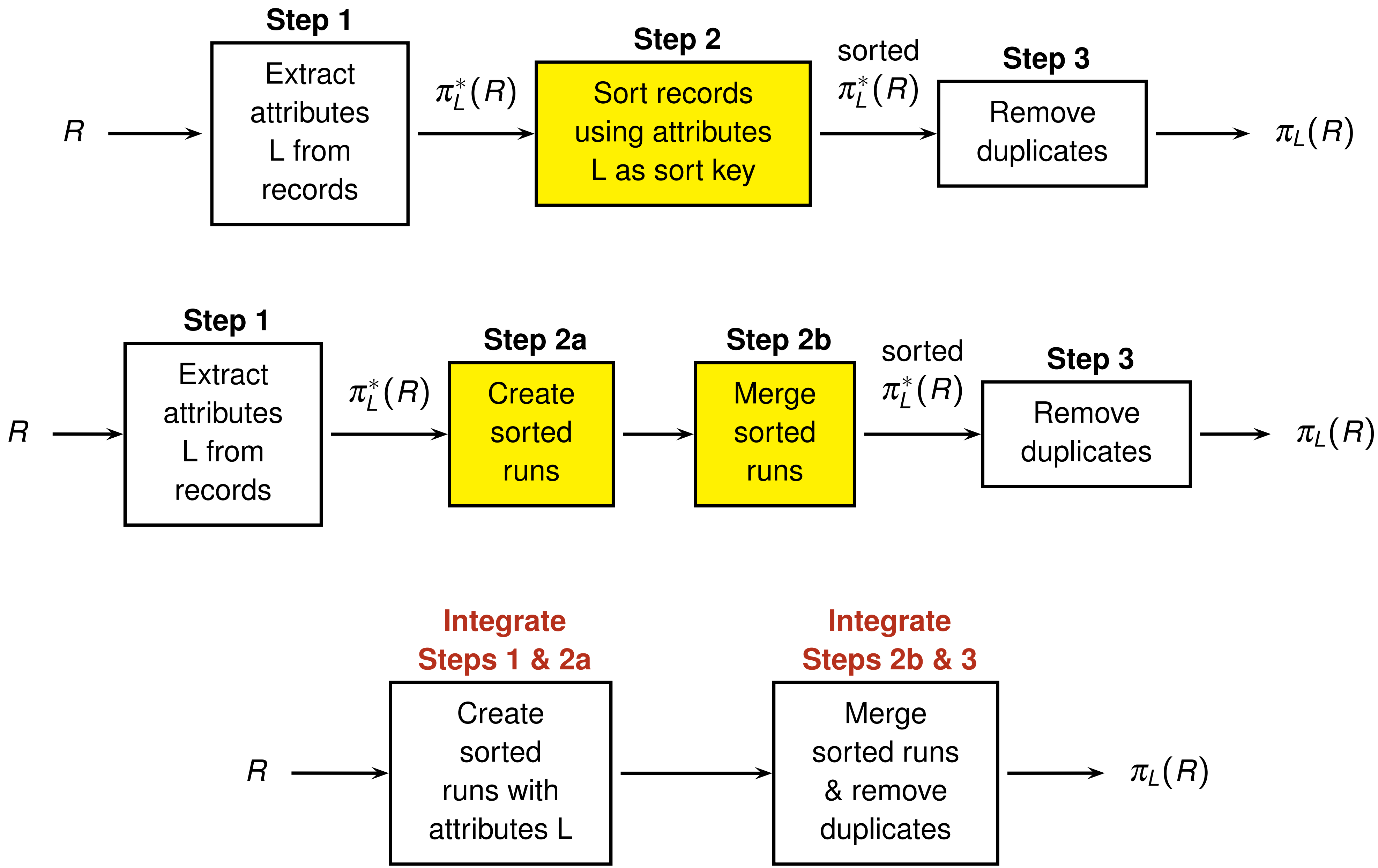


Optimized Sort-based Approach





Optimized Sort-based Approach



Hash-based Approach

- Consider $\pi_L(R)$
- Build a main-memory hash table to detect & remove duplicates

initialize an empty hash table T
for each tuple t in R do
    apply hash function h on $\pi_L(t)$
    let t be hashed to bucket B_i in T
    if ($\pi_L(t)$ is not in B_i) then
        insert $\pi_L(t)$ into B_i
return all entries in T

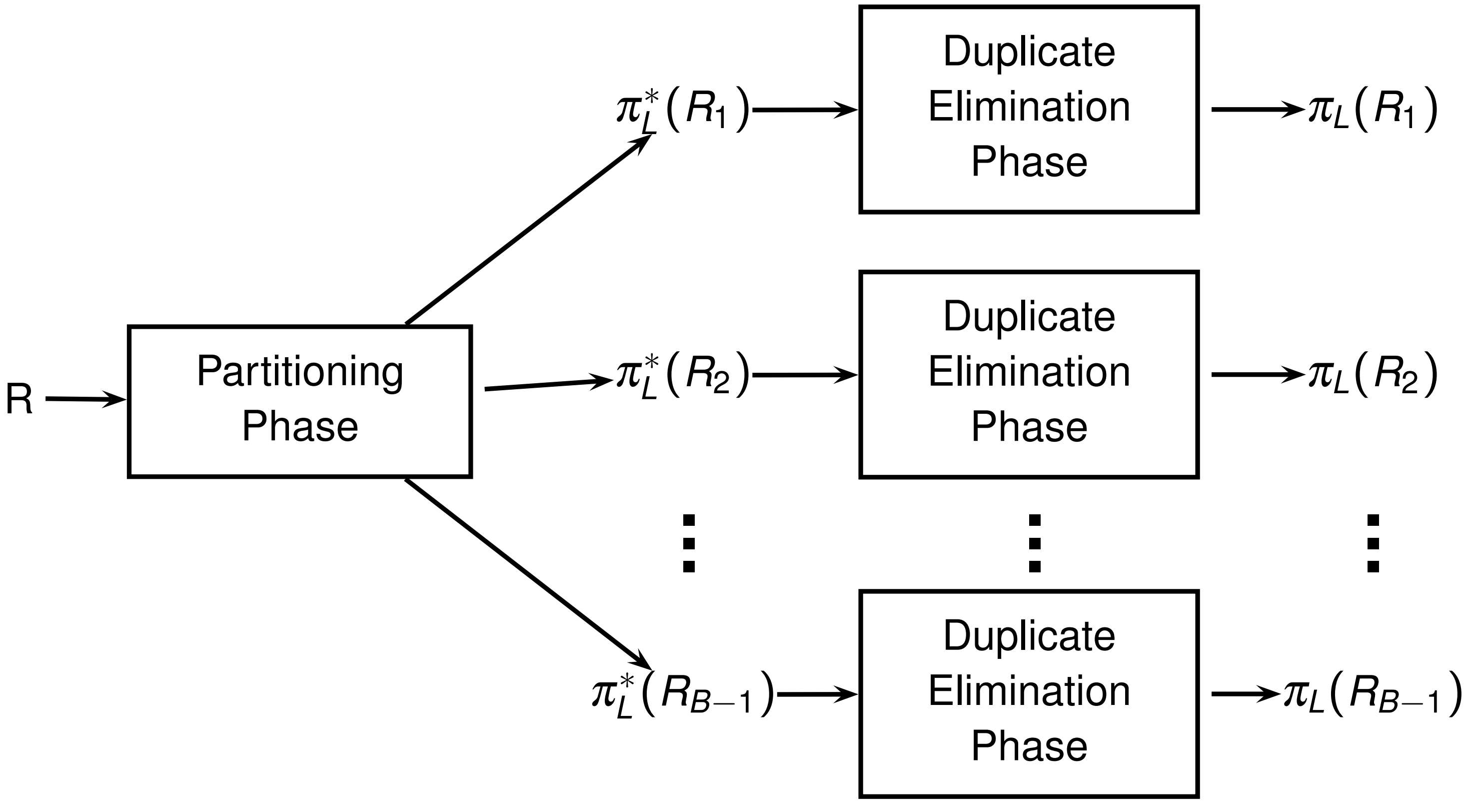
Hash-based Approach

Consists of two phases:

1. Partitioning phase: partitions R into $R_1, R_2, \cdots, R_{B-1}$
    - Hash on $\pi_L(t)$ for each tuple $t \in R$
    - $R = R_1 \cup R_2 \cup \cdots \cup R_{B-1}$
    - $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$ for each pair R_i & R_j, $i \neq j$
2. Duplicate elimination phase: eliminates duplicates from each $\pi_L^*(R_i)$

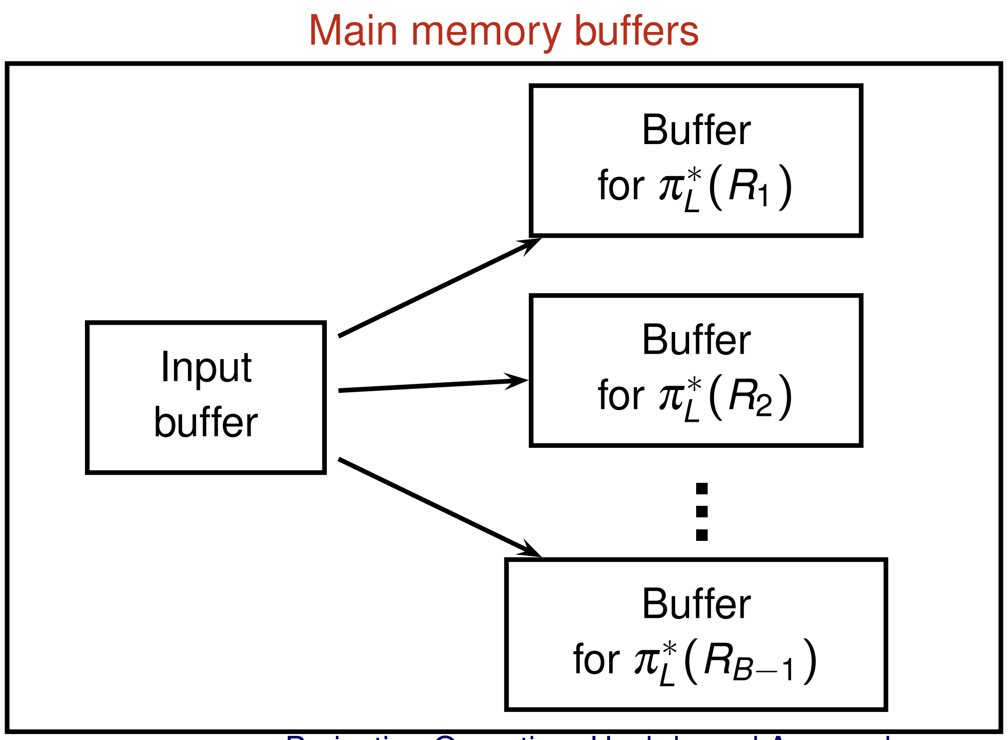
$\pi_L(R)$ = duplicate-free union of $\pi_L(R_1), \pi_L(R_2), \cdots, \pi_L(R_{B-1})$

Hash-based Approach (cont.)



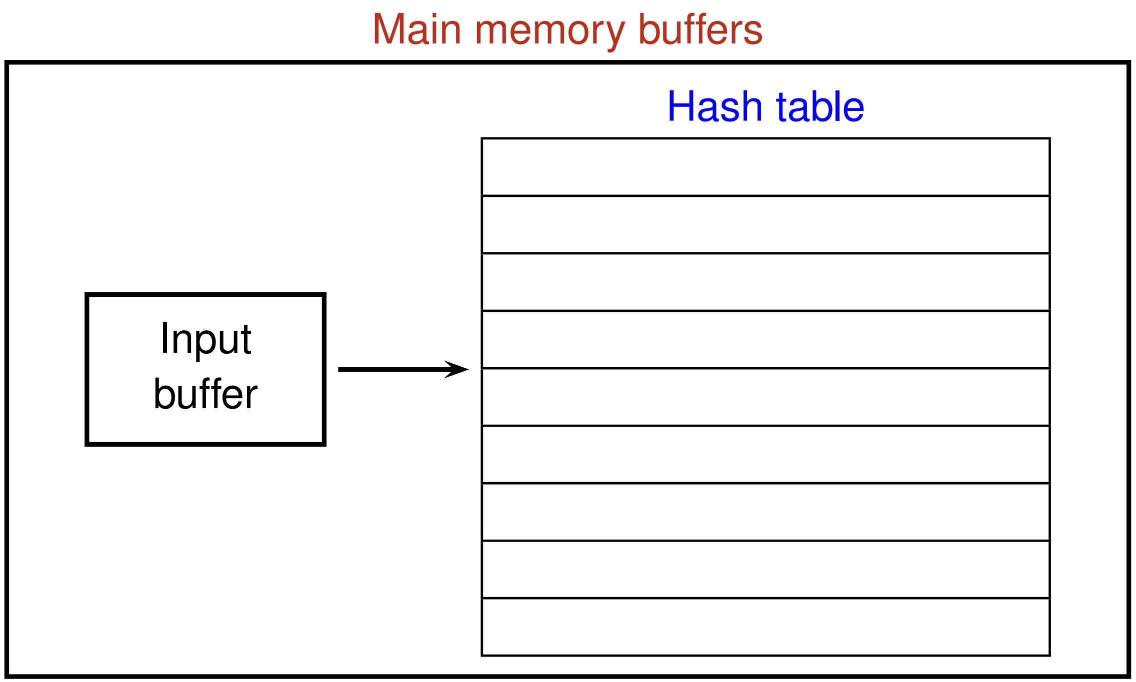
Partitioning Phase

- Use one buffer for input & $(B - 1)$ buffers for output
- Read R one page at a time into input buffer
- For each tuple t in input buffer,
  - project out unwanted attributes from t to form t'
  - apply a hash function h on t' to distribute t' into one output buffer
  - flush output buffer to disk whenever buffer is full

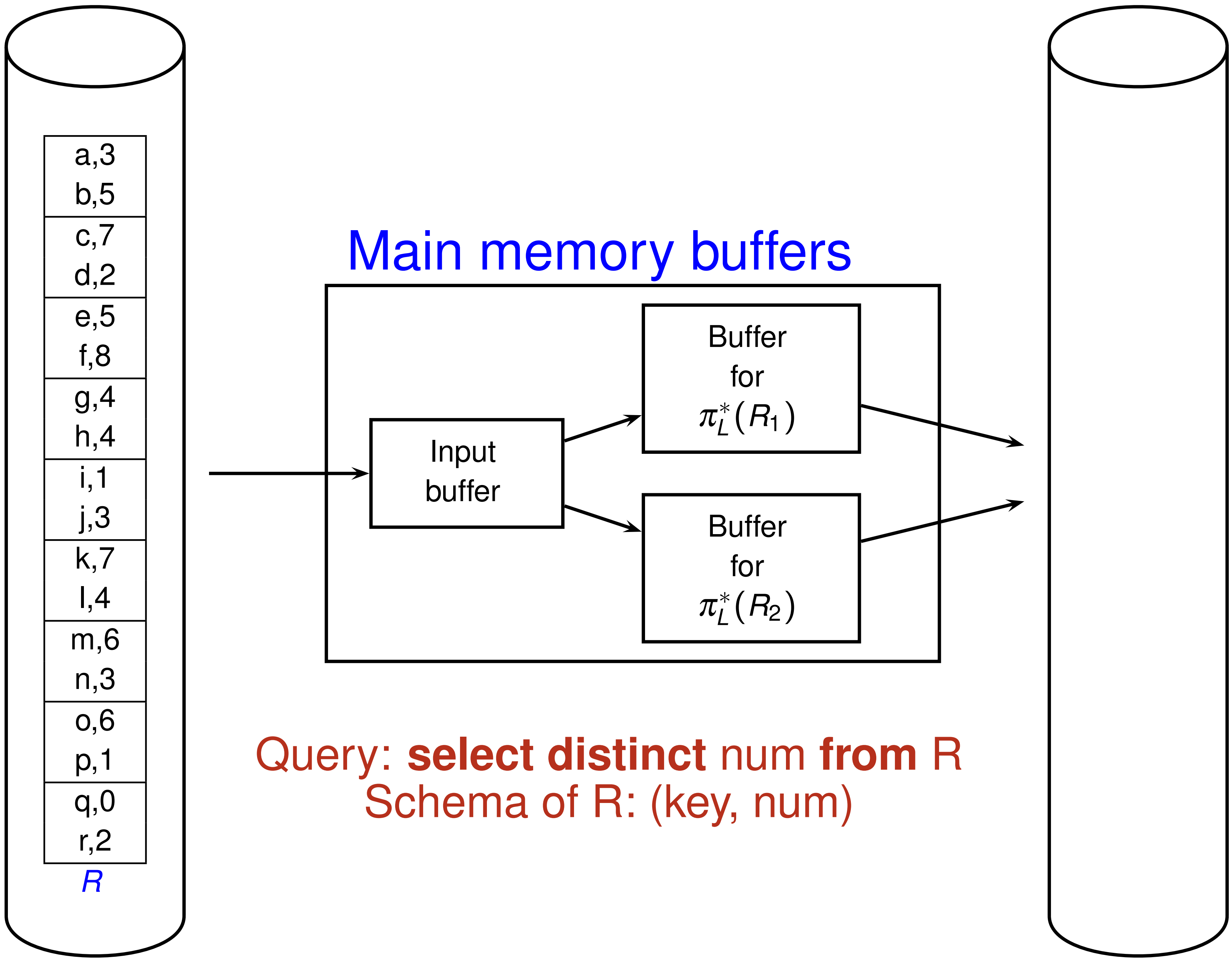


Duplicate Elimination Phase

- For each partition R_i,
  - Initialize an in-memory hash table
  - Read $\pi_L^*(R_i)$ one page at a time; for each tuple t read,
    - Hash t into bucket B_j with hash function h' ($h' \neq h$)
    - Insert t into B_j if $t \notin B_j$
  - Write out tuples in hash table to results

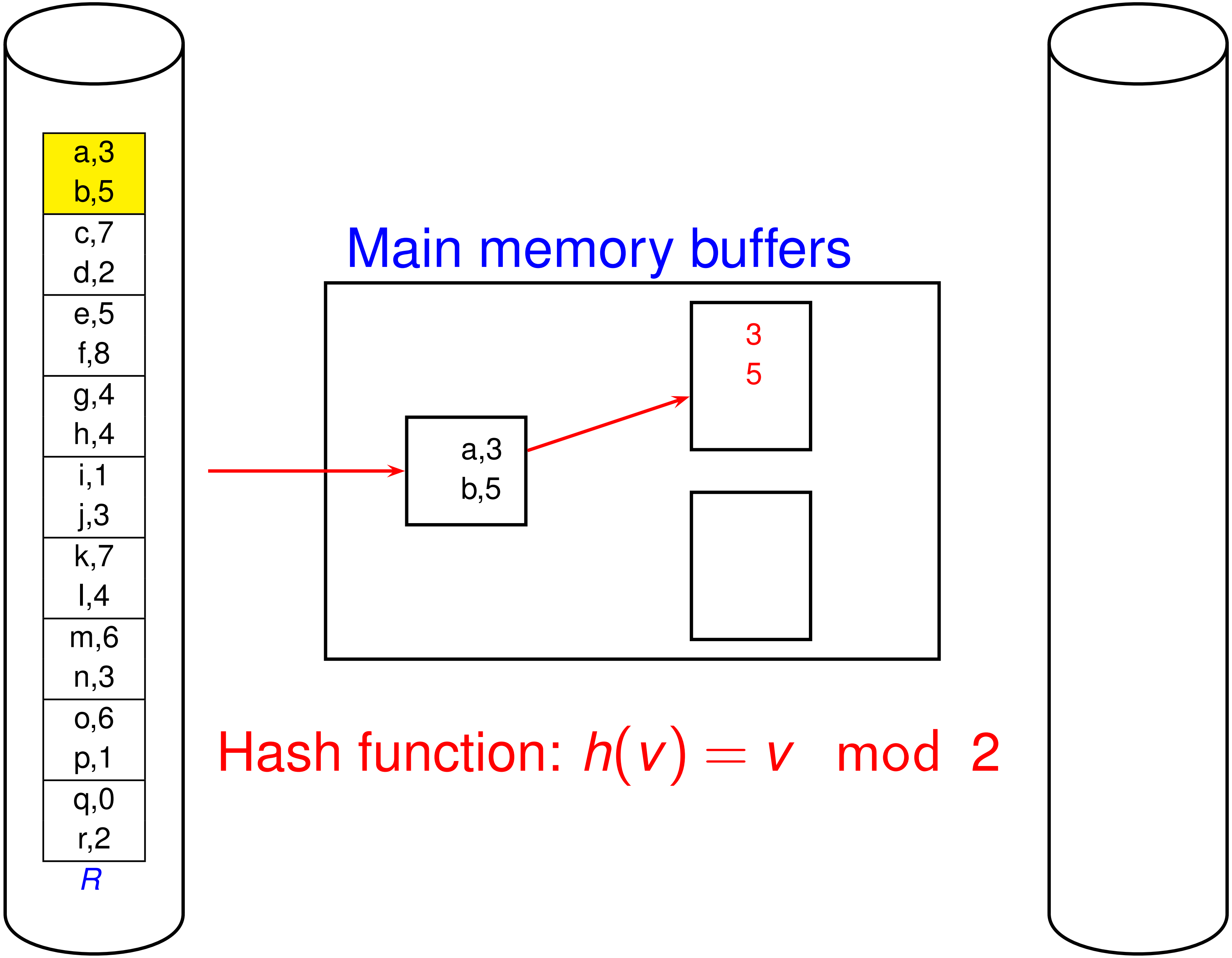


Partitioning Phase



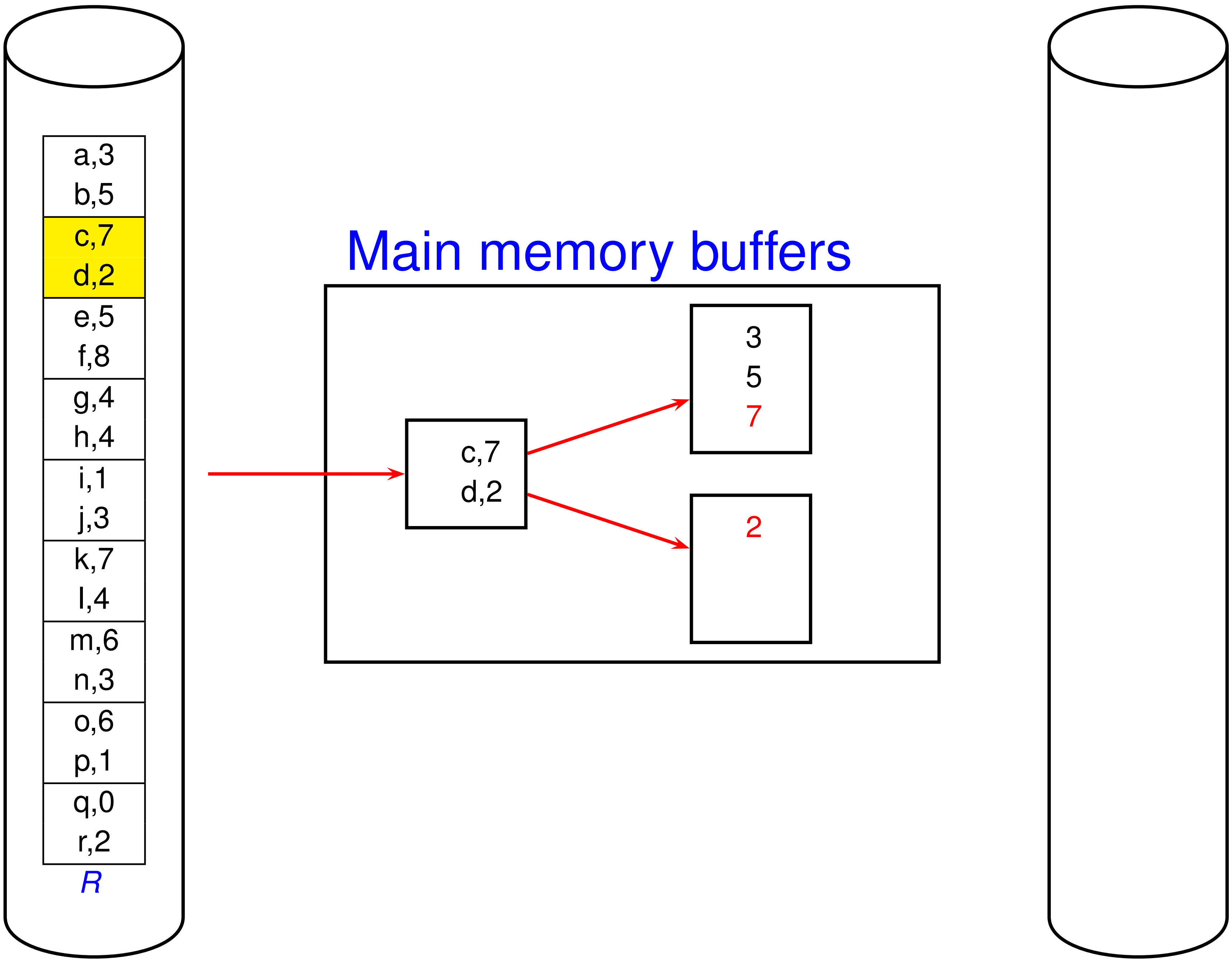
Query: **select distinct** num **from** R
Schema of R: (key, num)

Partitioning Phase

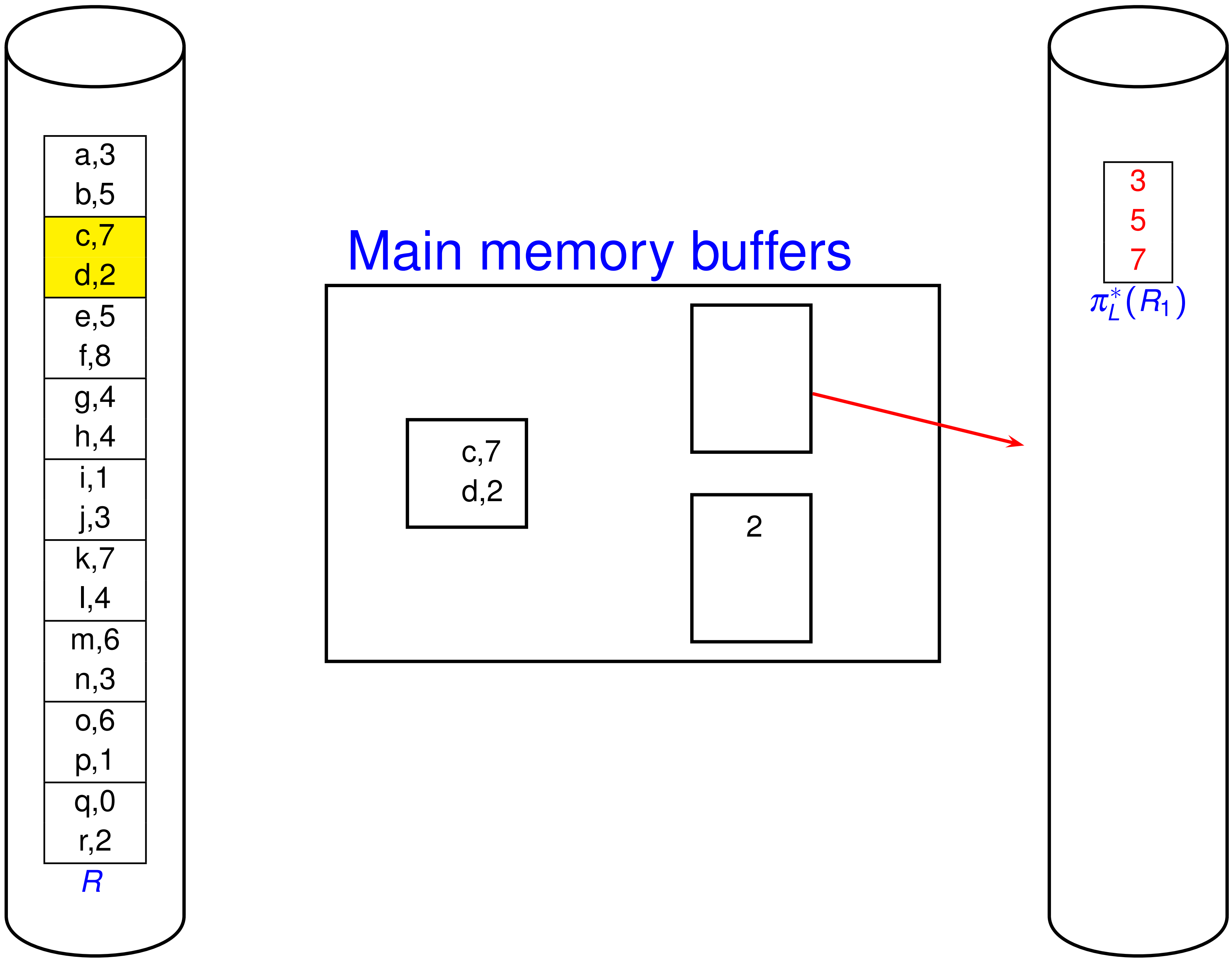


Hash function: $h(v) = v \mod 2$

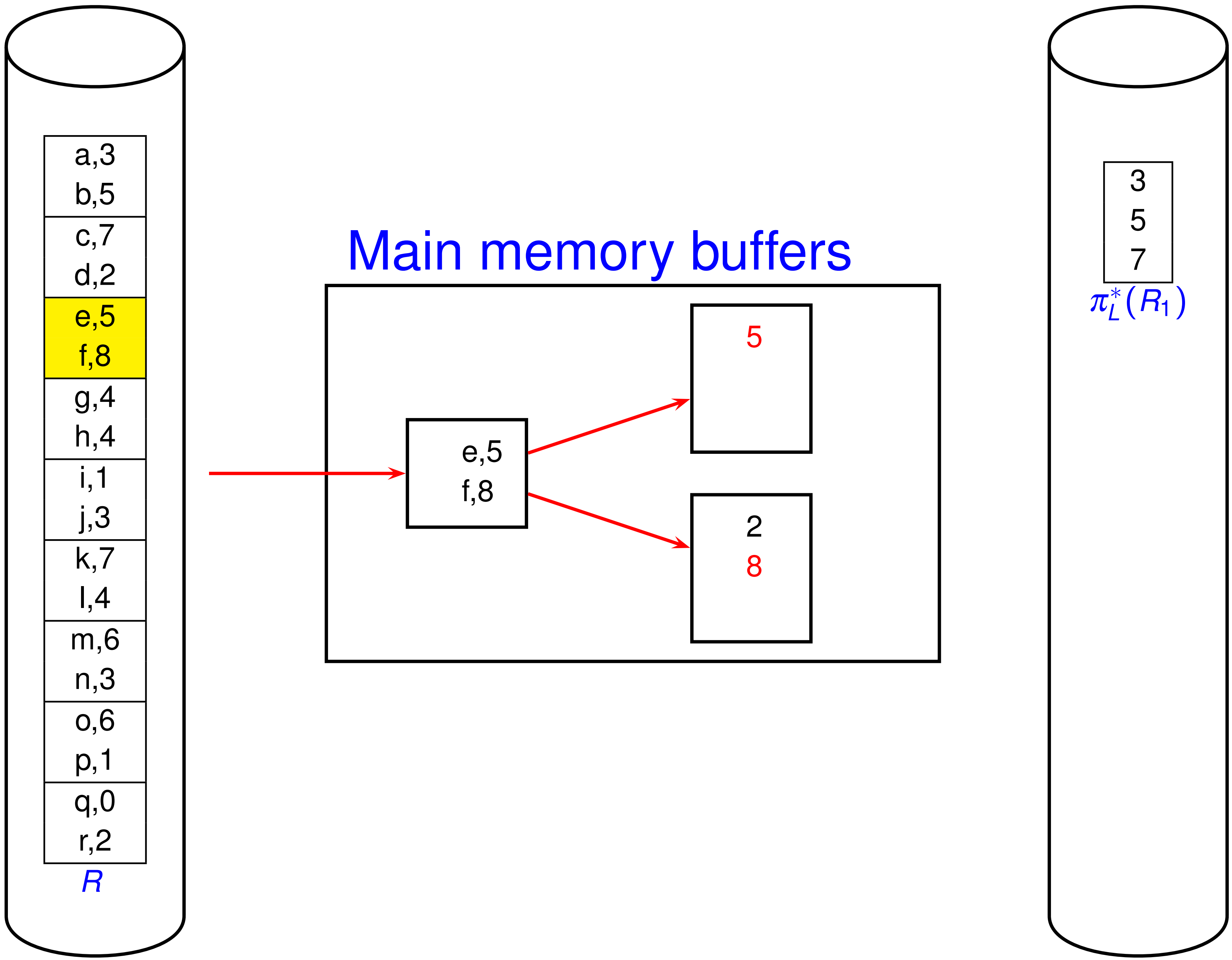
Partitioning Phase



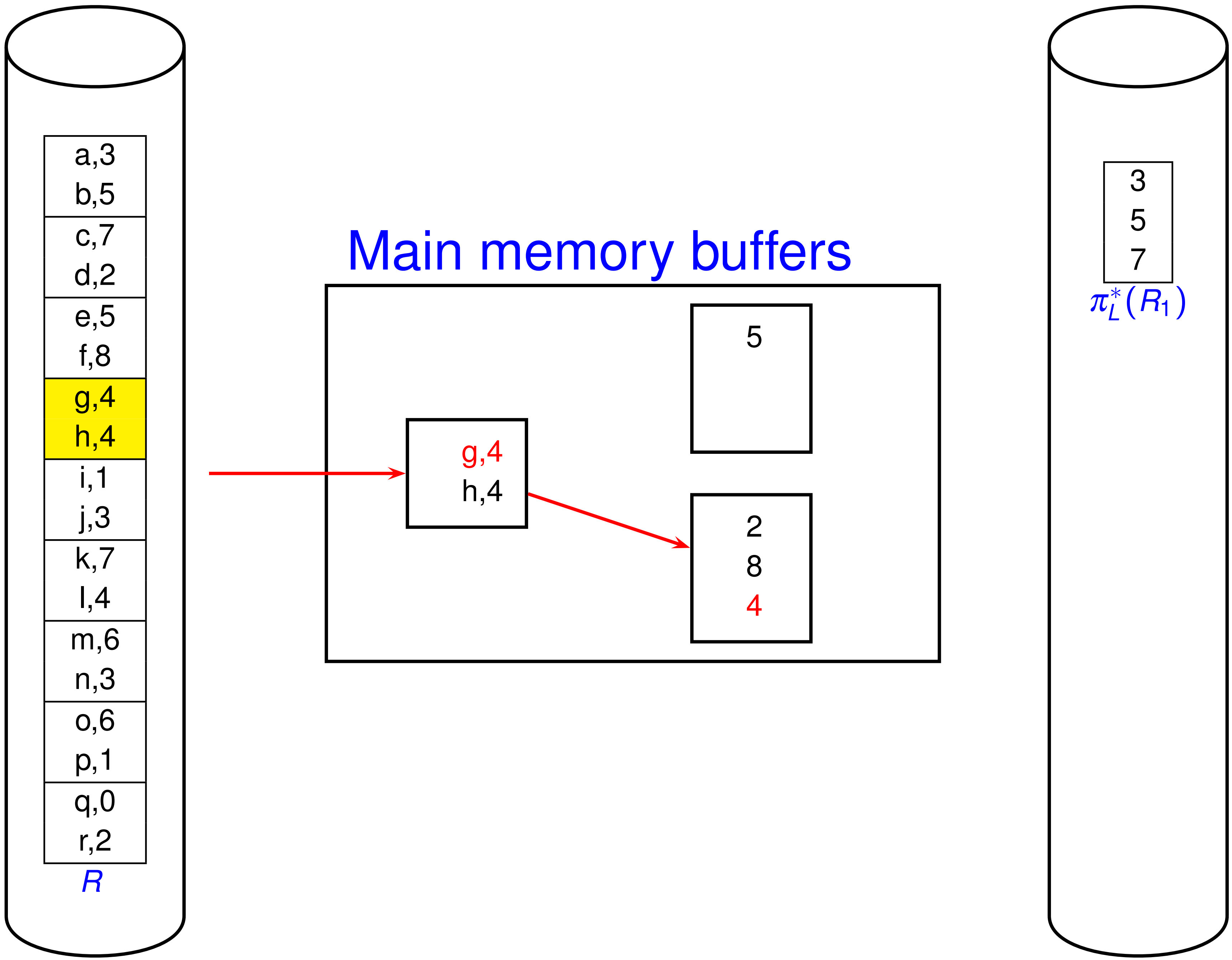
Partitioning Phase



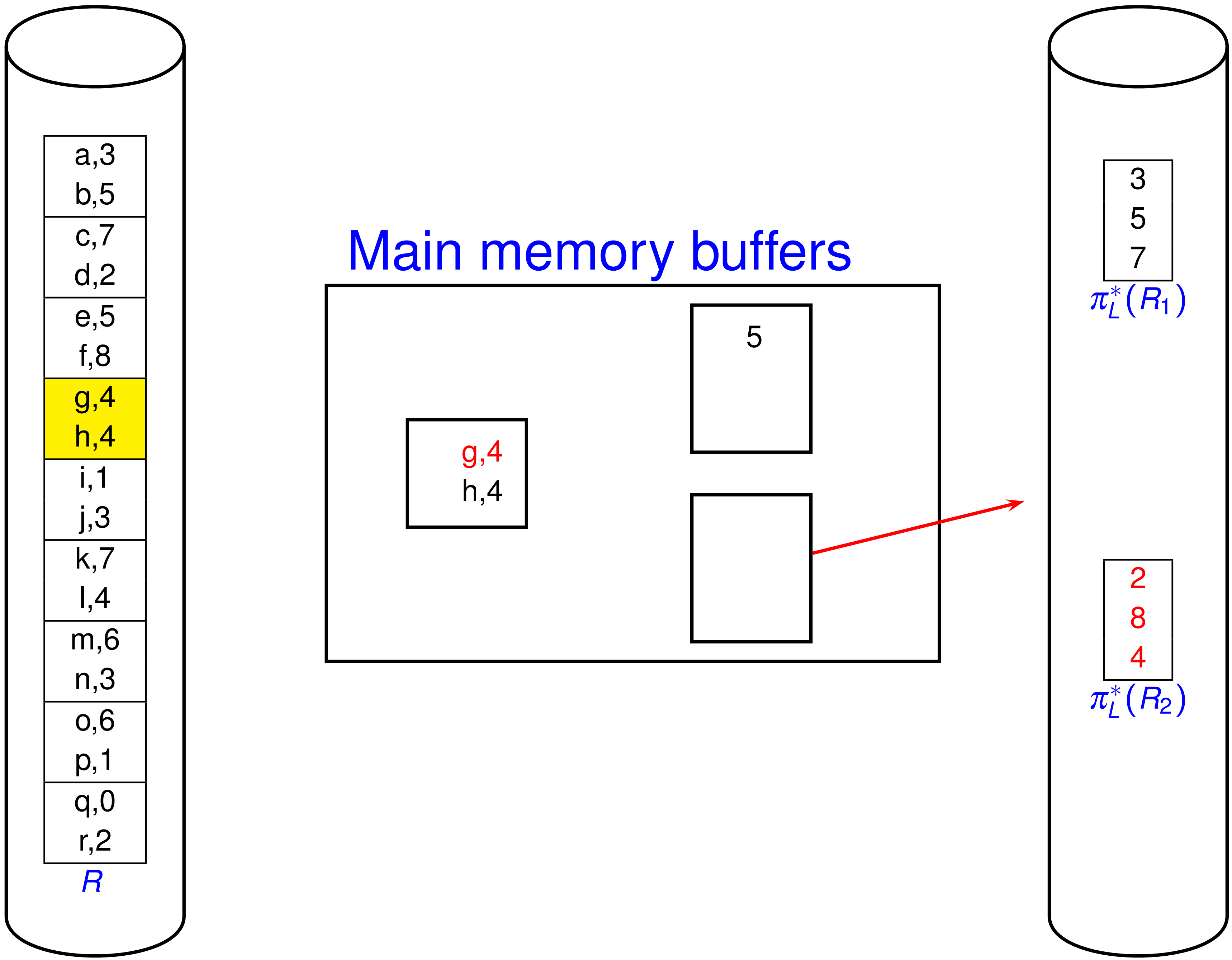
Partitioning Phase



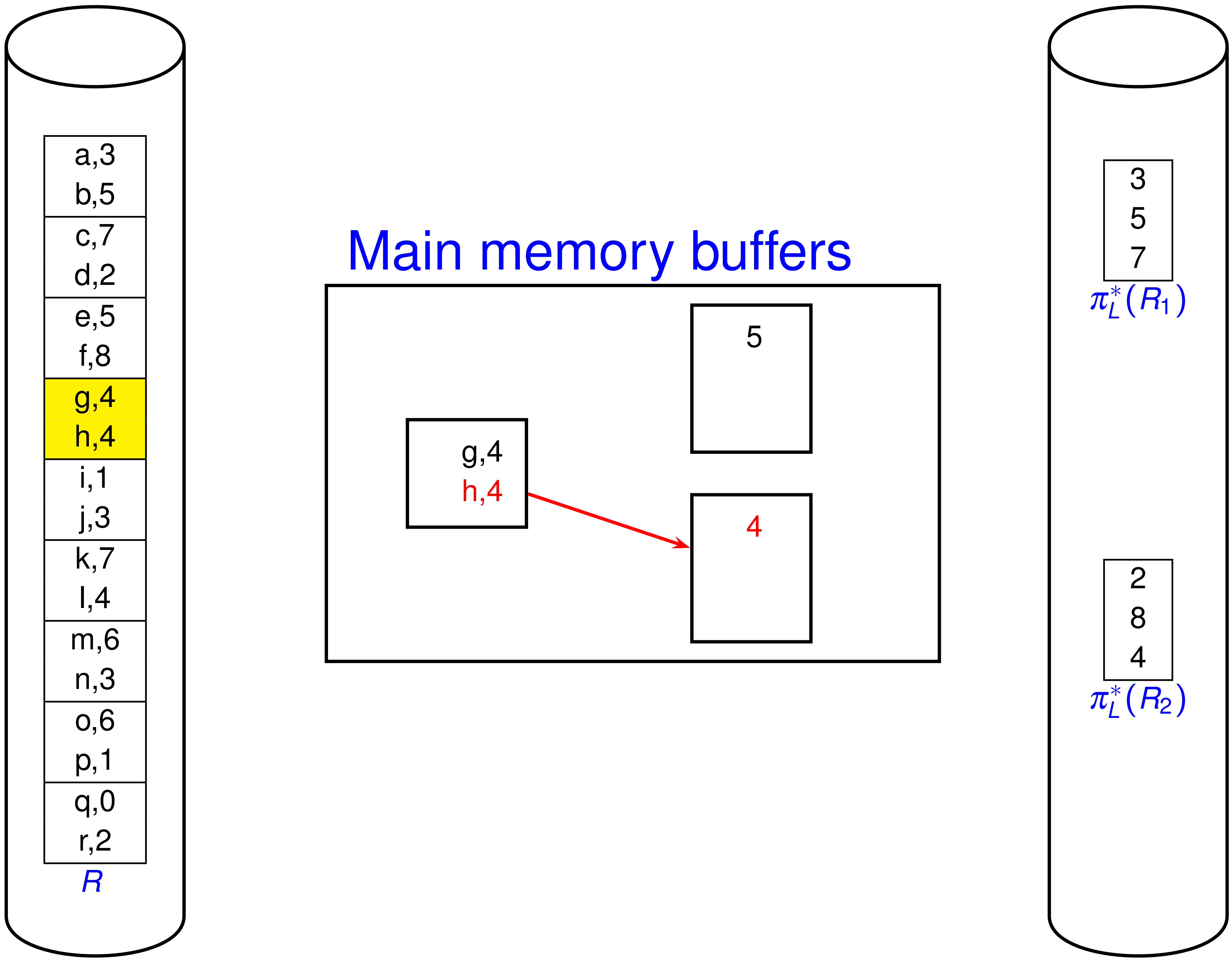
Partitioning Phase



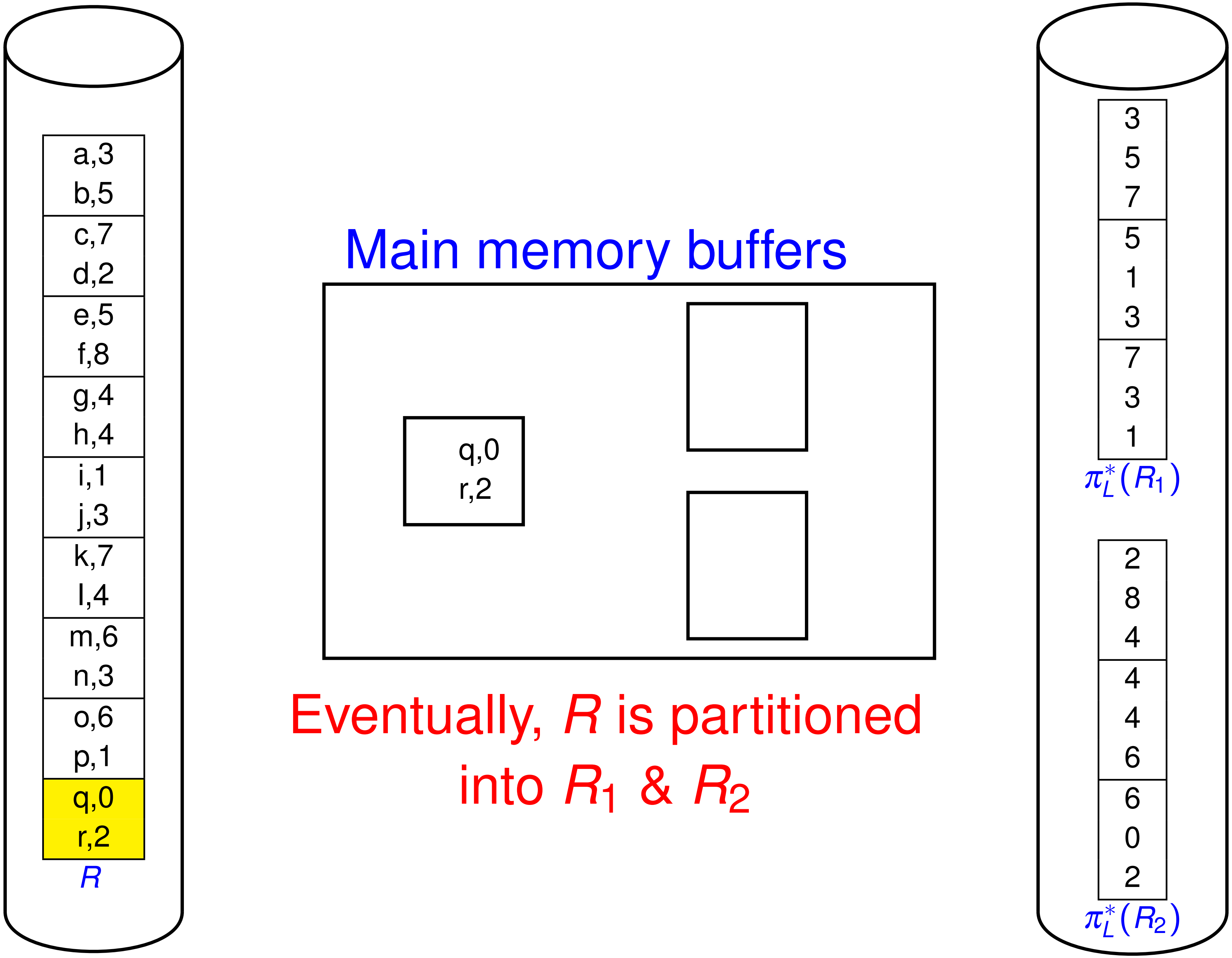
Partitioning Phase



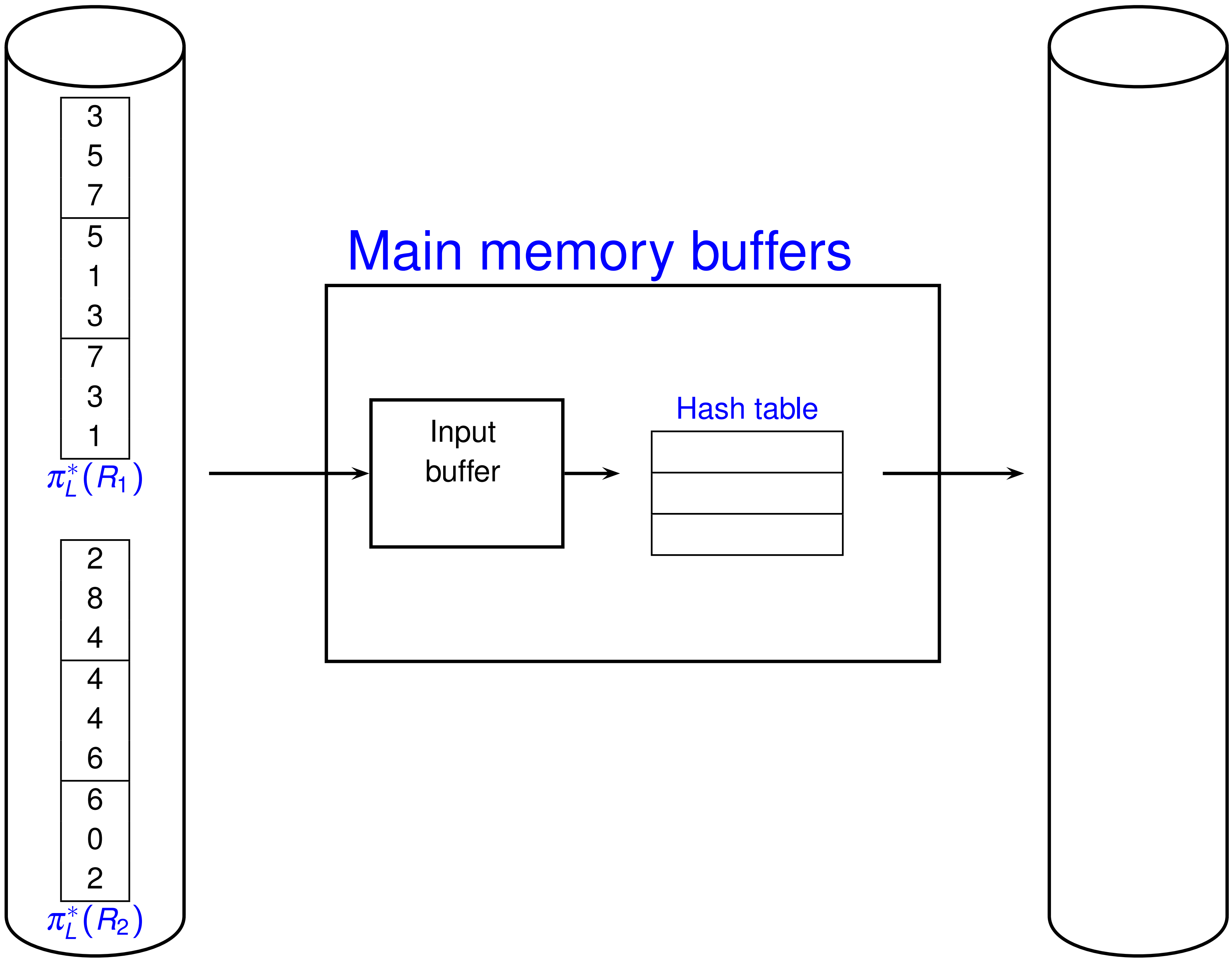
Partitioning Phase



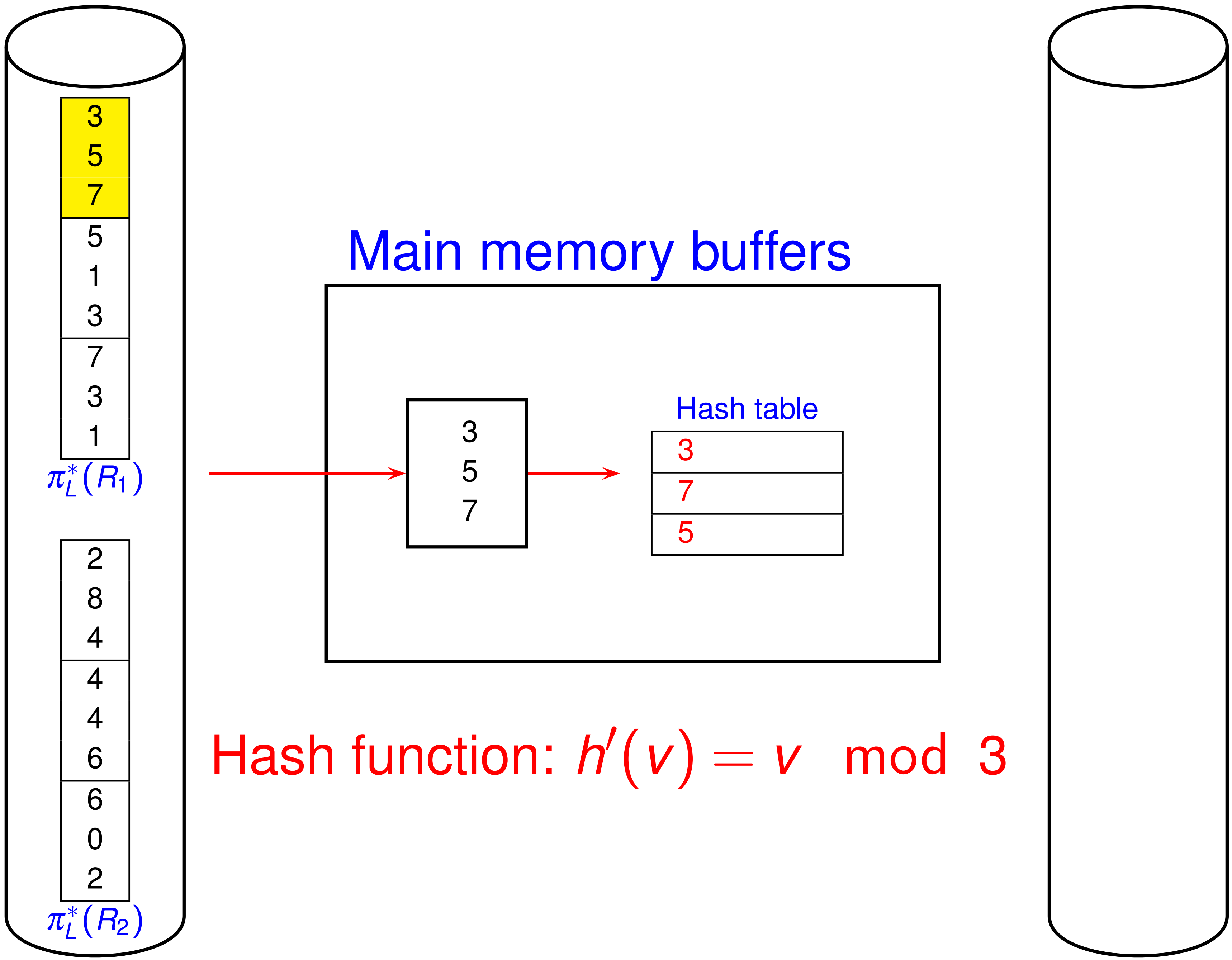
Partitioning Phase



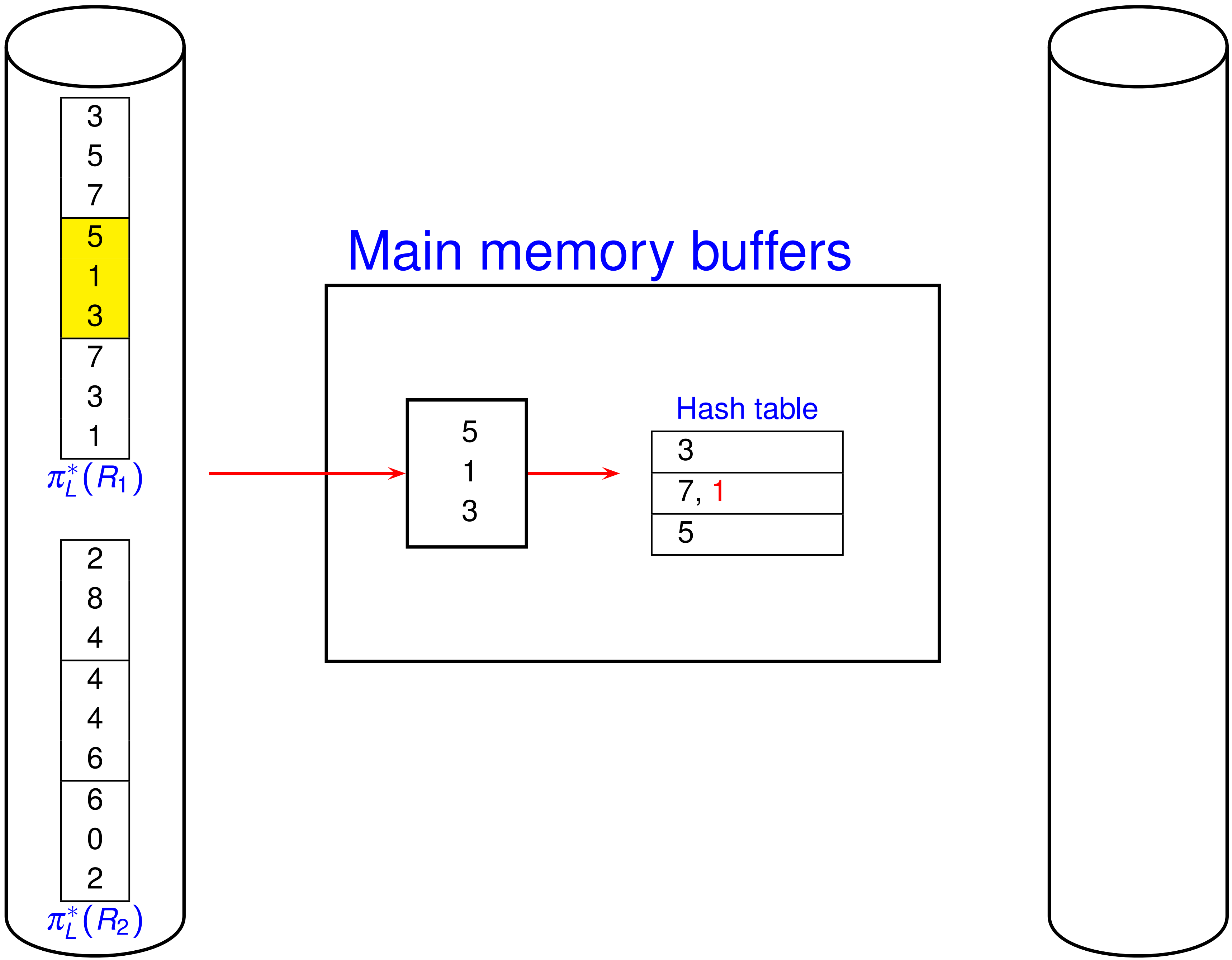
Duplicate Elimination Phase



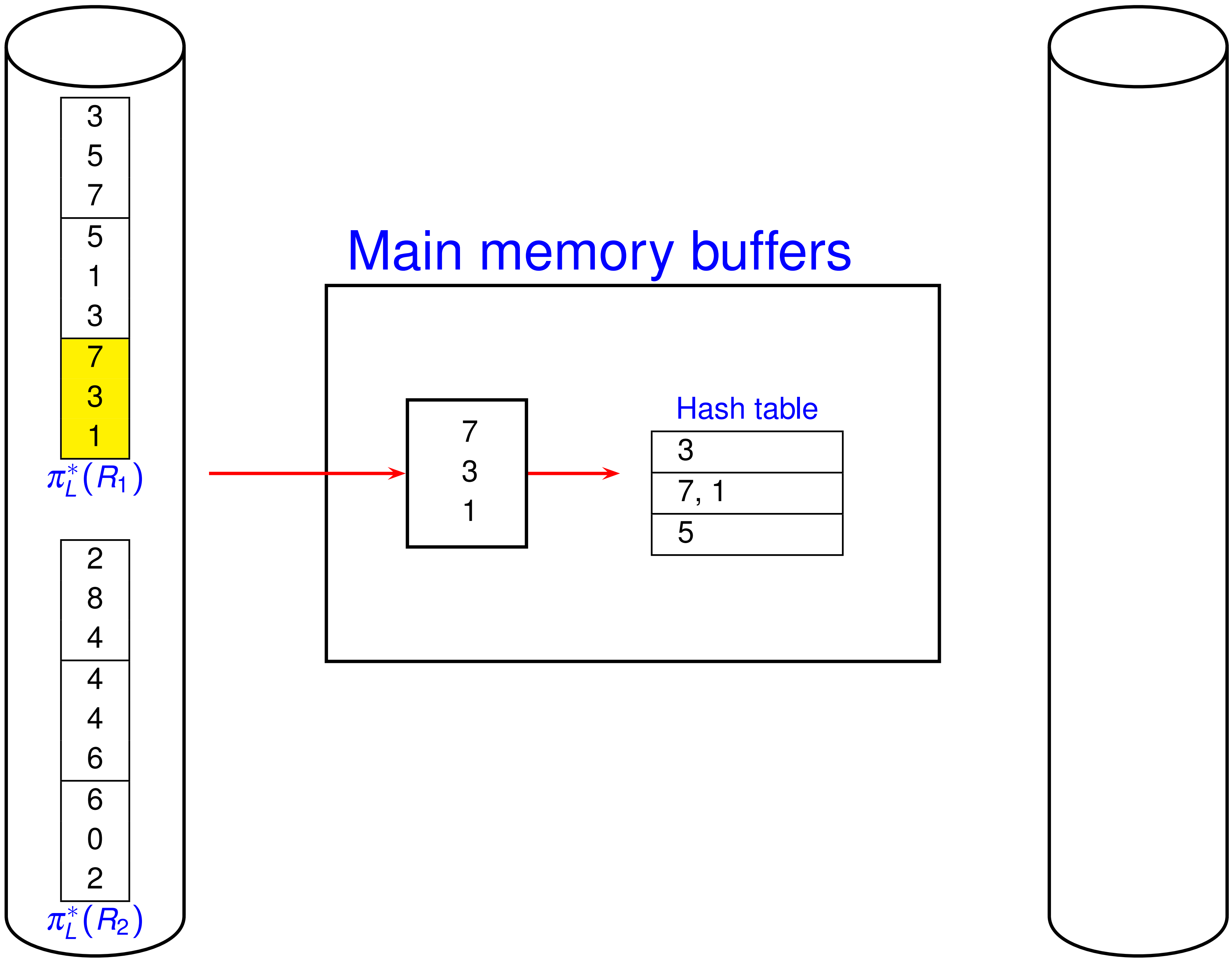
Duplicate Elimination Phase



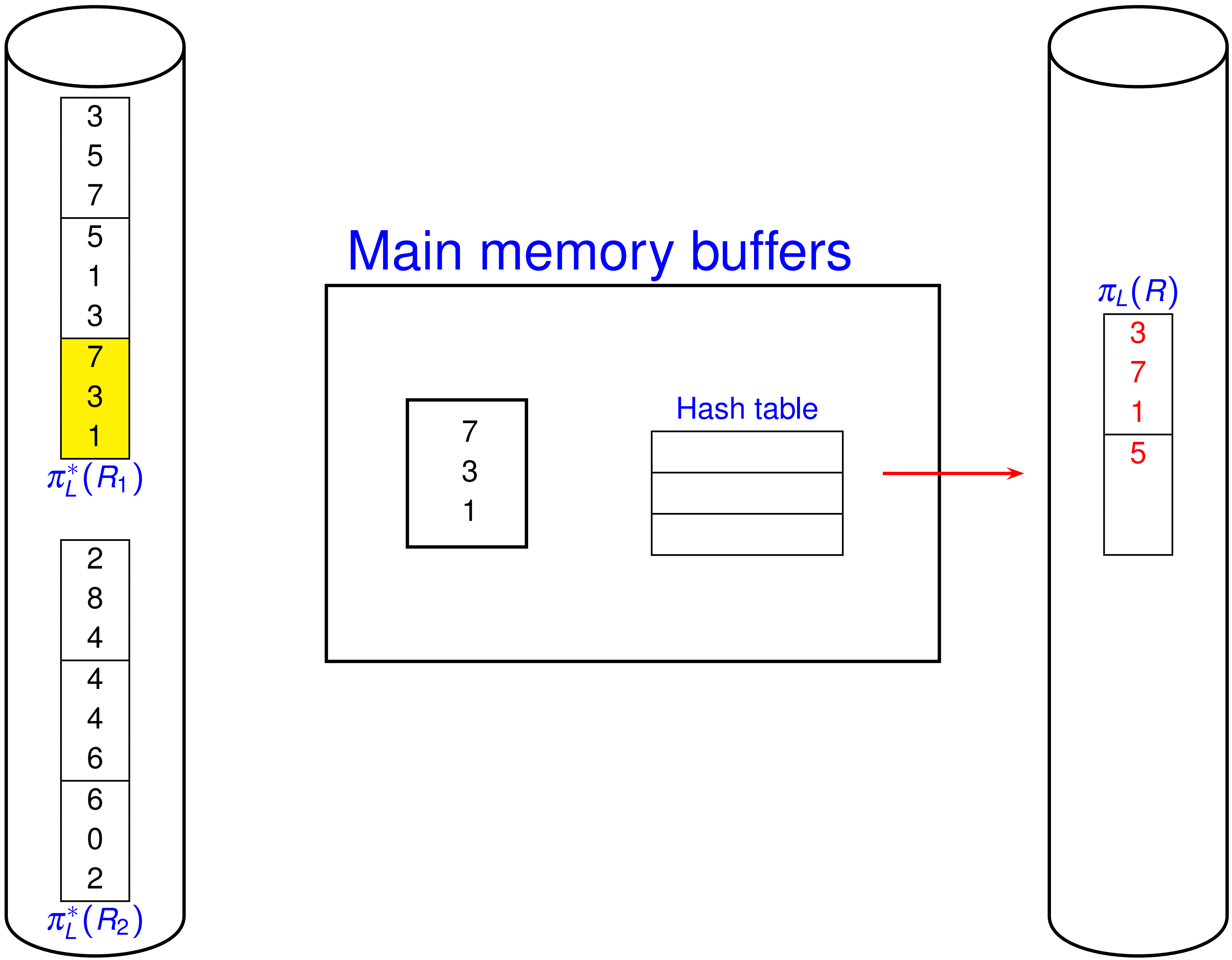
Duplicate Elimination Phase



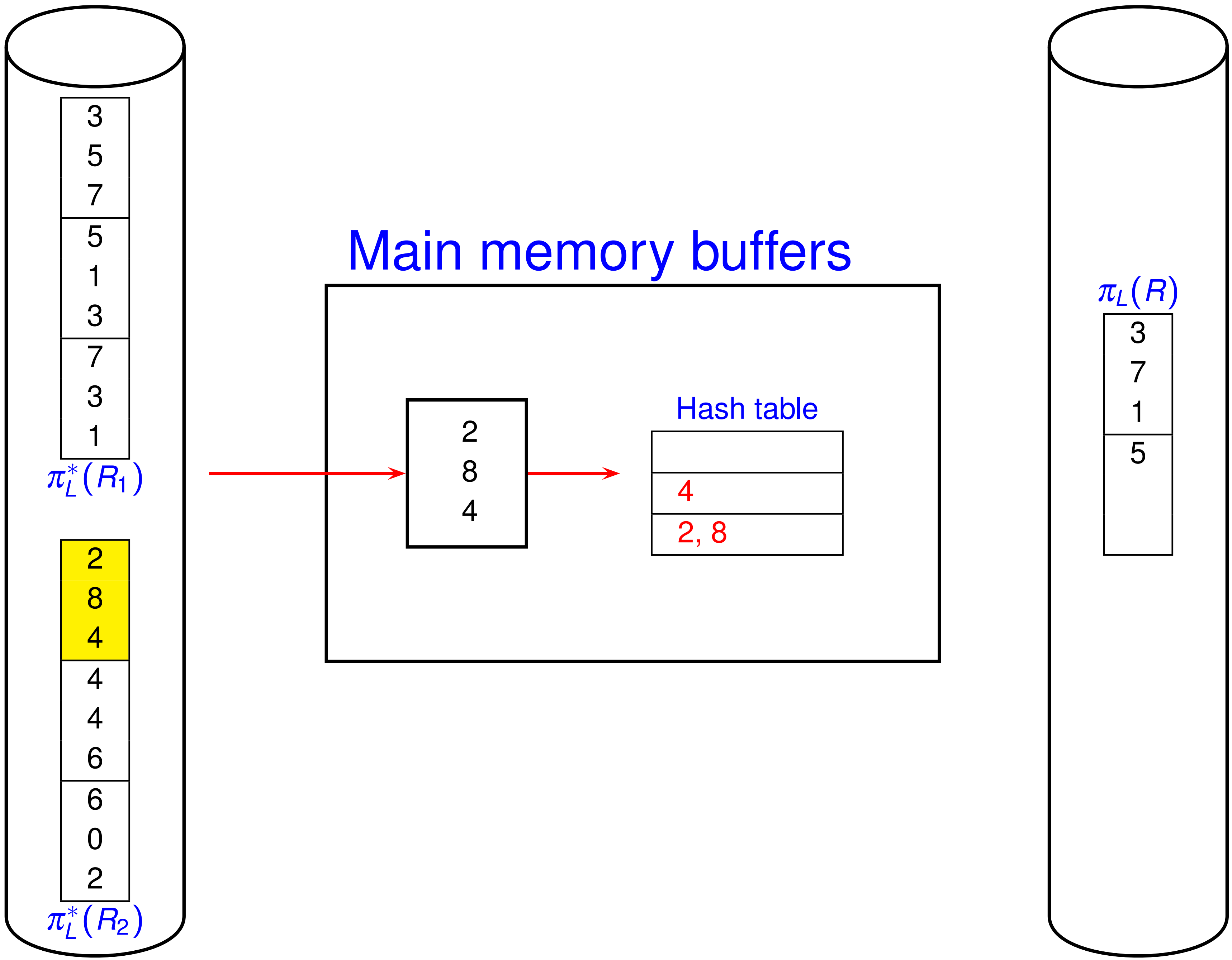
Duplicate Elimination Phase



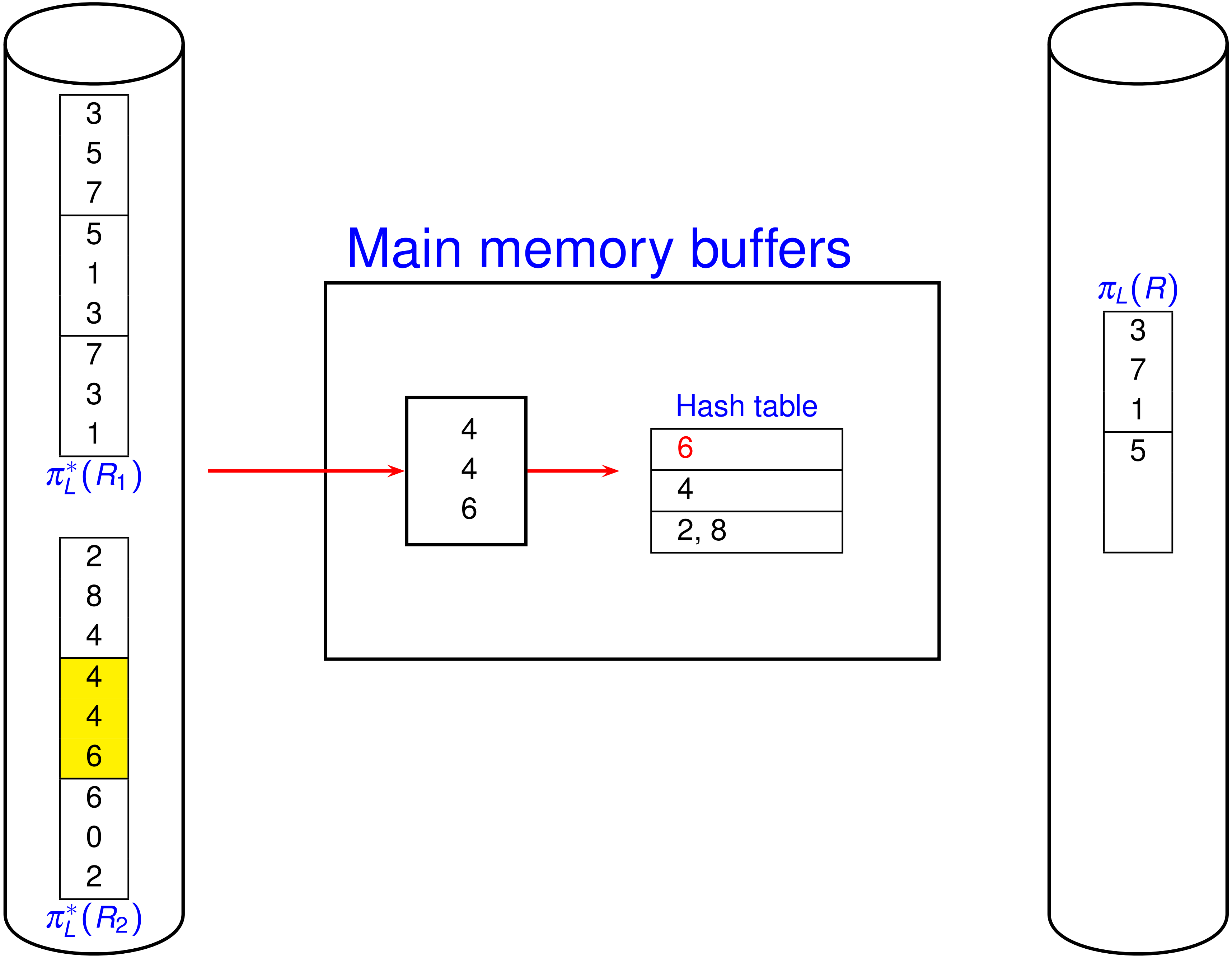
Duplicate Elimination Phase



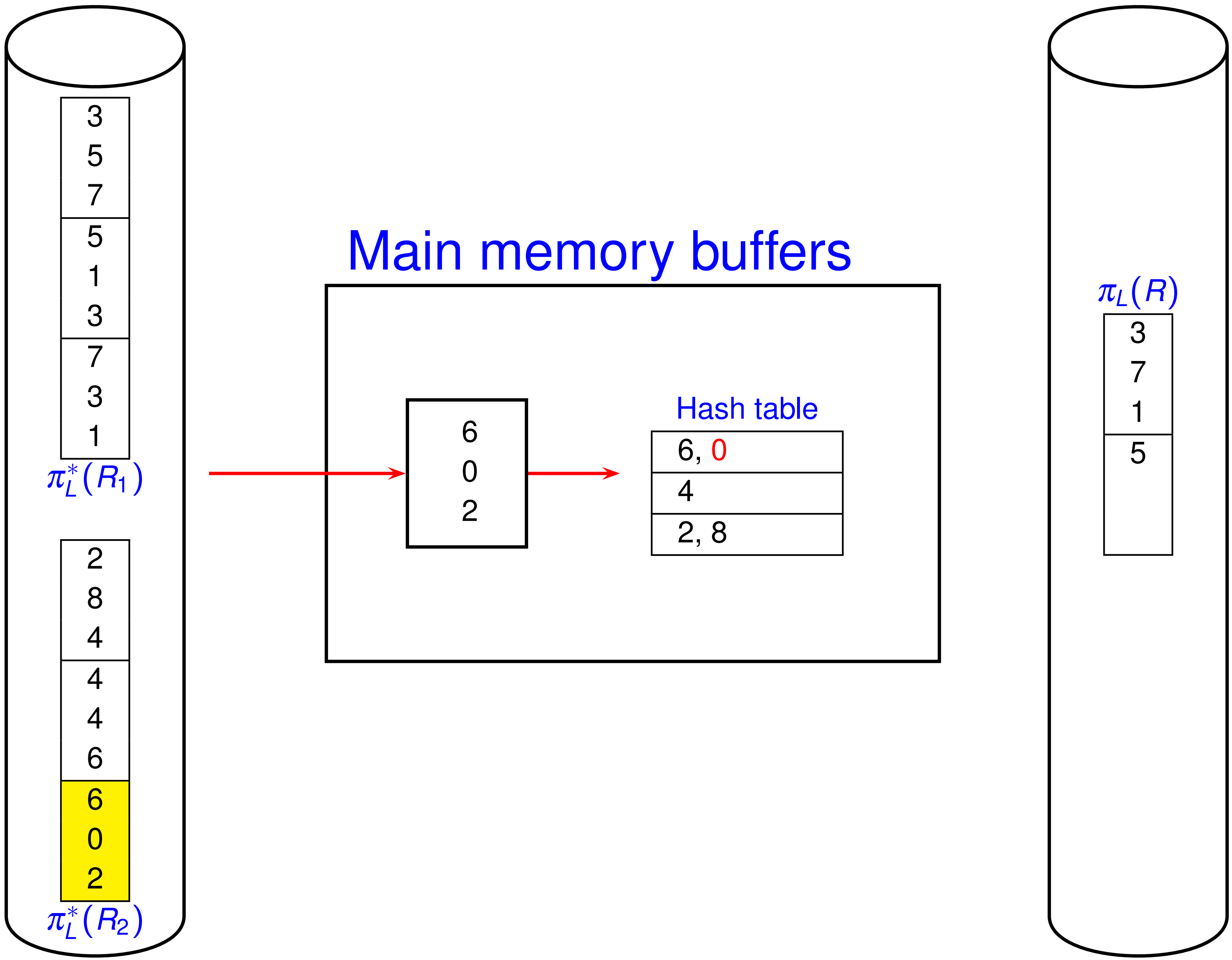
Duplicate Elimination Phase



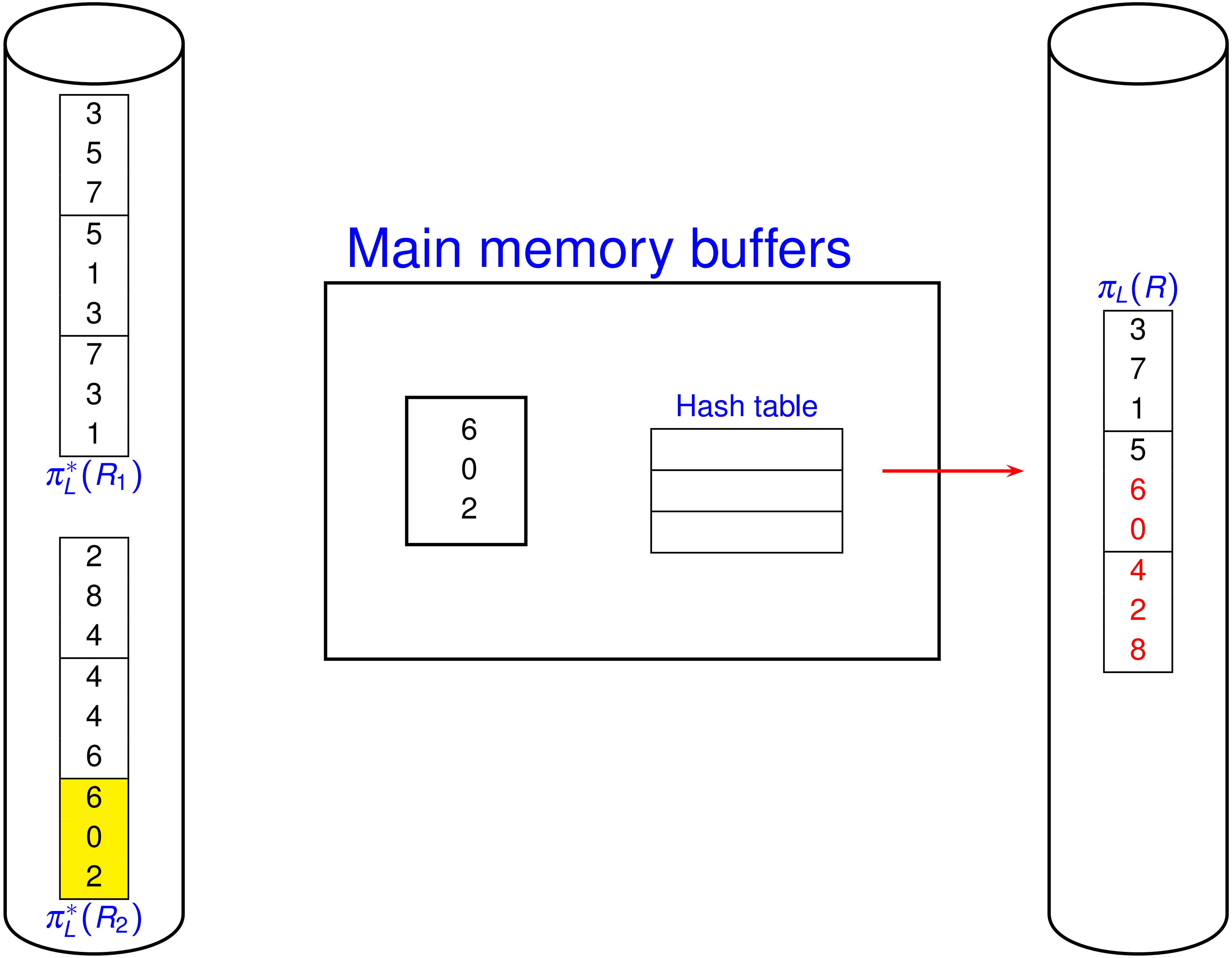
Duplicate Elimination Phase



Duplicate Elimination Phase

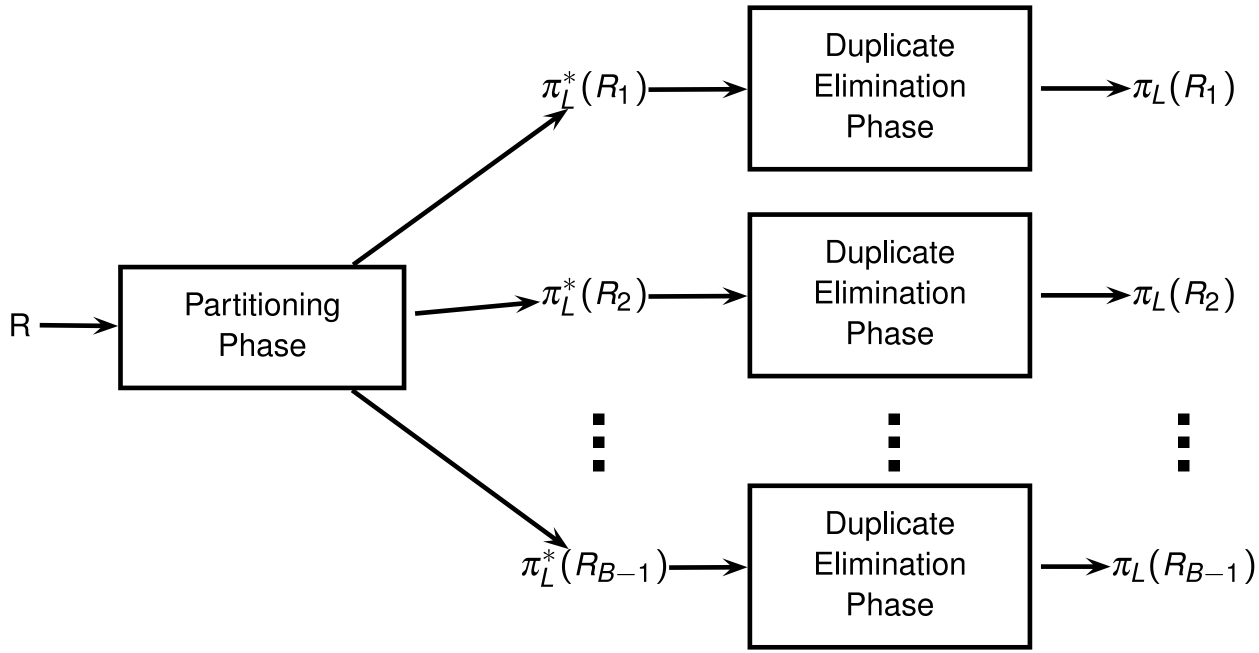


Duplicate Elimination Phase



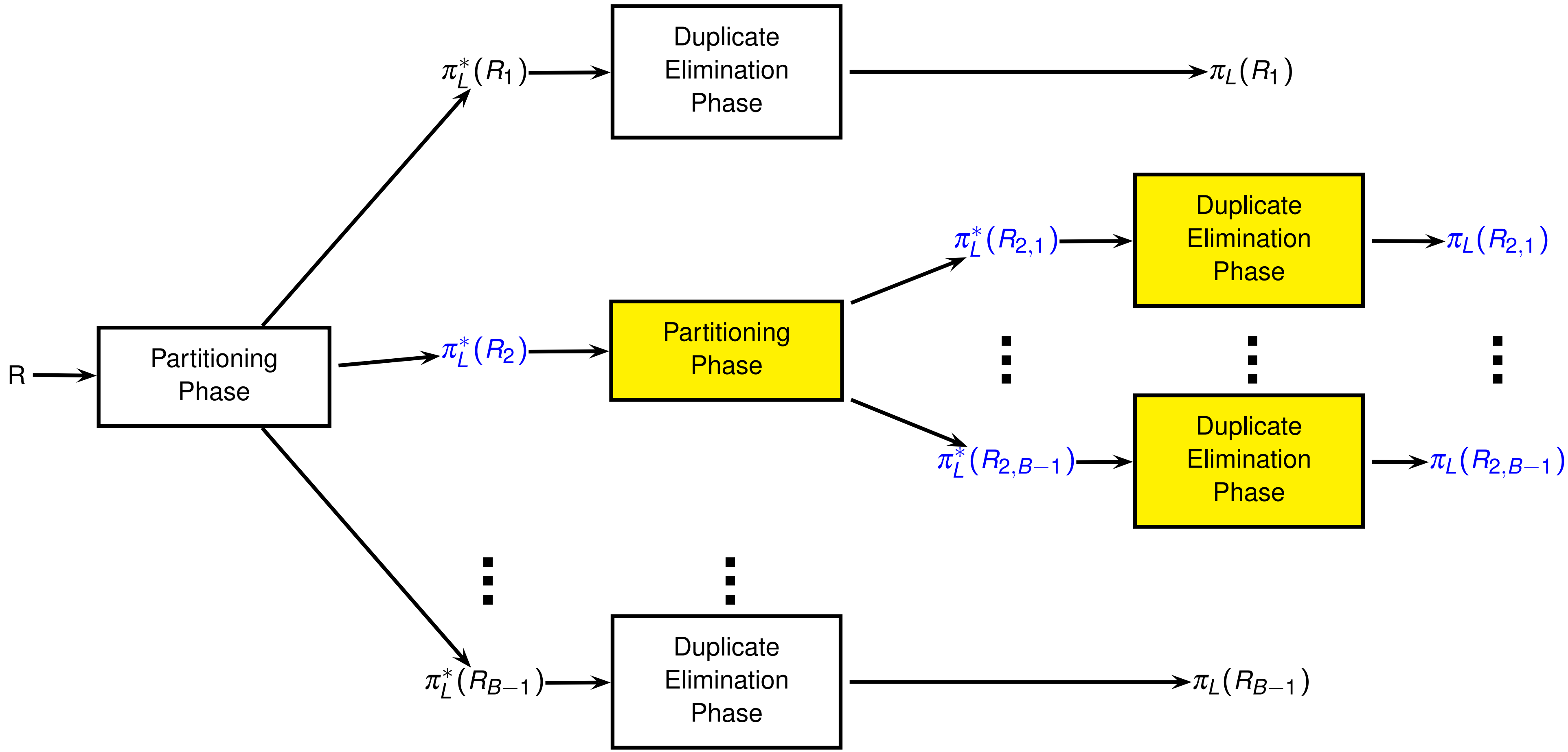
Hash-based Approach: Partition Overflow

- **Partition overflow problem**: Hash table for $\pi_L^*(R_i)$ is larger than available memory buffers
  - Recursively apply hash-based partitioning to the overflowed partition
- **Example**: Without partition overflow



Hash-based Approach: Partition Overflow

- **Example**: Partition R_2 overflows



Hash-based Approach: Analysis

- Approach is effective if B is large relative to $|R|$
- How large should B be?
  - Assume that h distributes tuples in R uniformly
  - Each R_i has $\frac{|\pi_L^*(R)|}{B-1}$ pages
  - Size of hash table for each R_i = $\frac{|\pi_L^*(R)|}{B-1} \times f$
    - ★ *f = fudge factor*
  - Therefore, to avoid partition overflow, $B > \frac{|\pi_L^*(R)|}{B-1} \times f$
    - ★ Approximately, $B > \sqrt{f \times |\pi_L^*(R)|}$
- Analysis: Assume there's no partition overflow
  - Cost of partitioning phase: $|R| + |\pi_L^*(R)|$
  - Cost of duplicate elimination phase: $|\pi_L^*(R)|$
  - Total cost = $|R| + 2|\pi_L^*(R)|$

Sort-based vs Hash-based

- Hash-based
  - Cost = $\underbrace{|R| + |\pi_L^*(R)|}_{\text{partitioning phase}} + \underbrace{|\pi_L^*(R)|}_{\text{duplicate elimination phase}}$
- Sort-based
  - Output is sorted
  - Good if there are many duplicates or if distribution of hashed values are non-uniform
  - If $B > \sqrt{|\pi_L^*(R)|}$,
    - ★ Number of initial sorted runs $N_0 = \lceil \frac{|R|}{B} \rceil \approx \sqrt{|\pi_L^*(R)|}$
    - ★ Number of merging passes = $log_{B-1}(N_0) \approx 1$
    - ★ Sort-based approach requires 2 passes for sorting
    - ★ Both hash-based & sort-based methods have same I/O cost

Projection Operation: Using Indexes

- If there is an index whose search key contains all the wanted attributes,
  - replace table scan with index scan!
- If index is ordered (e.g., B^+-tree) whose search key includes wanted attributes as a prefix,
  - scan data entries in order
  - compare adjacent data entries for duplicates
  - **Example**:
    - Use B^+-tree index on R with key (A, B) to evaluate query $\pi_A(R)$

Join: $R \bowtie_{\theta} S$

- **Database Schema**:
  - Employee (<u>eid</u>, ename, city, did)
  - Department (<u>did</u>, dname, city, managerId)

- **Example 1**: Find (eid, managerId) pairs where managerId is the manager of eid

$$\pi_{\text{eid,managerId}} \text{ (Employee } \bowtie_{\theta} \text{ Department)}$$
$$\text{where } \theta\text{: Employee.did = Department.did}$$

- **Example 2**: Find (eid, did) pairs where eid and did are co-located in the same city

$$\pi_{\text{eid,did}} \text{ (Employee } \bowtie_{\theta} \text{ Department)}$$
$$\text{where } \theta\text{: Employee.city = Department.city}$$

Join Algorithms

1. Iteration-based
   - block nested loop
2. Index-based
   - index nested loop
3. Partition-based
   - sort-merge join
   - hash join

Join Algorithms (cont.)

- Things to consider when choosing an algorithm:
  - Types of join predicates:
    - equality predicates (e.g. $R.A_i = S.B_j$)
    - inequality predicates: (e.g., $R.A_i < S.B_j$)
  - Sizes of join operands
  - Available buffer space
  - Available access methods
- Given a join $R \bowtie_\theta S$
  - R is referred to as the outer relation
  - S is referred to as the inner relation

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result

5, 10
2, 10
13, 7

R

4, 10
5, 10

S

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result

R
$\boxed{5}$, 10
2, 10
13, 7

S
$\boxed{4}$, 10
5, 10

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result

R
$\boxed{5}$, 10
2, 10
13, 7

S
4, $\boxed{10}$
5, 10

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result

5, 10
2, 10
13, 7

R

4, 10
5, 10

S

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

for each tuple $r \in R$ do
    for each tuple $s \in S$ do
        if (r matches s) then
            output (r, s) to result

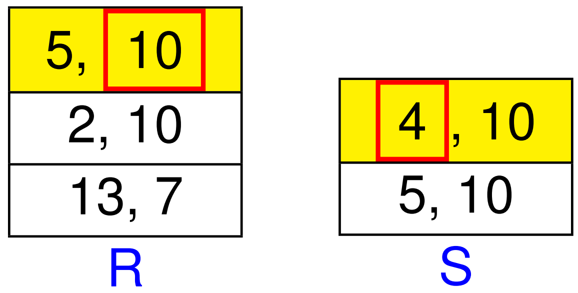
R
5, 10
2, 10
13, 7

S
4, 10
5, 10

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

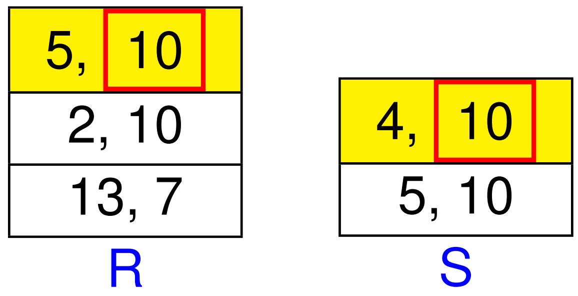
  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result



Tuple-based Nested Loop Join

- Tuple-based Algorithm:

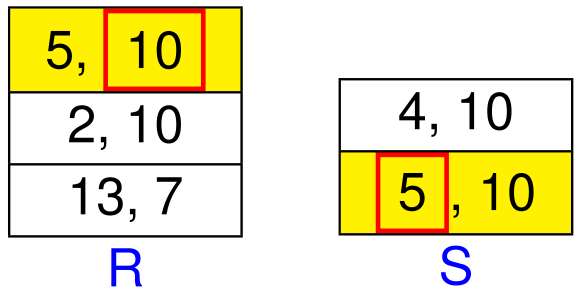
  for each tuple $r \in R$ do
  for each tuple $s \in S$ do
  if (r matches s) then
  output (r, s) to result



Tuple-based Nested Loop Join

- Tuple-based Algorithm:

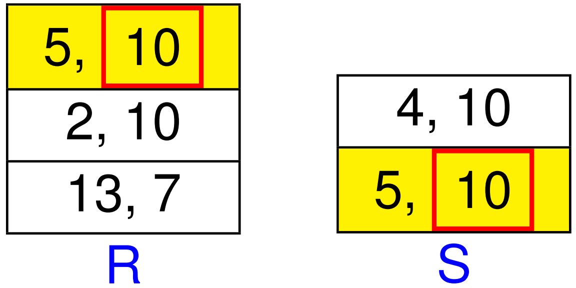
  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result



Tuple-based Nested Loop Join

- Tuple-based Algorithm:

for each tuple $r \in R$ do
  for each tuple $s \in S$ do
    if (r matches s) then
      output (r, s) to result



Tuple-based Nested Loop Join

- Tuple-based Algorithm:

for each tuple $r \in R$ do
    for each tuple $s \in S$ do
        if (r matches s) then
            output (r, s) to result

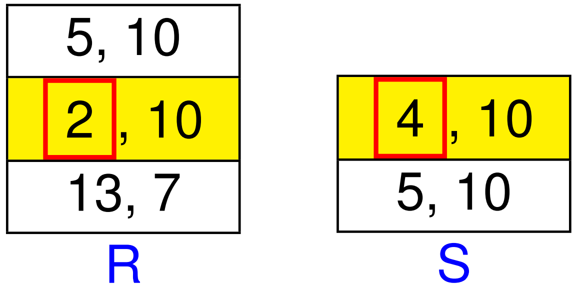
R
5, 10
$\boxed{2}$, 10
13, 7

S
$\boxed{4}$, 10
5, 10

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

for each tuple $r \in R$ do
  for each tuple $s \in S$ do
    if (r matches s) then
      output (r, s) to result

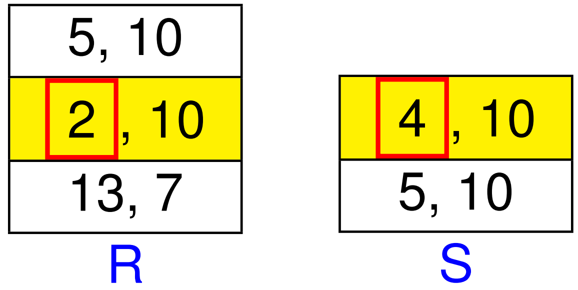


etc ...

Tuple-based Nested Loop Join

- Tuple-based Algorithm:

  for each tuple $r \in R$ do
      for each tuple $s \in S$ do
          if (r matches s) then
              output (r, s) to result



etc ...

- I/O Cost Analysis:

$$\underbrace{|R|}_{\text{scan R}} + \underbrace{||R|| \times |S|}_{\text{scan S}}$$

Page-based Nested Loop Join

- Page-based Algorithm:

for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result

5, 10
2, 10
13, 7

R

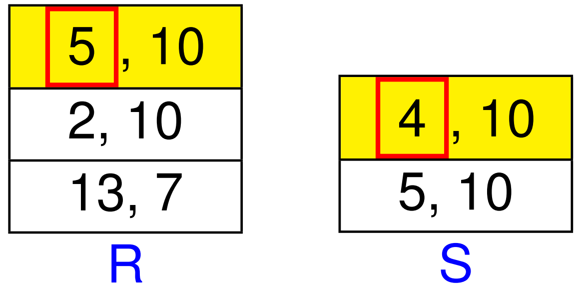
4, 10
5, 10

S

Page-based Nested Loop Join

- Page-based Algorithm:

for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result

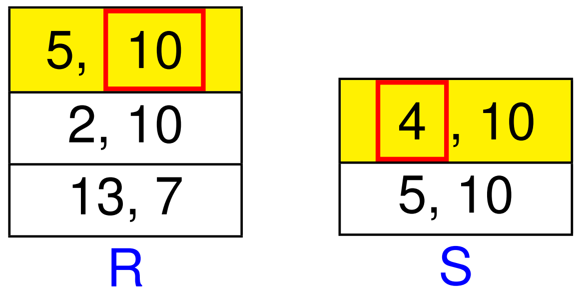
R
$\boxed{5}$, 10
2, 10
13, 7

S
4, $\boxed{10}$
5, 10

Page-based Nested Loop Join

- Page-based Algorithm:

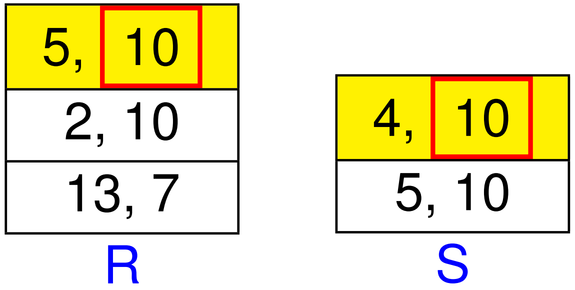
  for each page P_R of R do
      for each page P_S of S do
          for each tuple $r \in P_R$ do
              for each tuple $s \in P_S$ do
                  if (r matches s) then
                      output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

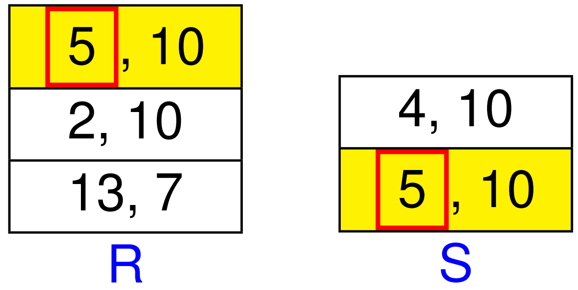
  for each page P_R of R do
      for each page P_S of S do
          for each tuple $r \in P_R$ do
              for each tuple $s \in P_S$ do
                  if (r matches s) then
                      output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

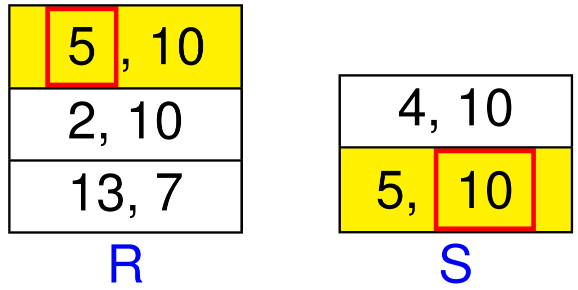
for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

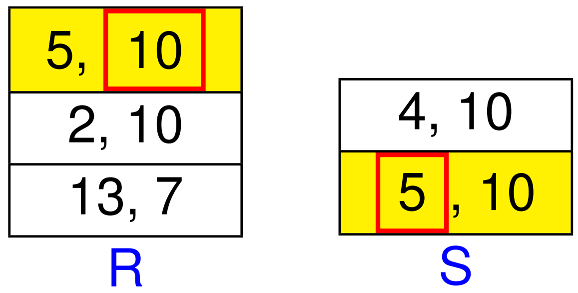
for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

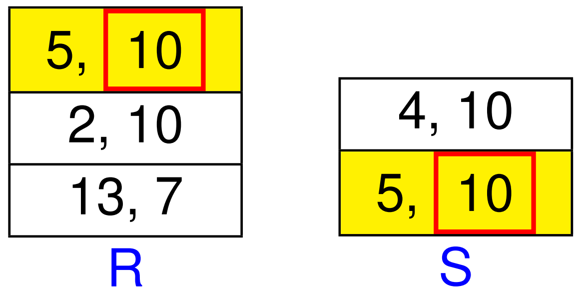
for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

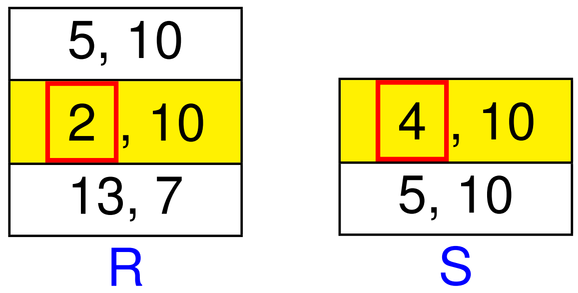
for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

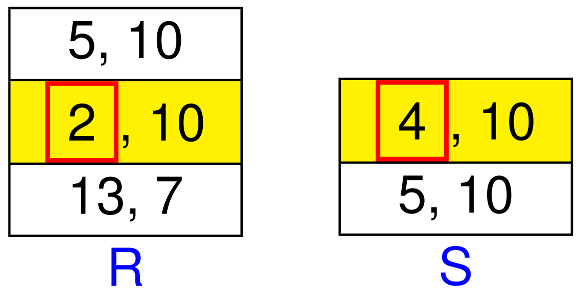
for each page P_R of R do
    for each page P_S of S do
        for each tuple $r \in P_R$ do
            for each tuple $s \in P_S$ do
                if (r matches s) then
                    output (r, s) to result



Page-based Nested Loop Join

- Page-based Algorithm:

for each page P_R of R do
  for each page P_S of S do
    for each tuple $r \in P_R$ do
      for each tuple $s \in P_S$ do
        if (r matches s) then
          output (r, s) to result

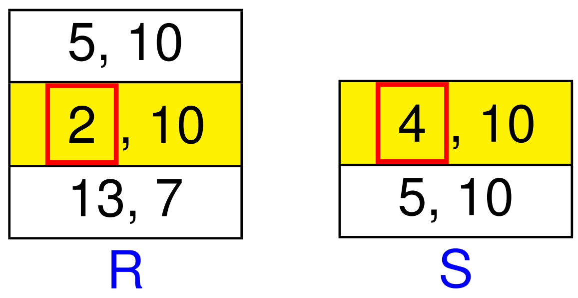


etc ...

Page-based Nested Loop Join

- Page-based Algorithm:

  for each page P_R of R do
      for each page P_S of S do
          for each tuple $r \in P_R$ do
              for each tuple $s \in P_S$ do
                  if (r matches s) then
                      output (r, s) to result



etc ...

- I/O Cost Analysis:

$$\underbrace{|R|}_{\text{scan R}} + \underbrace{|R| \times |S|}_{\text{scan S}}$$

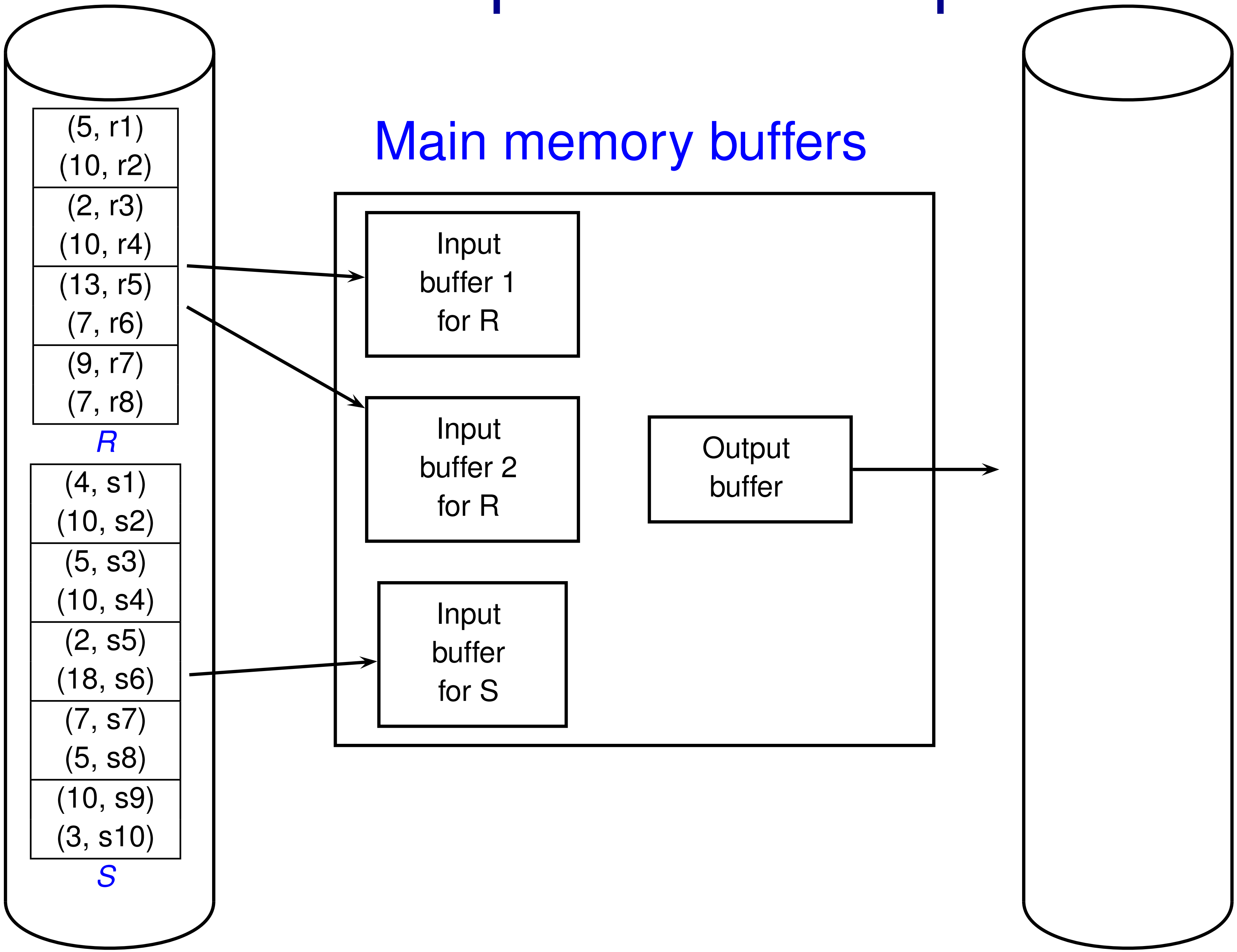
Block Nested Loop Join

- **Motivation**: How to better exploit buffer space to minimize number of I/Os?
- Assume $|R| \leq |S|$
- **Buffer space allocation**: Allocate one page for S, one page for output & remaining pages for R
- **Algorithm (using B buffer pages)**:

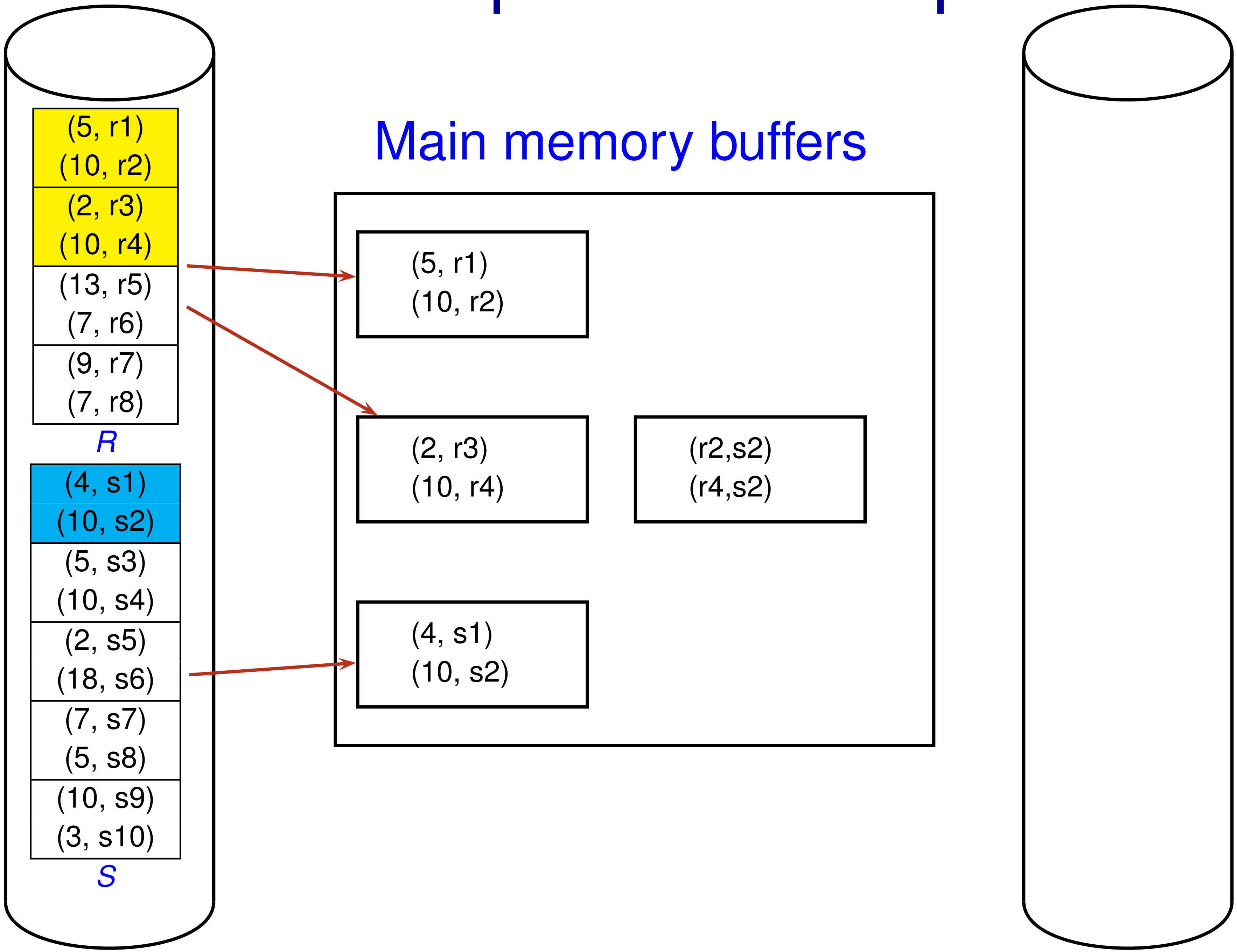
```
while (scan of R is not done) do
    read next (B − 2) pages of R into buffer
    for each page P_S of S do
        read P_S into buffer
        for each tuple r of R in buffer and each tuple s ∈ P_S do
            if (r matches s) then output (r, s) to result
```

- I/O Cost: $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$

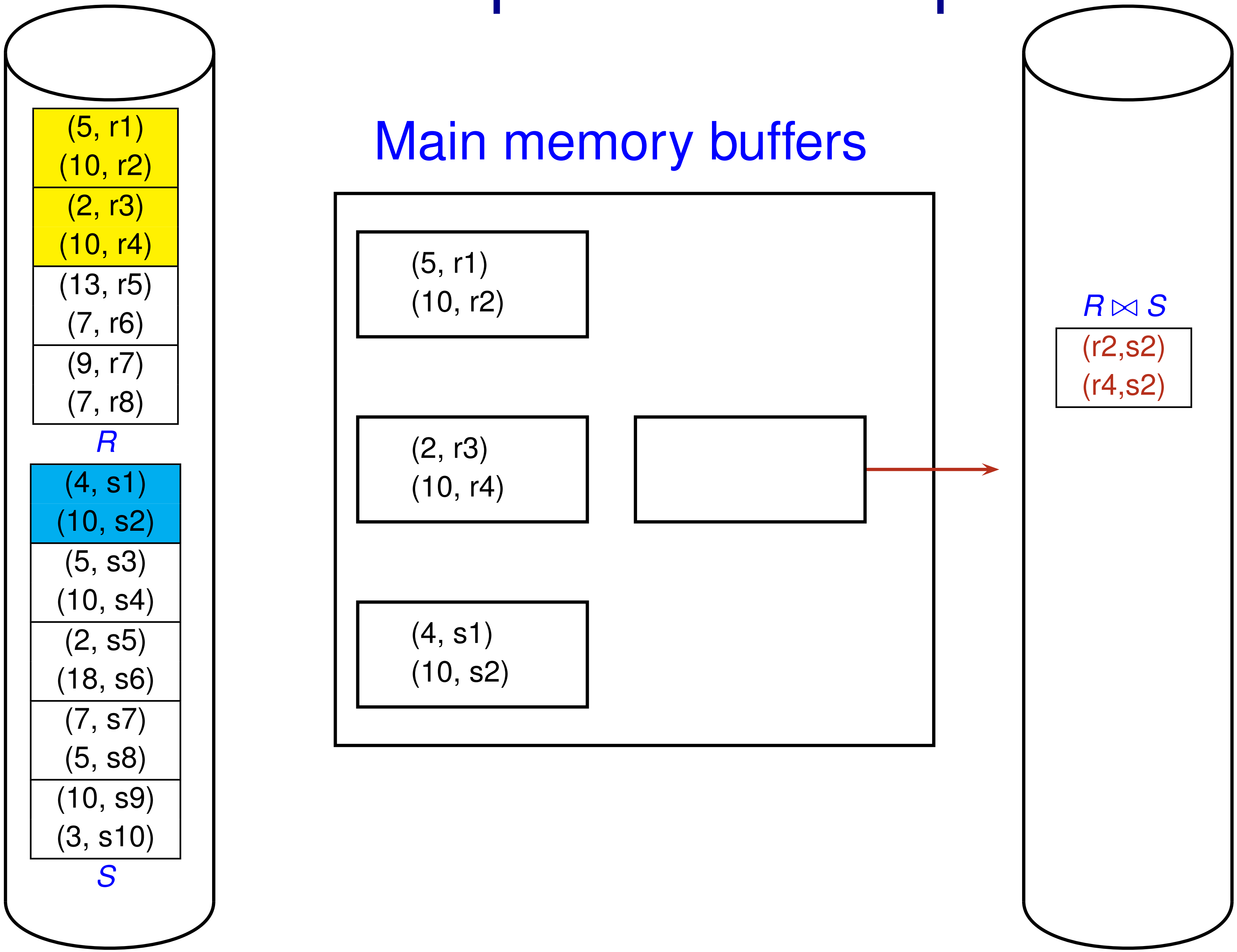
Block Nested Loop Join: Example



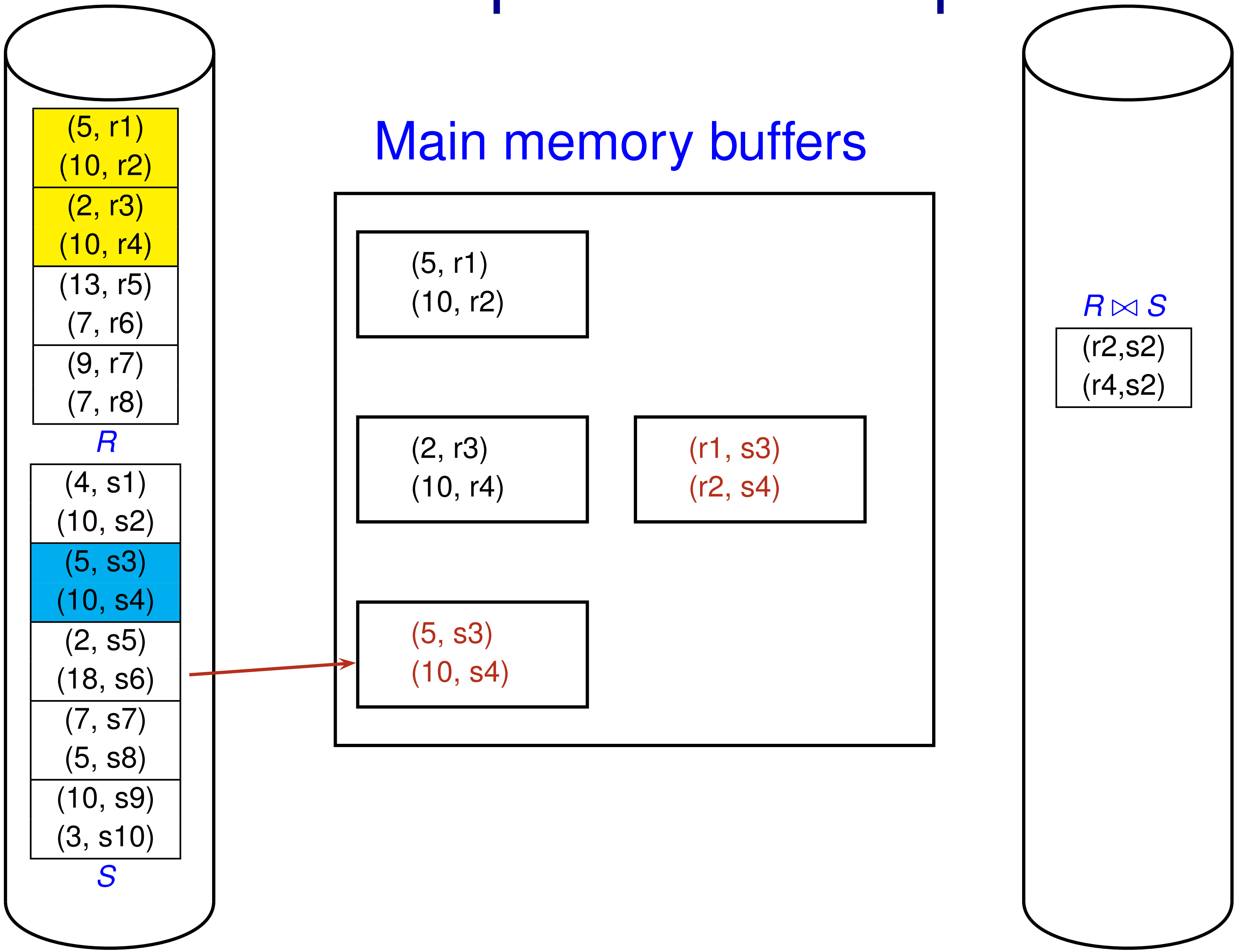
Block Nested Loop Join: Example



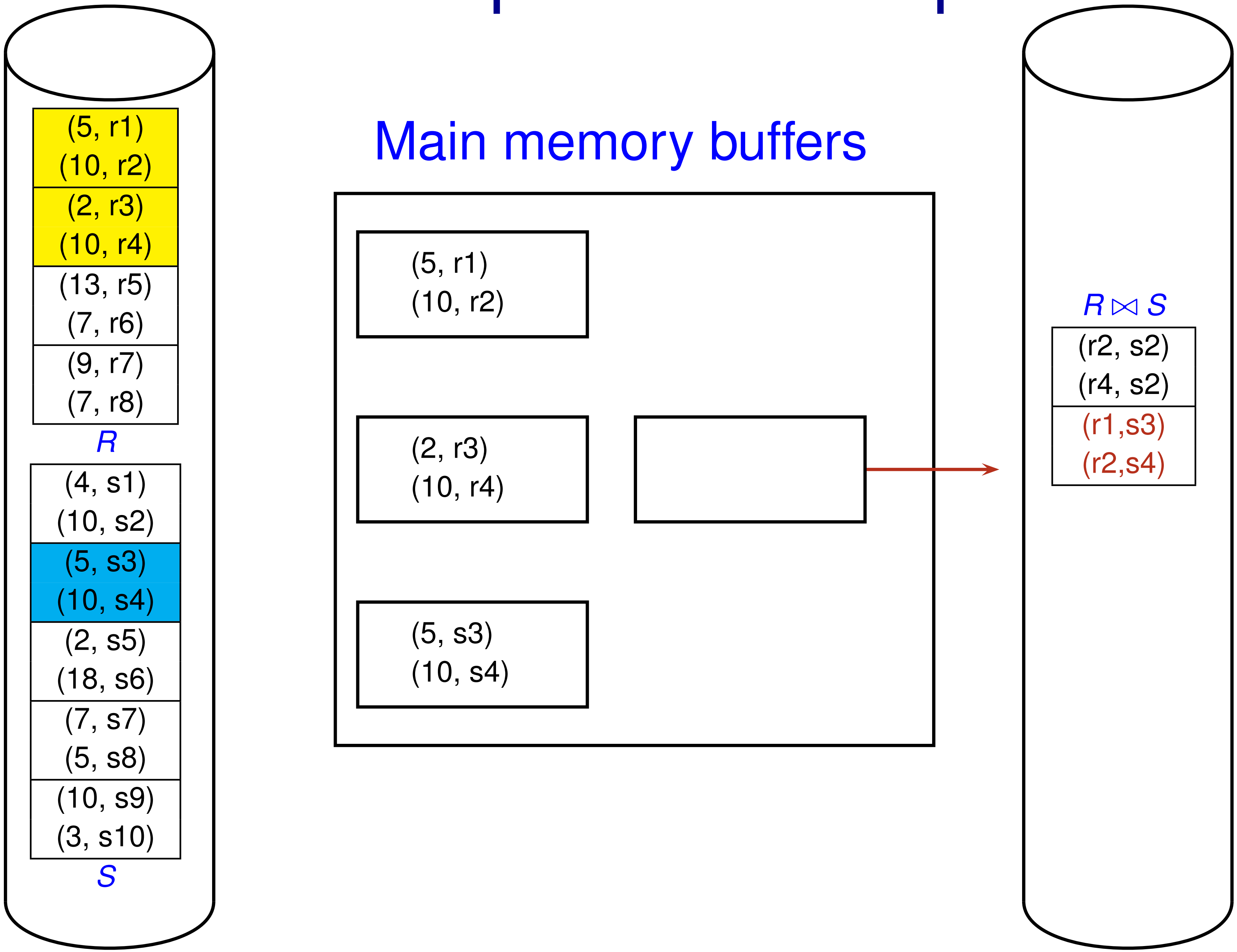
Block Nested Loop Join: Example



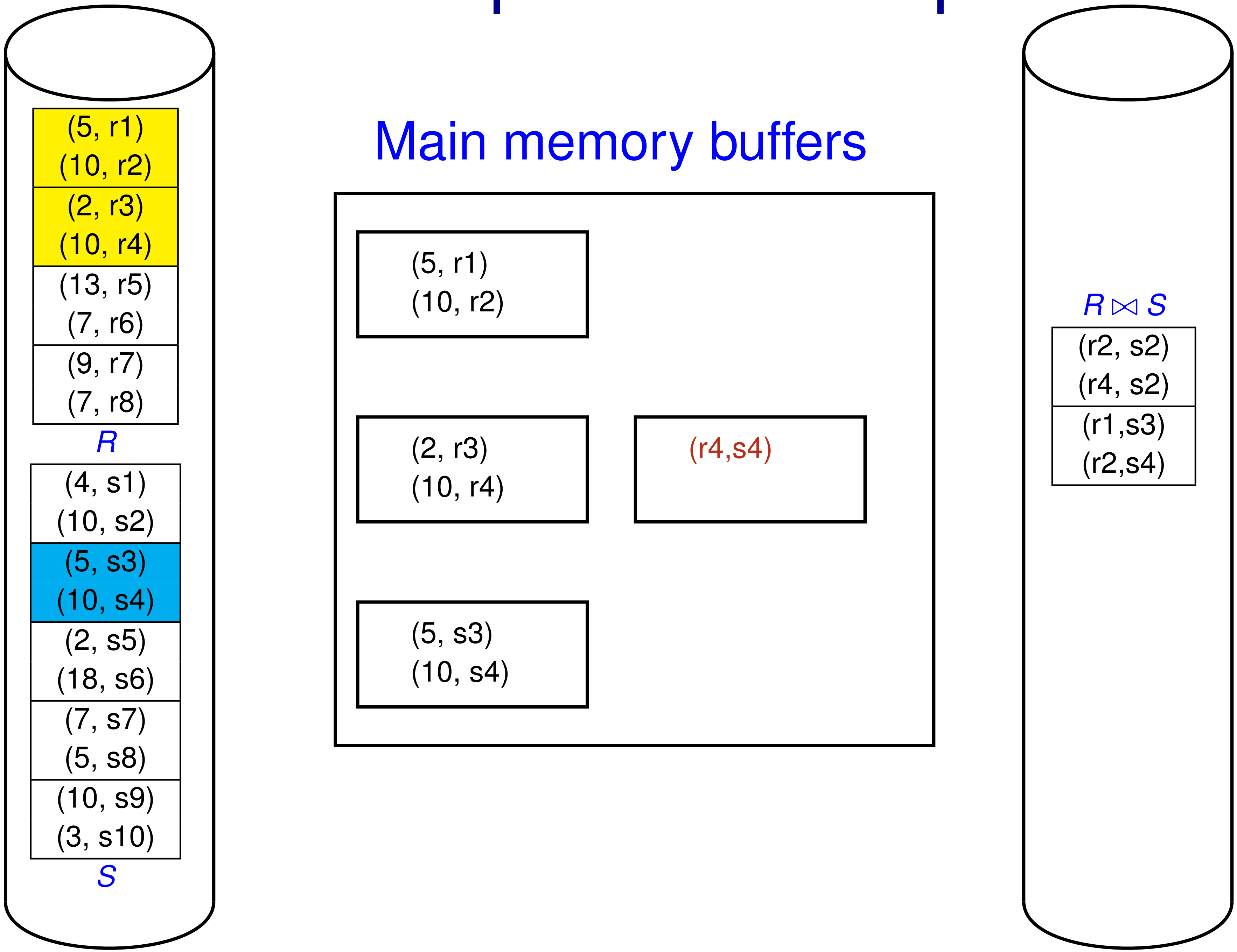
Block Nested Loop Join: Example



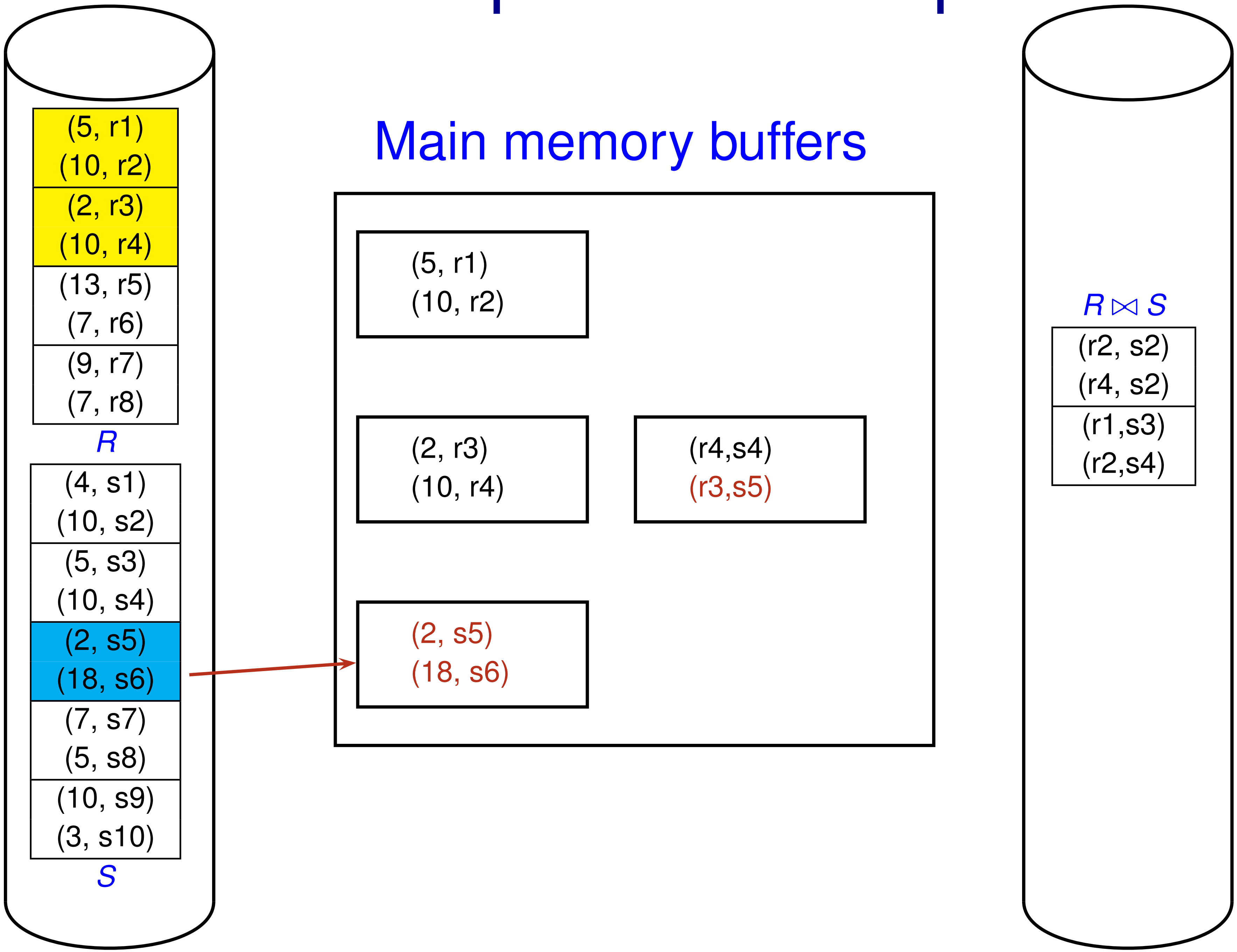
Block Nested Loop Join: Example



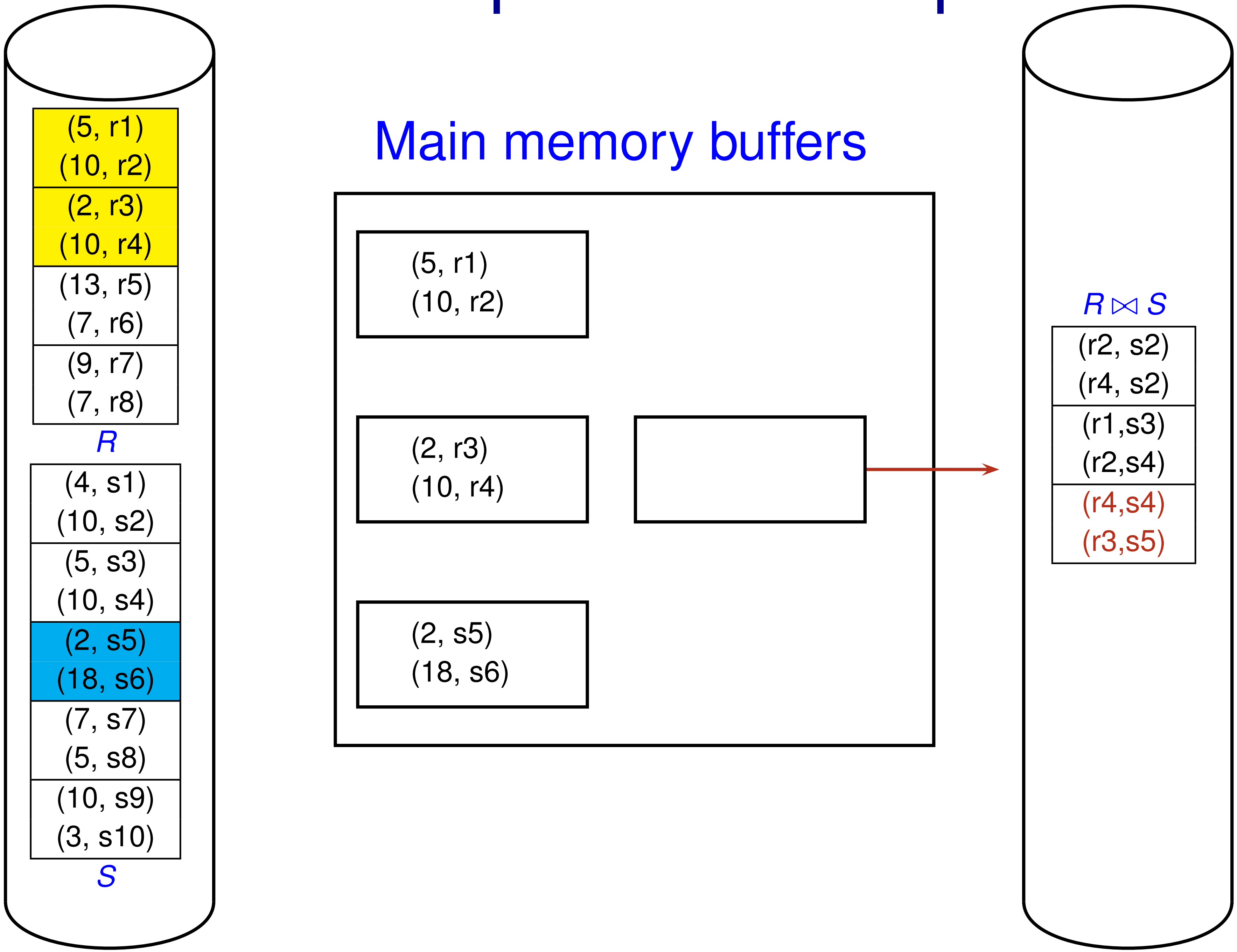
Block Nested Loop Join: Example



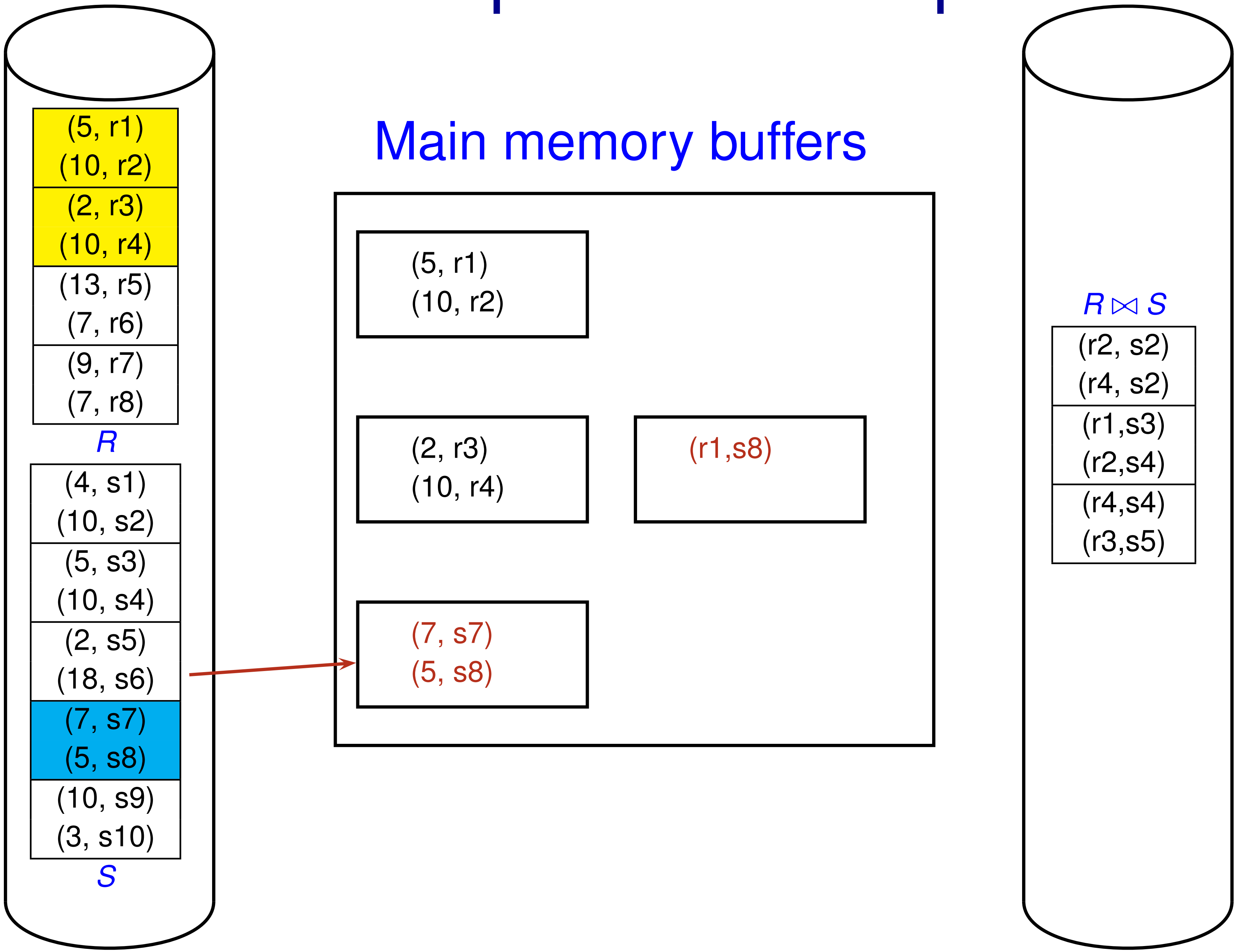
Block Nested Loop Join: Example



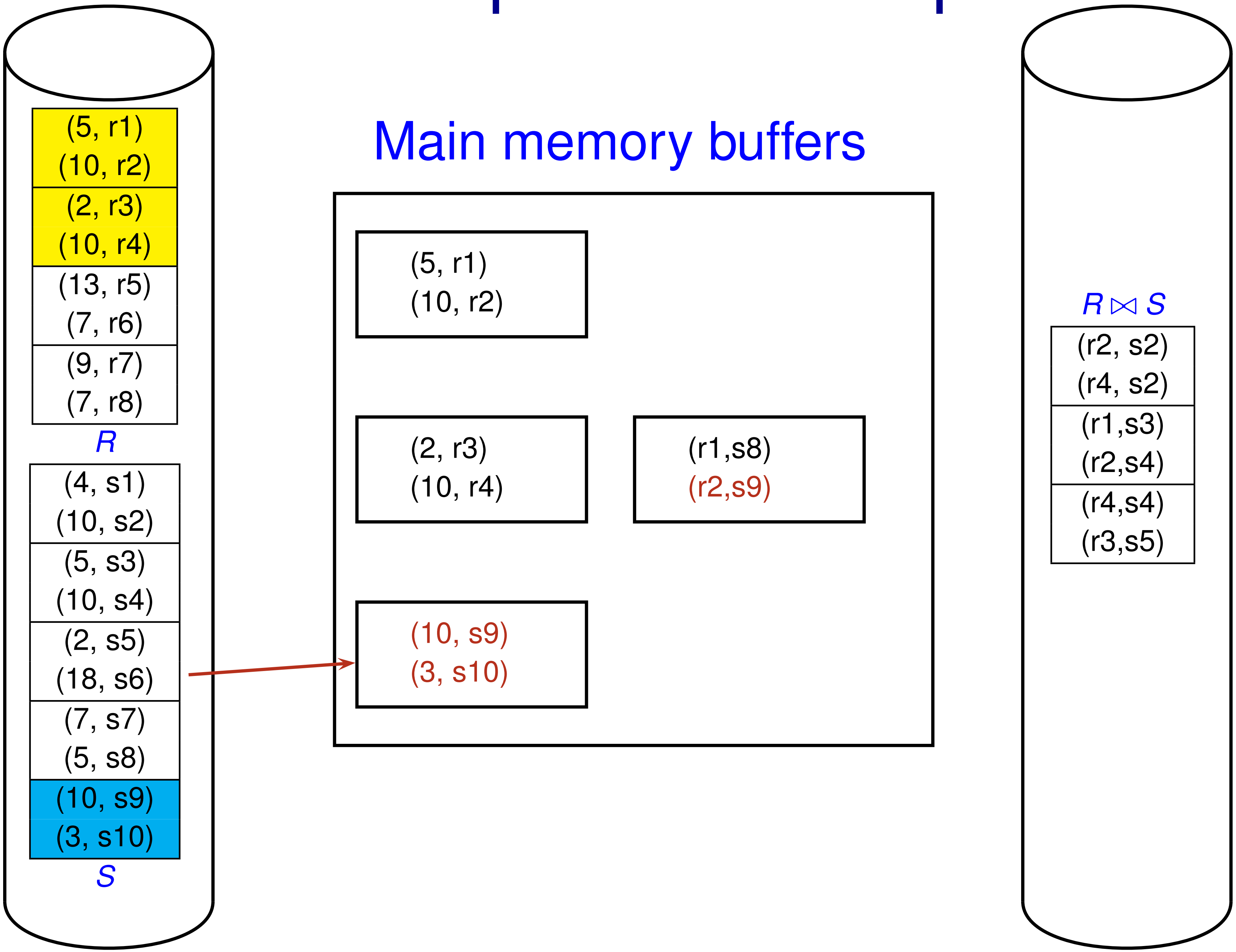
Block Nested Loop Join: Example



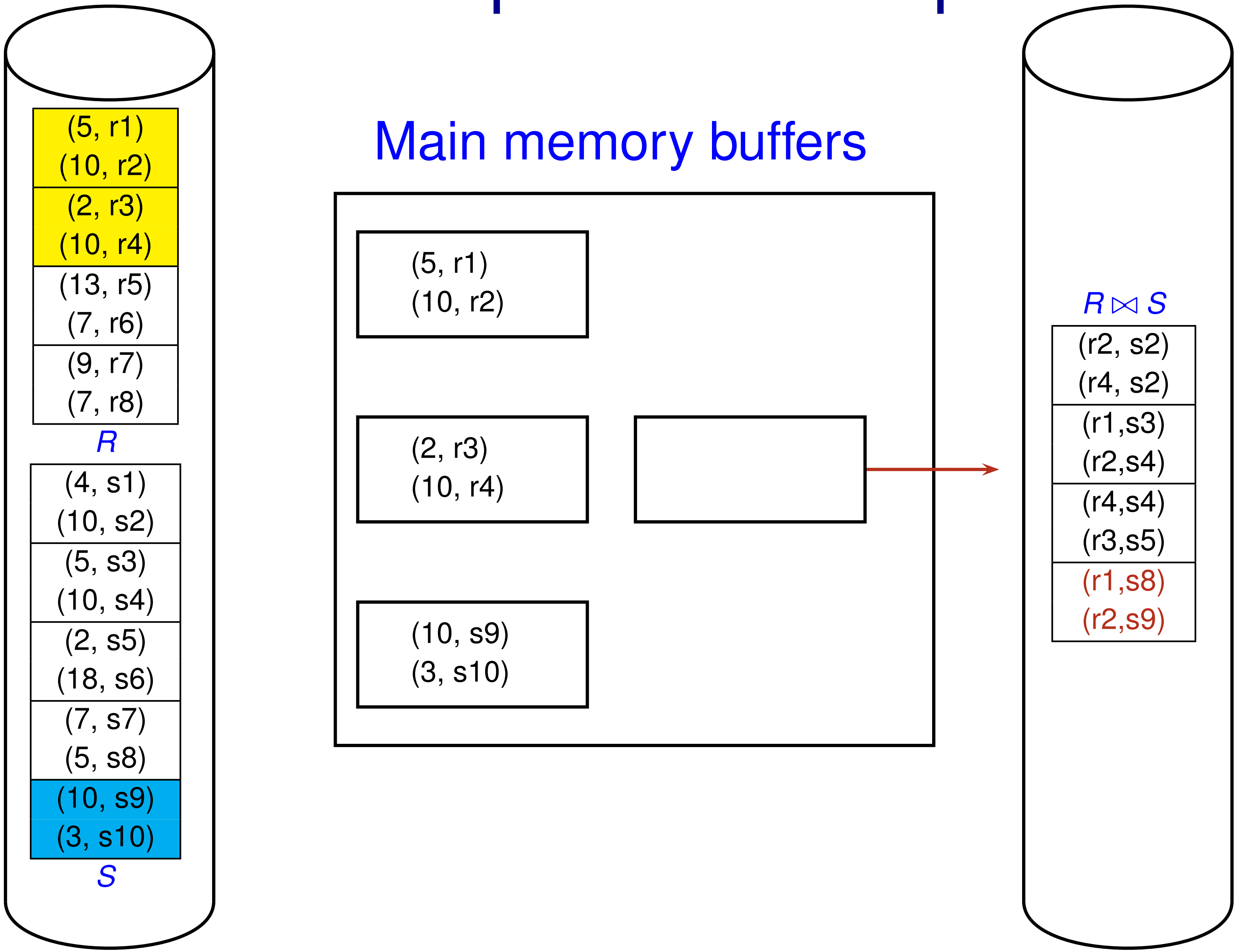
Block Nested Loop Join: Example



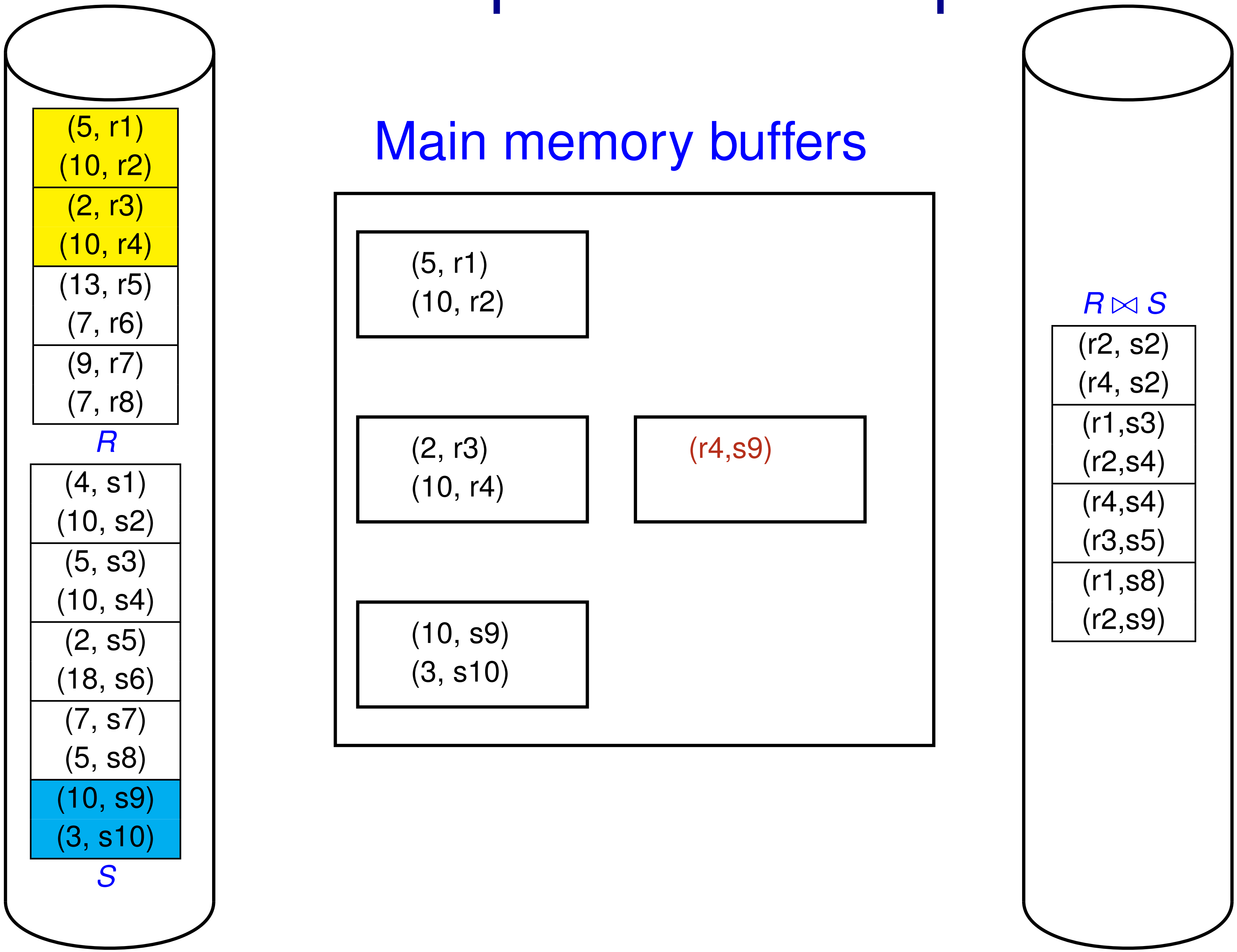
Block Nested Loop Join: Example



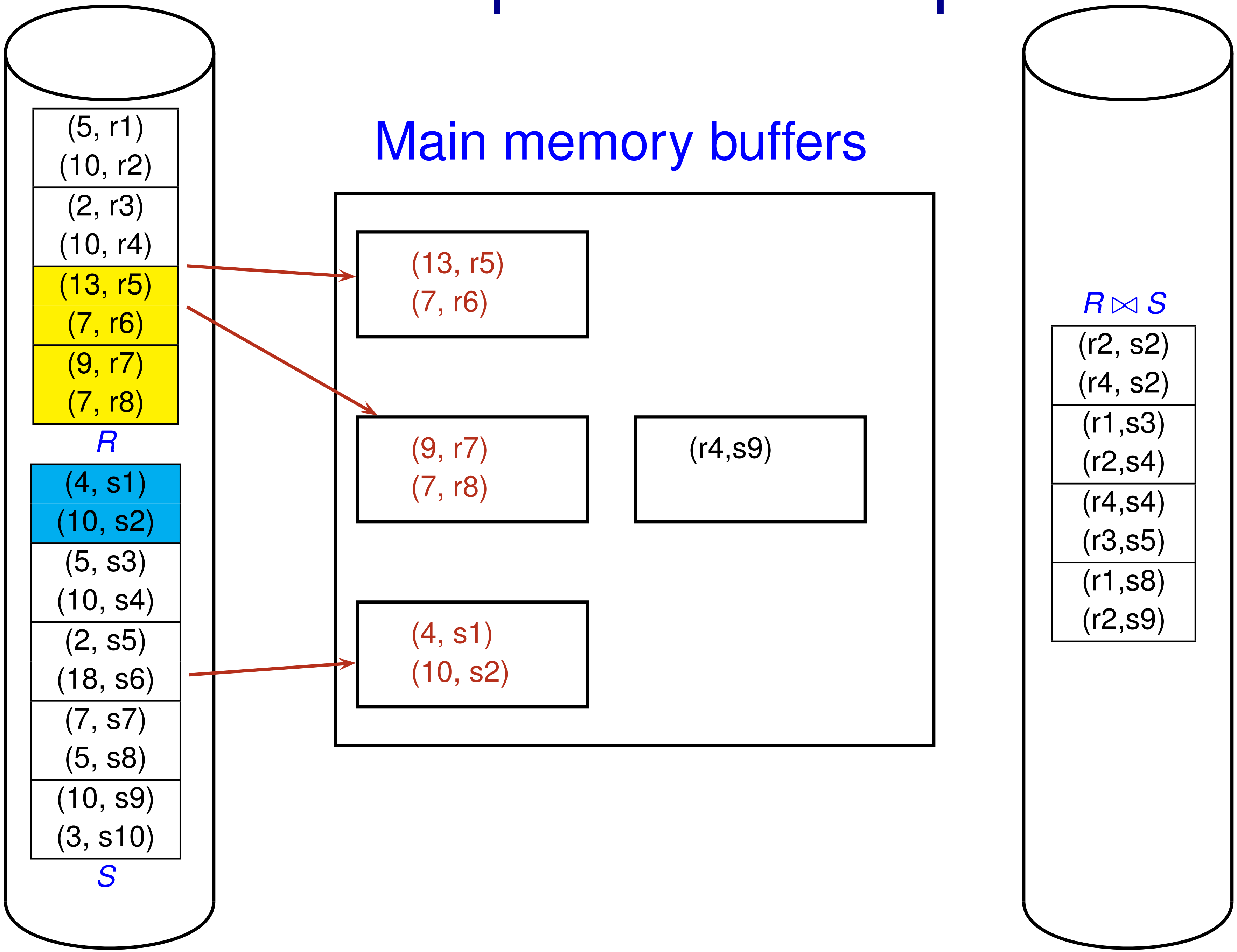
Block Nested Loop Join: Example



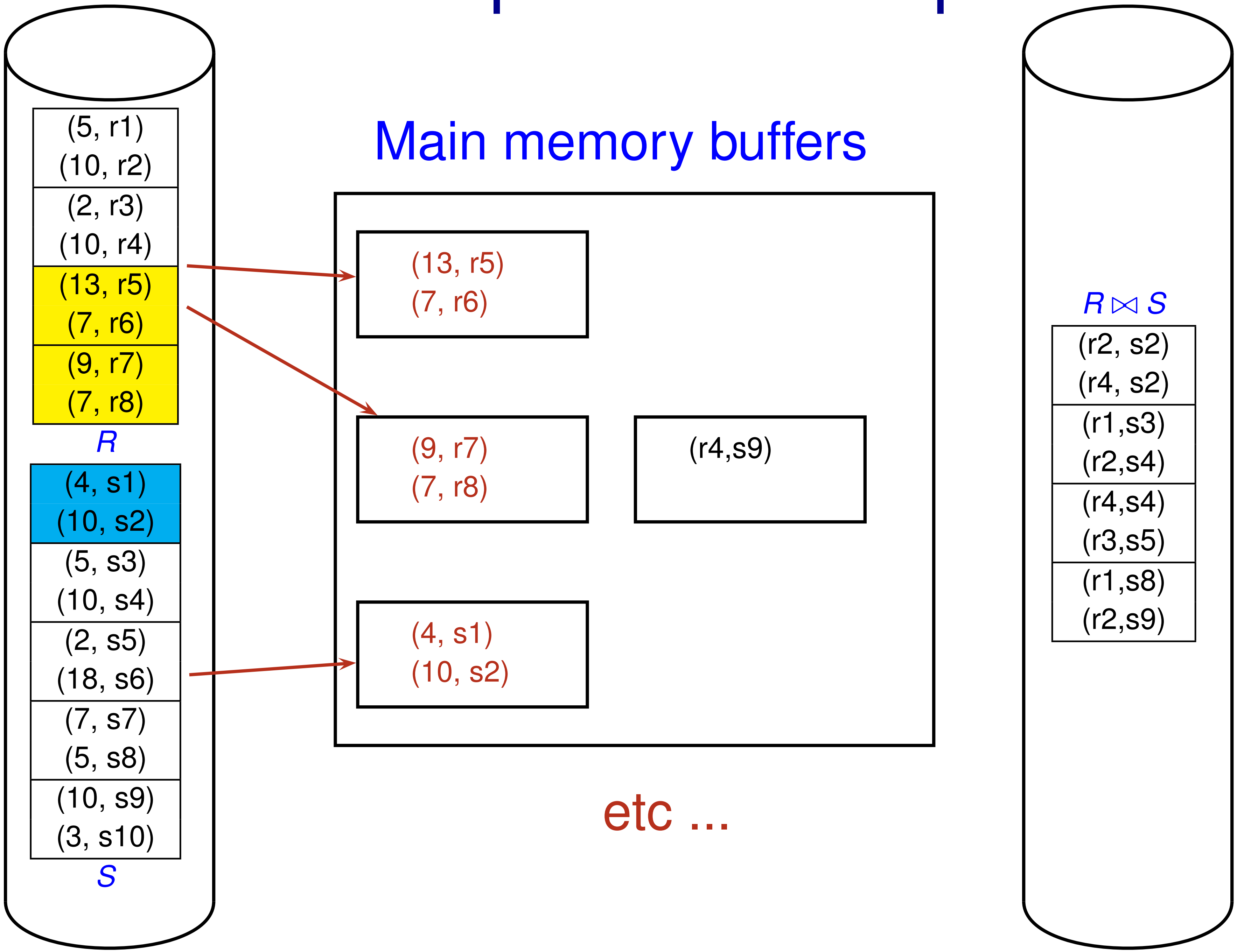
Block Nested Loop Join: Example



Block Nested Loop Join: Example

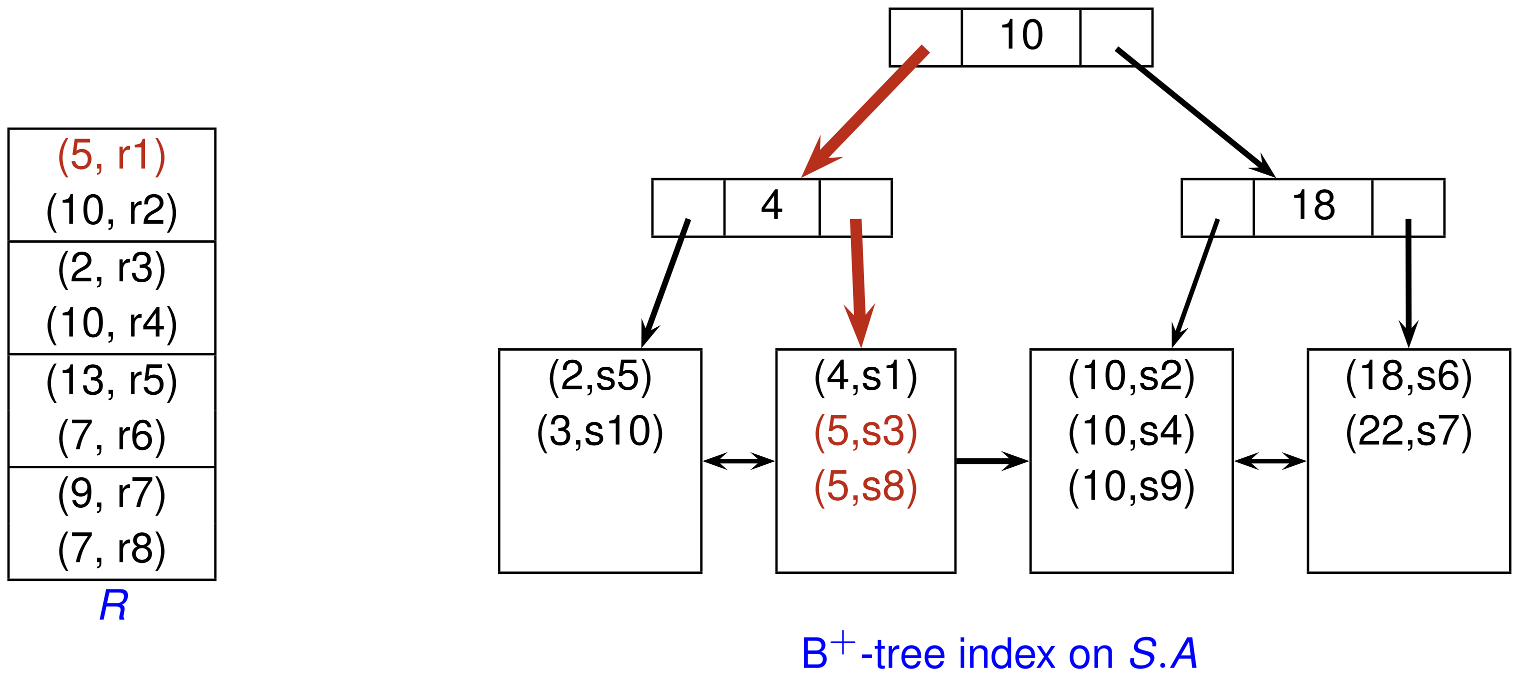


Block Nested Loop Join: Example



Index Nested Loop Join

- Consider $R(A, B) \bowtie_A S(A, C)$
- Assume that there's a B$^+$-tree index on $S.A$



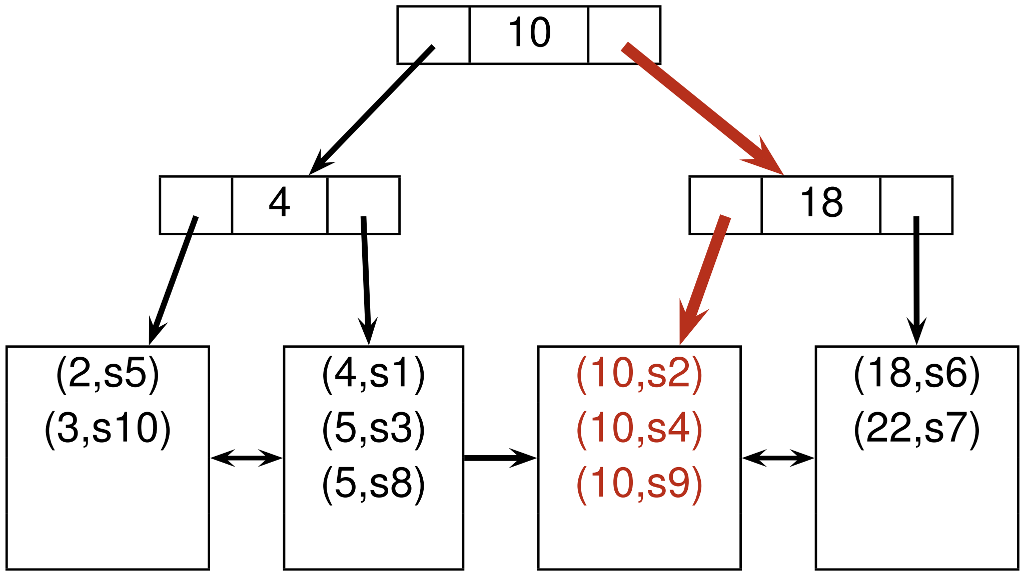
First, join $(5, r1) \in R$ with matching tuples in S

Index Nested Loop Join

- Consider $R(A, B) \bowtie_A S(A, C)$
- Assume that there's a B$^+$-tree index on $S.A$

(5, r1) (10, r2)
(2, r3) (10, r4)
(13, r5) (7, r6)
(9, r7) (7, r8)

R



B$^+$-tree index on $S.A$

Next, join $(10, r2) \in R$ with matching tuples in S, and so on ...

Index Nested Loop Join

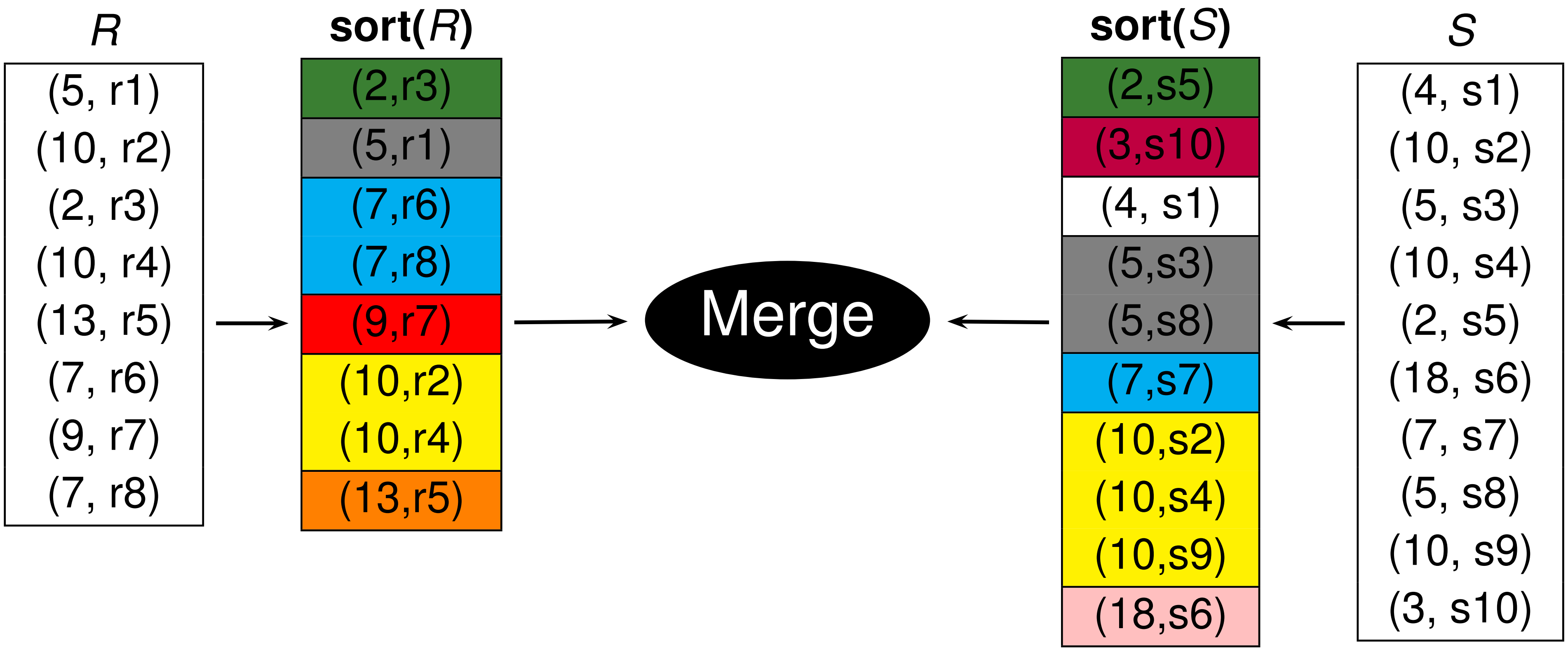
- Precondition: there is an index on the join attribute(s) of inner relation
- Idea:

  for each tuple $r \in R$ do

  use r to probe S's index to find matching tuples
- Analysis:
  - Let $R.A_i = S.B_j$ be the join condition
  - Uniform distribution assumption:
    each R-tuple joins with $\lceil \frac{||S||}{||\pi_{B_j}(S)||} \rceil$ number of S-tuples
  - For a format-1 B$^+$-tree index on S,
    - ★ I/O Cost = $\underbrace{|R|}_{\text{scan R}} + \underbrace{||R|| \times J}_{\text{join each R-tuple with S}}$
    - ★ $J = \underbrace{\log_F(\lceil \frac{||S||}{b_d} \rceil)}_{\text{search index's internal nodes}} + \underbrace{\lceil \frac{||S||}{b_d \; ||\pi_{B_j}(S)||} \rceil}_{\text{search index's leaf nodes}}$

Sort-Merge Join

- Idea: sort both relations based on join attributes & merge them
- A sorted relation R consists of partitions R_i of records where $r, r' \in R_i$ iff r and r' have the same values for the join attribute(s)



Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 10 13

S: 4 5 5 10 10 18 22

$R \bowtie S$:

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: $\boxed{2}$ 5 7 10 10 13

S: $\boxed{4}$ 5 5 10 10 18 22

$R \bowtie S$:

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R:	2	$\boxed{5}$	7	10	10	13	
S:	$\boxed{4}$	5	5	10	10	18	22

$R \bowtie S$:

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 $\boxed{5}$ 5 10 10 18 22

$R \bowtie S$:

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 $\boxed{5}$ 5 10 10 18 22

$R \bowtie S$:

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 $\boxed{5}$ 5 10 10 18 22

$R \bowtie S$:(5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 5 $\boxed{5}$ 10 10 18 22

$R \bowtie S$:(5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 5 $\boxed{5}$ 10 10 18 22

$R \bowtie S$:(5,5) (5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 $\boxed{5}$ 7 10 10 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5) (5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 $\boxed{7}$ 10 10 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5) (5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5)(5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5) (5,5)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 10 $\boxed{10}$ 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 10 $\boxed{10}$ 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 $\boxed{10}$ 10 13

S: 4 5 5 10 10 $\boxed{18}$ 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 10 10 $\boxed{18}$ 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 $\boxed{10}$ 10 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 10 $\boxed{10}$ 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 10 $\boxed{10}$ 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 $\boxed{10}$ 13

S: 4 5 5 10 10 $\boxed{18}$ 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a “smaller” tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 10 $\boxed{13}$

S: 4 5 5 10 10 $\boxed{18}$ 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)(10,10)

Sort-Merge Join: Merging Phase

- Each tuple in R-partition merges with all tuples in matching S-partition
- A *pointer* is maintained for each sorted join operand
- Each pointer is initialized to the first tuple in sorted operand
- Search for matching partitions by advancing the pointer that is pointing to a "smaller" tuple
- Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer
- **Example**:

R: 2 5 7 10 10 13

S: 4 5 5 10 10 18 22

$R \bowtie S$:(5,5)(5,5)(10,10)(10,10)(10,10)(10,10)

Sort-Merge Join Algorithm for $R \bowtie_{R.A_i = S.B_j} S$

if (R is not sorted) then sort R
if (S is not sorted) then sort S
t_r = first tuple in R
t_s = first tuple in S
p_s = first tuple in S partition
while ($t_r \neq null$) and ($p_s \neq null$) do
while ($t_r.A_i < p_s.B_j$) do
t_r = next tuple in R after t_r
while ($t_r.A_i > p_s.B_j$) do
p_s = next tuple in S afer p_s
$t_s = p_s$
while ($t_r.A_i = p_s.B_j$) do
$t_s = p_s$
while ($t_s.B_j = t_r.A_i$) do
add (t_r, t_s) to result
t_s = next tuple in S after t_s
t_r = next tuple in R after t_p
$p_s = t_s$

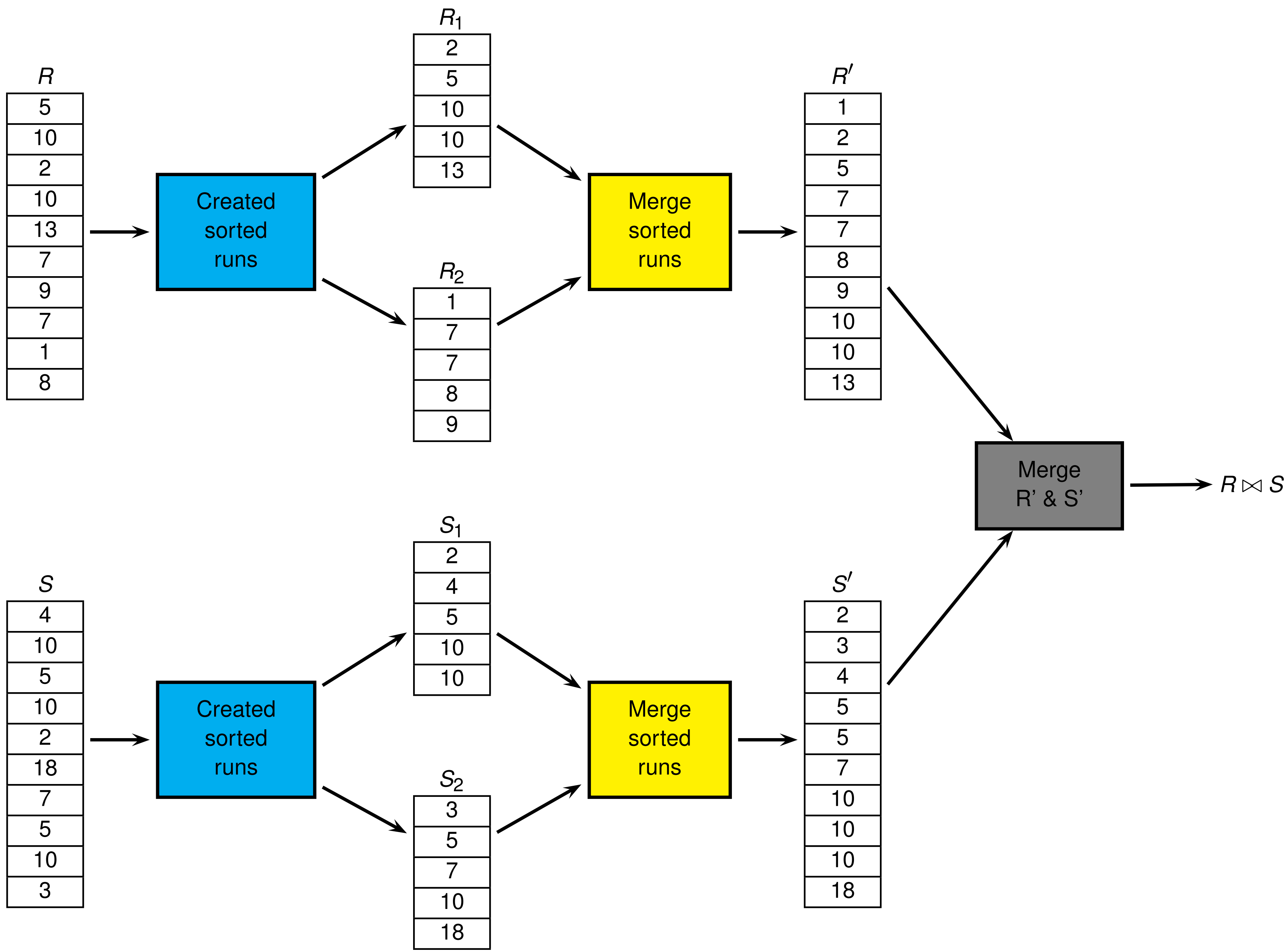
Sort-Merge Join: Analysis

- I/O cost = Cost to sort R + Cost to sort S + Merging cost
- **Cost to sort R** = $2|R|\ (\log_m(N_R) + 1)$ if using external merge sort
  - $N_R =$ number of initial sorted runs of R, $\quad m =$ merge factor
- **Cost to sort S** = $2|S|\ (\log_m(N_S) + 1)$ if using external merge sort
  - $N_S =$ number of initial sorted runs of S, $\quad m =$ merge factor
- If each S partition is scanned at most once during merging,
  - **Merging cost** = $|R| + |S|$
- Worst case occurs when each tuple of R requires scanning entire S!
  - **Merging cost** = $|R| + ||R|| \times |S|$

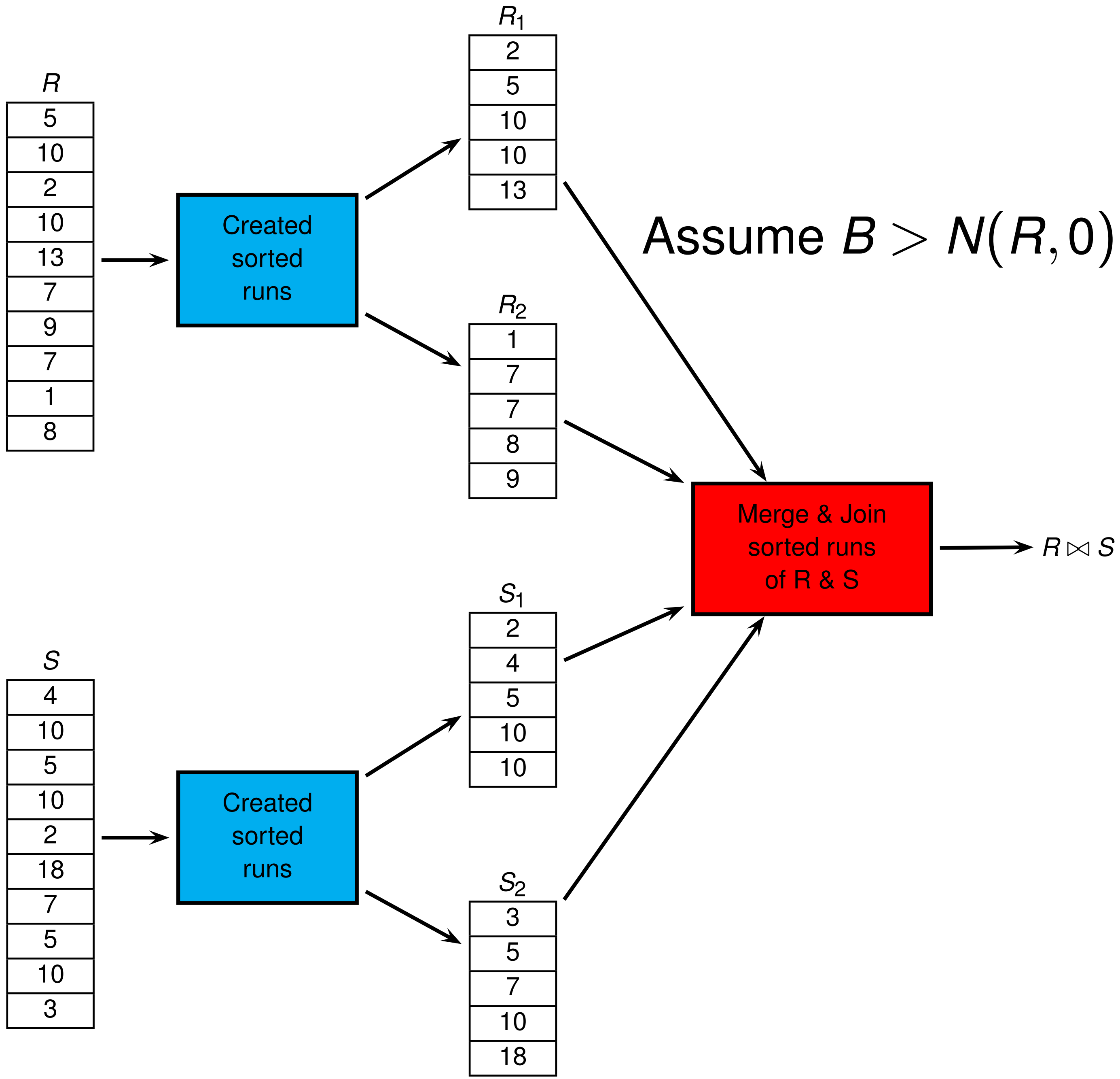
Sort-Merge Join: Optimization

- **Conventional Sort-Merge Join**
  - **Sort R**: create sorted runs of R; merge sorted runs of R
  - **Sort S**: create sorted runs of S; merge sorted runs of S
  - **Join R and S**: merge sorted R & sorted S
- Idea: Combine merge phase of sorting & merge phase of join
  - It's not necessary to merge sorted runs into a single run before performing join
  - If $B > N(R,i) + N(S,j)$ for some i & j, sorting of R and S can stop
    - N(R,i) = total number of sorted runs of R at the end of pass i of sorting R
- **Optimized Sort-Merge Join**
  - Create sorted runs of R; merge sorted runs of R partially
  - Create sorted runs of S; merge sorted runs of S partially
  - Merge remaining sorted runs of R & S and join them at the same time

Conventional Sort-Merge Join



Optimized Sort-Merge Join



Assume $B > N(R, 0) + N(S, 0)$

Optimized Sort-Merge Join: Analysis

- Assume $|R| \leq |S|$
- If $B > \sqrt{2|S|}$
  - Number of initial sorted runs of S < $\sqrt{\frac{|S|}{2}}$
  - Total number of initial sorted runs of R and S < $\sqrt{2|S|}$
  - One pass is sufficient to merge and join the initial sorted runs R & S
  - I/O Cost = $3 \times (|R| + |S|)$

Hash Join, $R \bowtie_{R.A=S.B} S$

- **Idea**:
  - Partition R and S into k partitions using some hash function h
    - ⋆ $R = R_1 \cup R_2 \cup \cdots \cup R_k,\ t \in R_i$ iff $h(t.A) = i$
    - ⋆ $S = S_1 \cup S_2 \cup \cdots \cup S_k,\ t \in S_i$ iff $h(t.B) = i$
    - ⋆ $\pi_A(R_i) \cap \pi_B(S_j) = \emptyset$ for each R_i & S_j, $i \neq j$
  - Joins corresponding pair of partitions
    - ⋆ $R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2) \cup \cdots \cup (R_k \bowtie S_k)$
- **Algorithms**:
  - Grace hash join
  - Hybrid hash join (not covered in lecture)

Grace Hash Join, $R \bowtie_{R.A=S.B} S$

- Consists of three phases:
  1. Partition R into $R_1, \cdots, R_k$
  2. Partition S into $S_1, \cdots, S_k$
  3. Probing phase: probes each R_i with S_i
     - Read R_i to build a hash table
     - Read S_i to probe hash table
- R is called the build relation & S is called the probe relation

Grace Hash Join, $R \bowtie_{R.A=S.B} S$

- Consists of three phases:
  1. Partition R into $R_1, \cdots, R_k$
  2. Partition S into $S_1, \cdots, S_k$
  3. Probing phase: probes each R_i with S_i
     - Read R_i to build a hash table
     - Read S_i to probe hash table
- R is called the build relation & S is called the probe relation

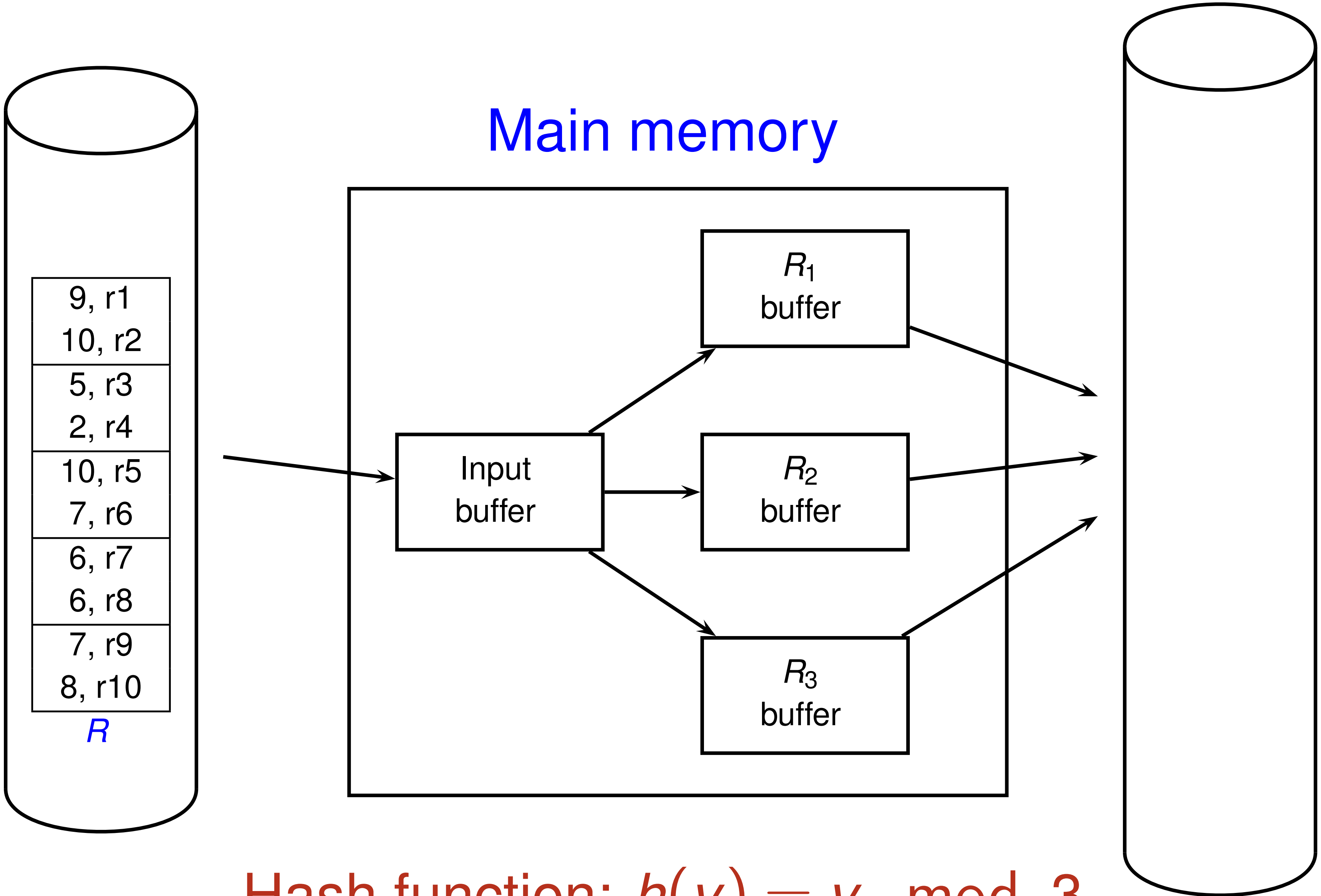
Partitioning (building) phases

```
initialize a hash table T with k buckets
for each tuple r ∈ R do
    insert r into bucket h(r.A) of T
write each bucket R_i of T to disk
initialize a hash table T with k buckets
for each tuple s ∈ S do
    insert s into bucket h(s.B) of T
write each bucket S_i of T to disk
```

Probing (matching) phase

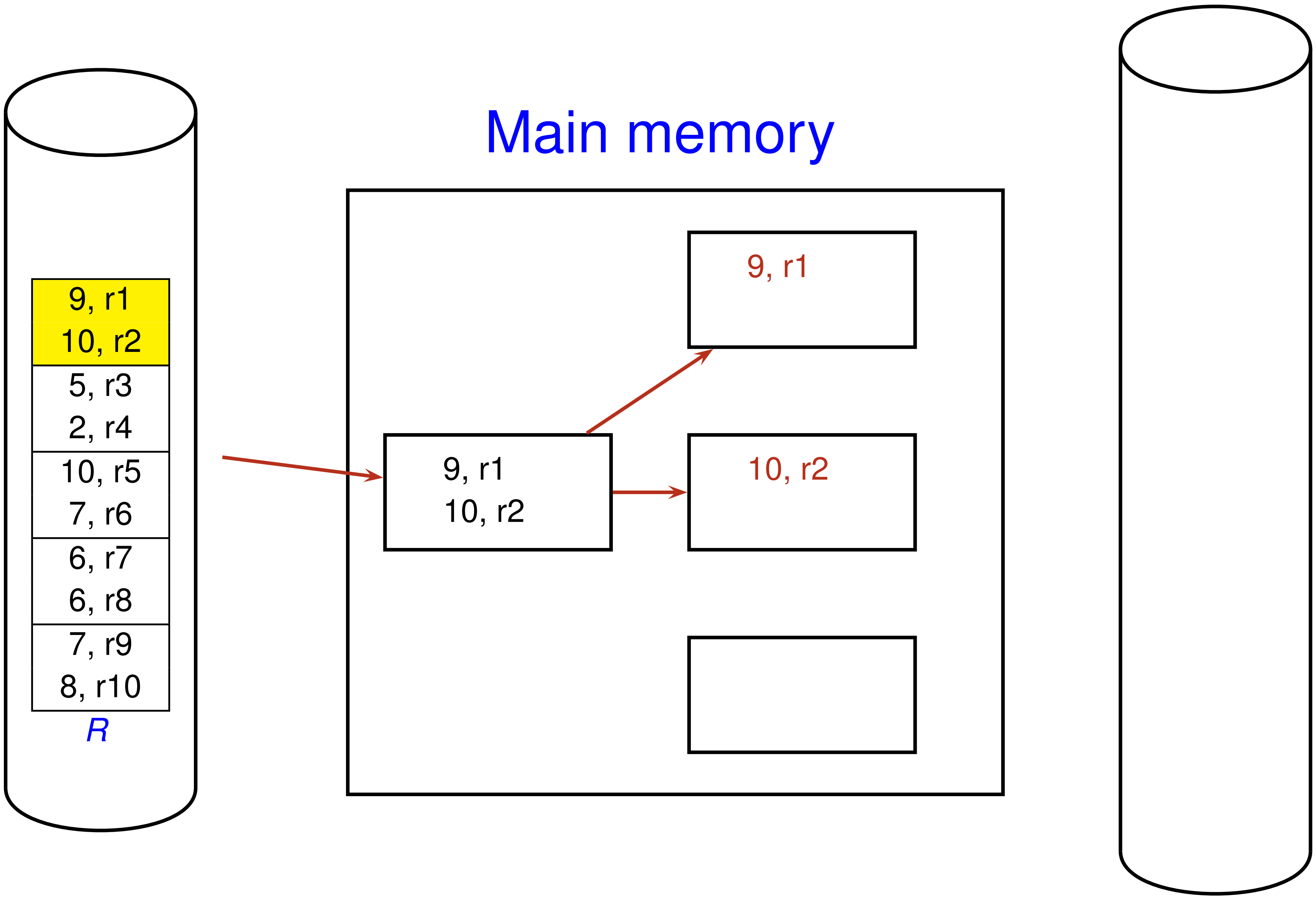
```
for i = 1 to k do
    initialize a hash table T
    for each tuple r in partition R_i do
        insert r into bucket h'(r.A) of T
    for each tuple s in partition S_i do
        for each tuple r in bucket h'(s.B) of T do
            if r and s matches then output (r, s)
```

Grace Hash Join: Partitioning Relation R

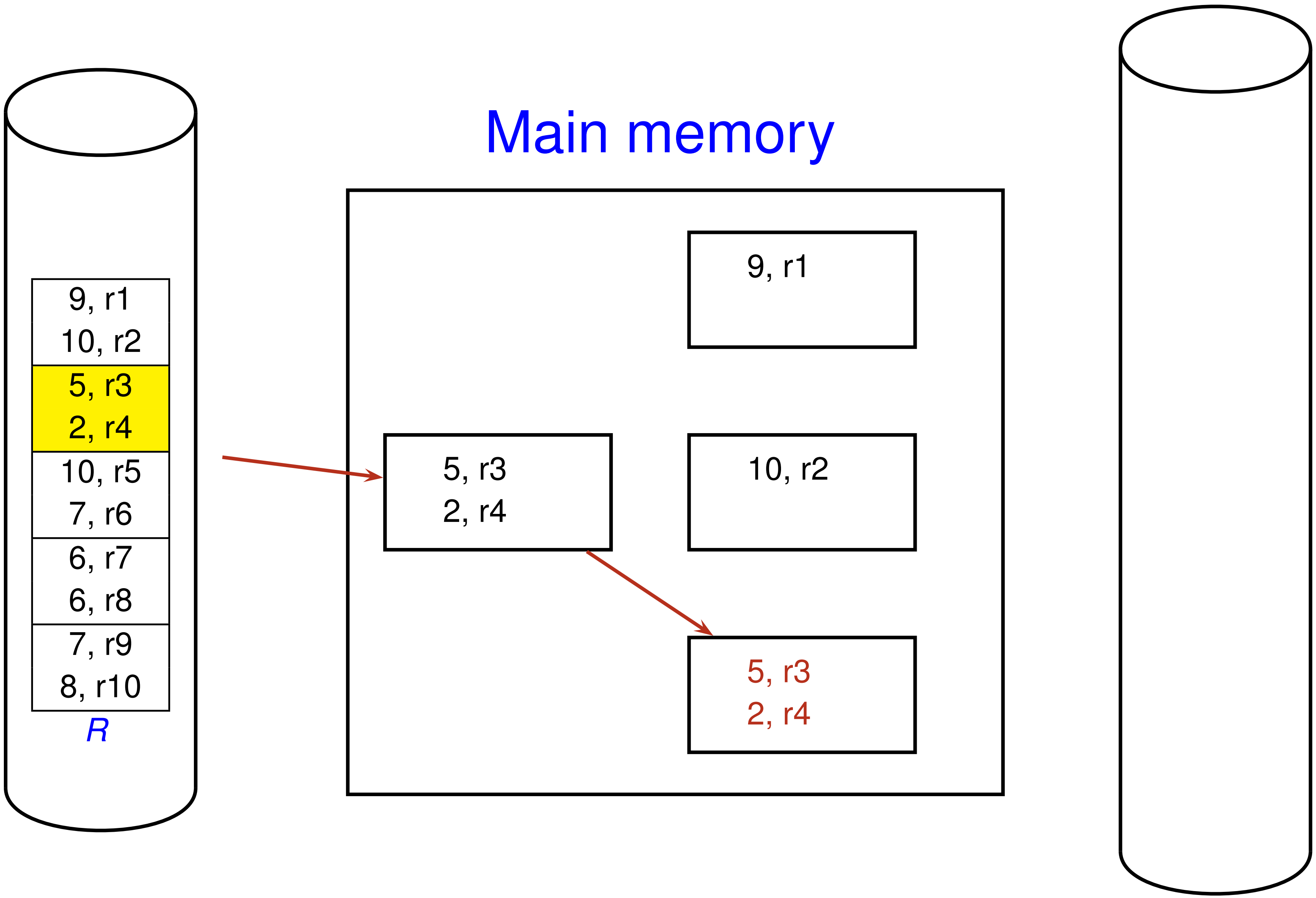


Hash function: $h(v) = v \mod 3$

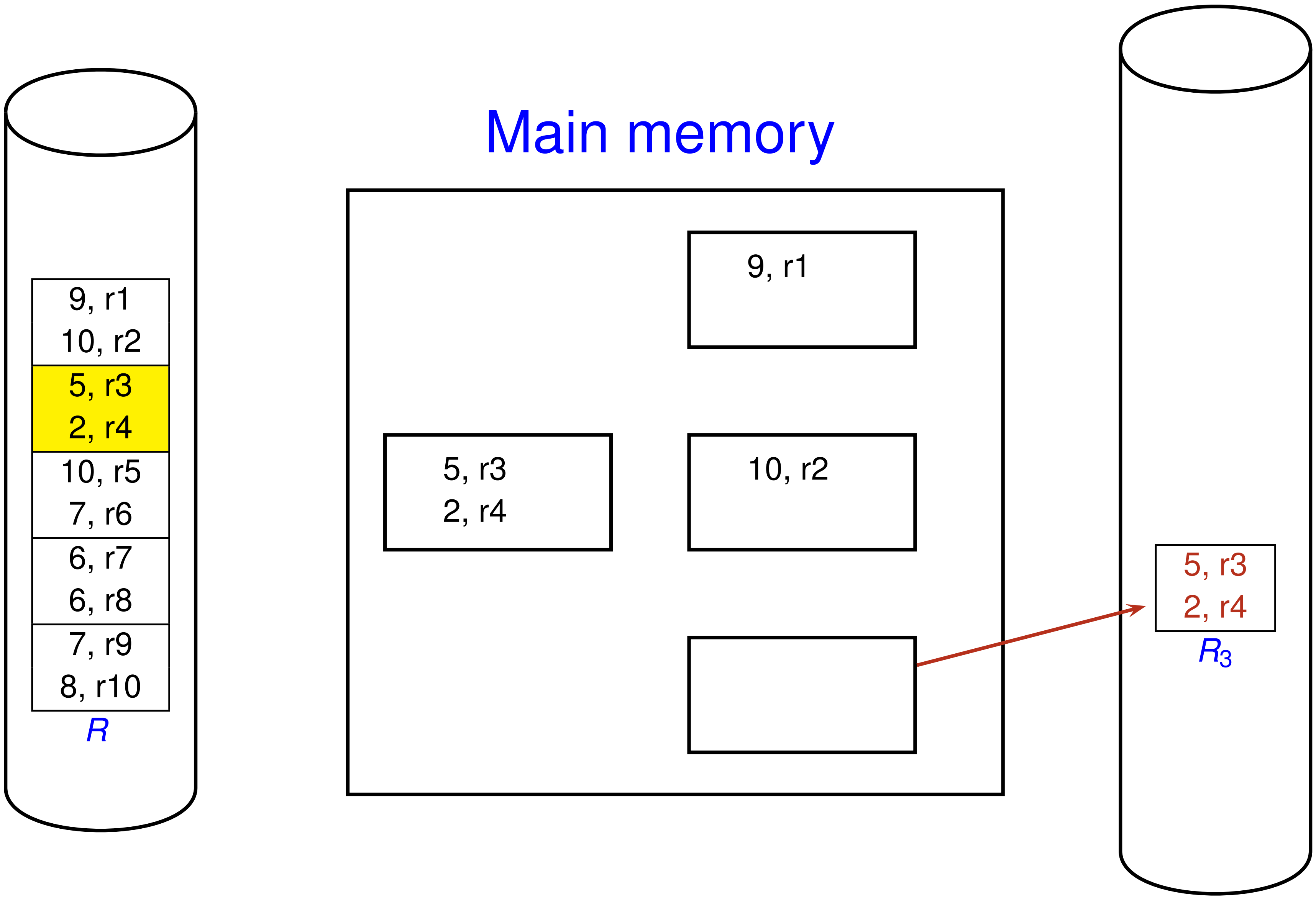
Grace Hash Join: Partitioning Relation R



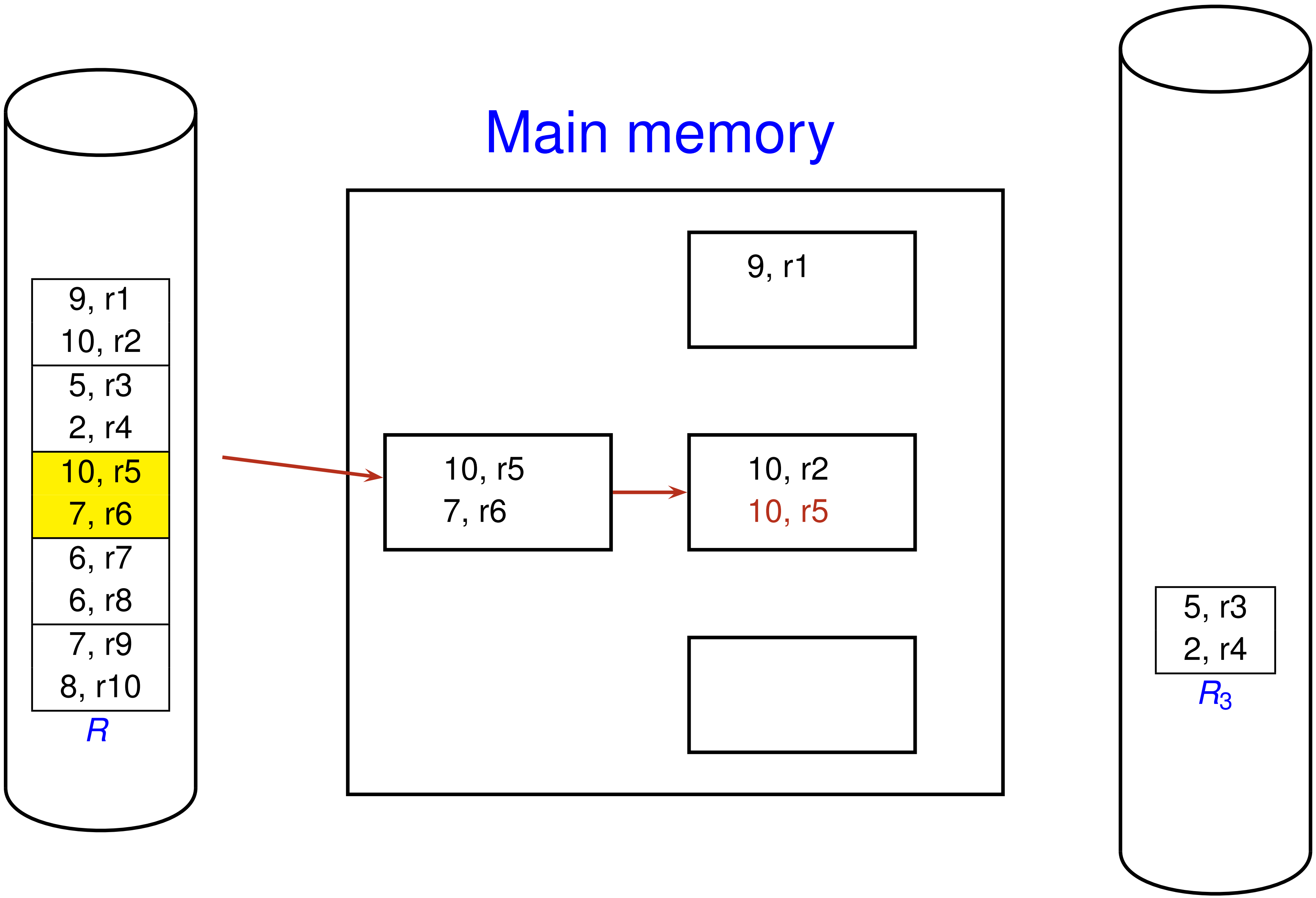
Grace Hash Join: Partitioning Relation R



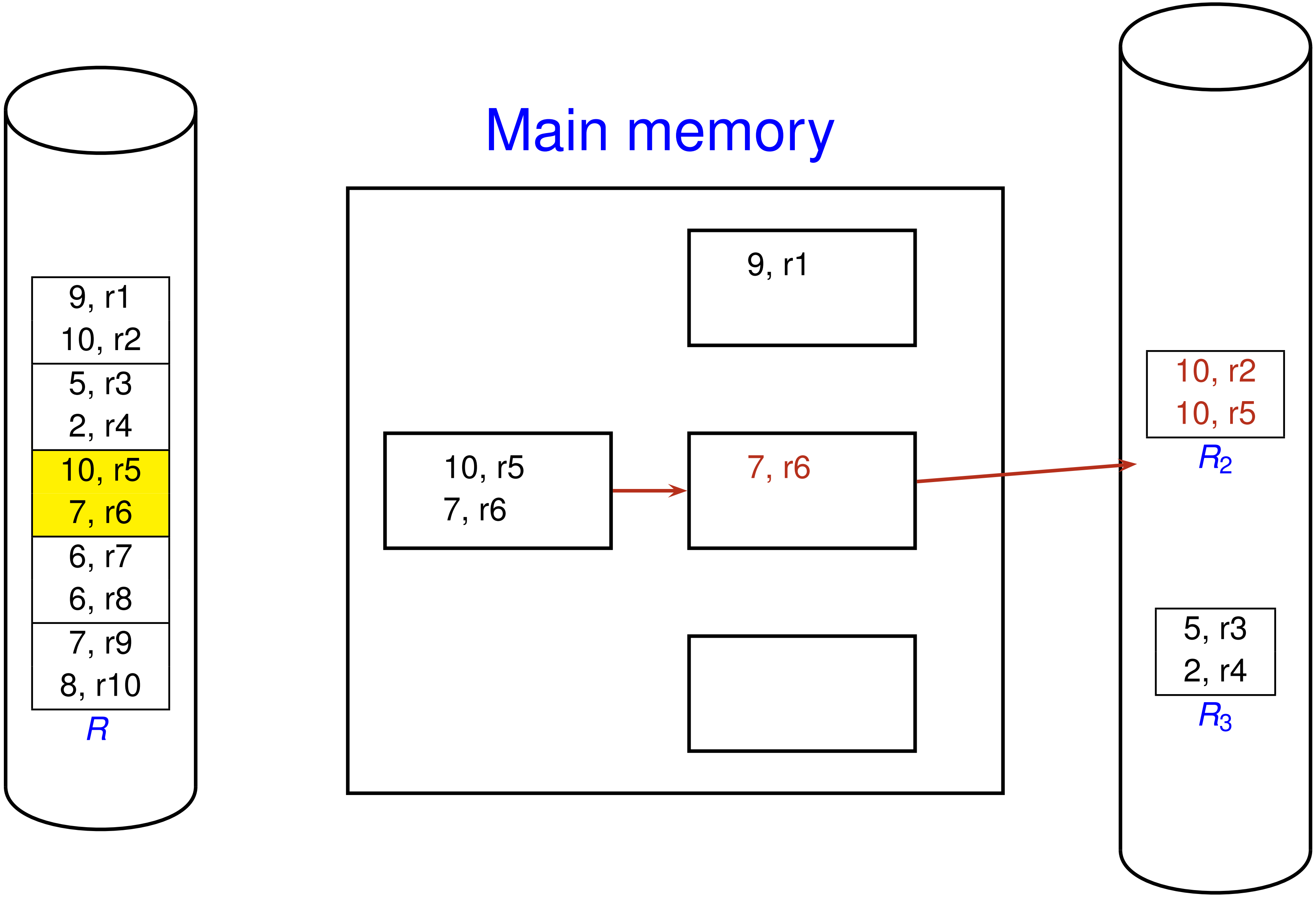
Grace Hash Join: Partitioning Relation R



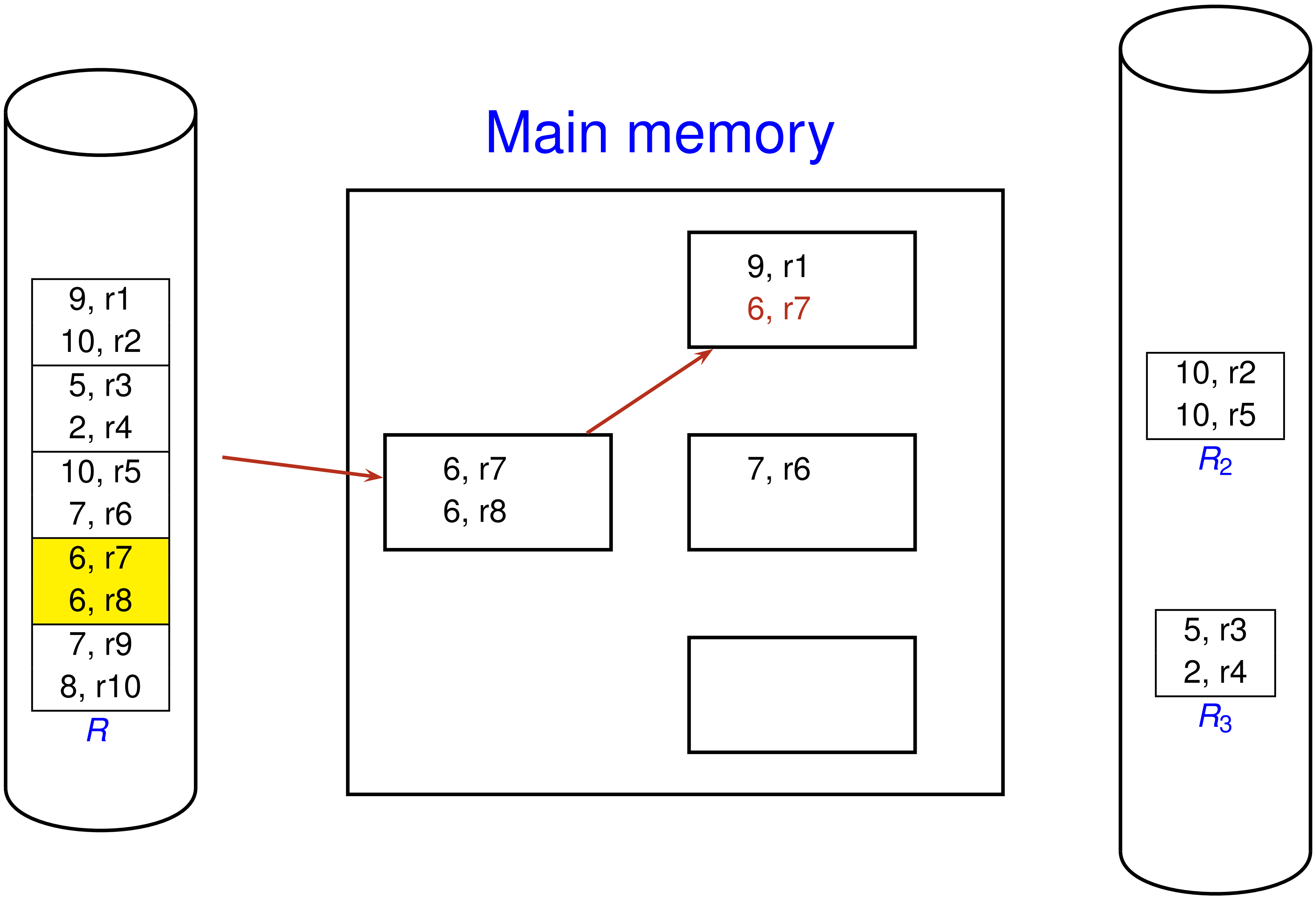
Grace Hash Join: Partitioning Relation R



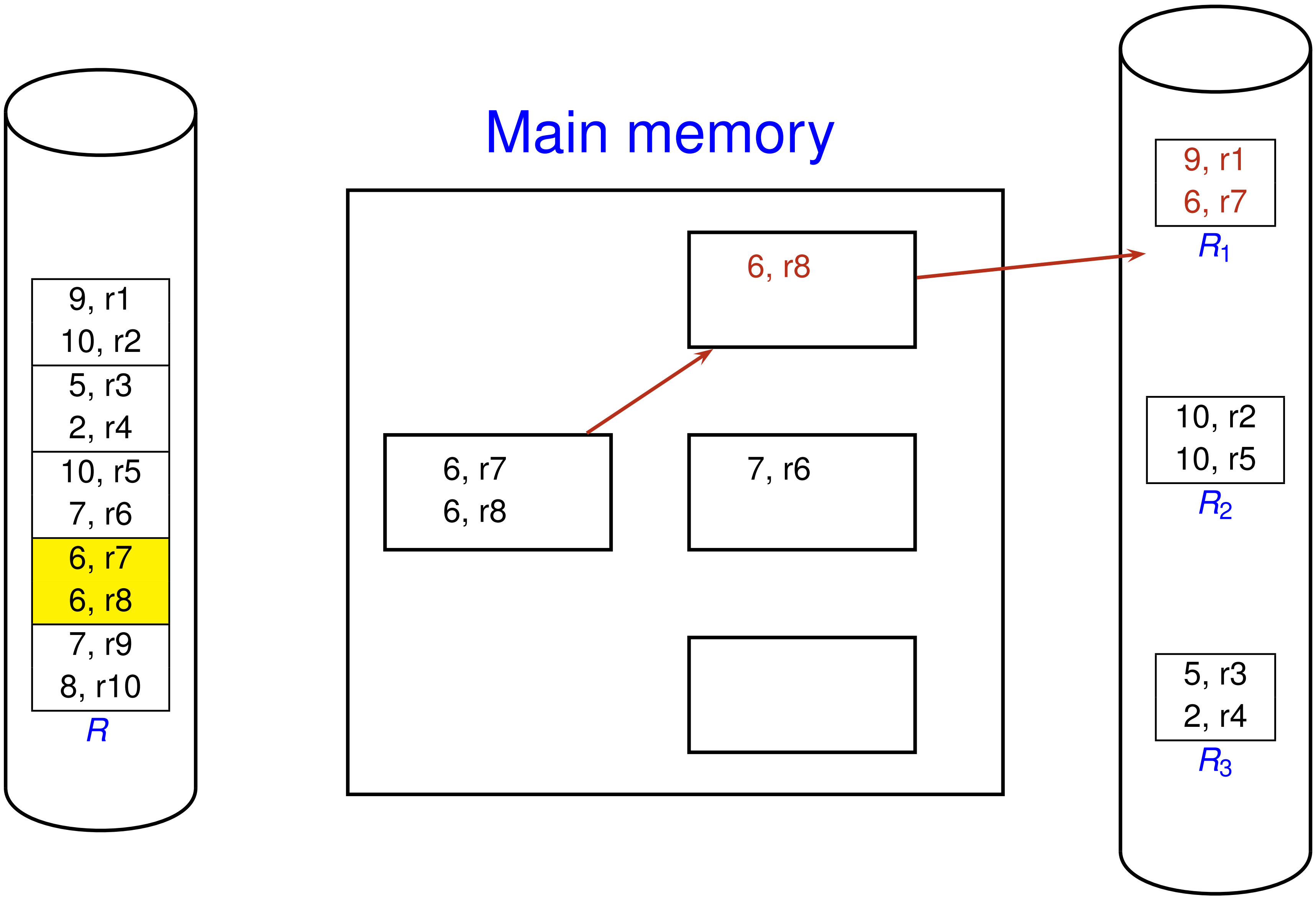
Grace Hash Join: Partitioning Relation R



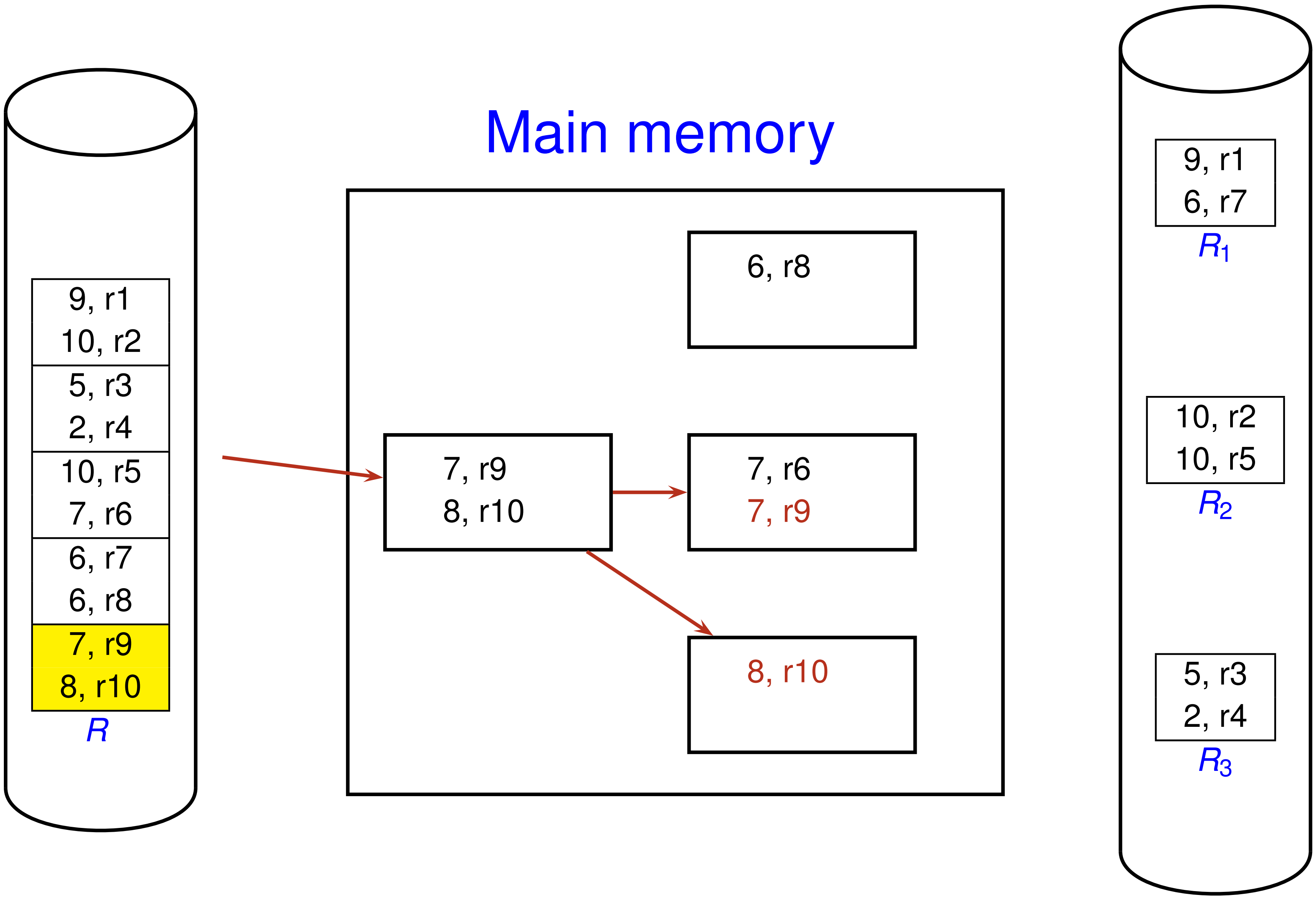
Grace Hash Join: Partitioning Relation R



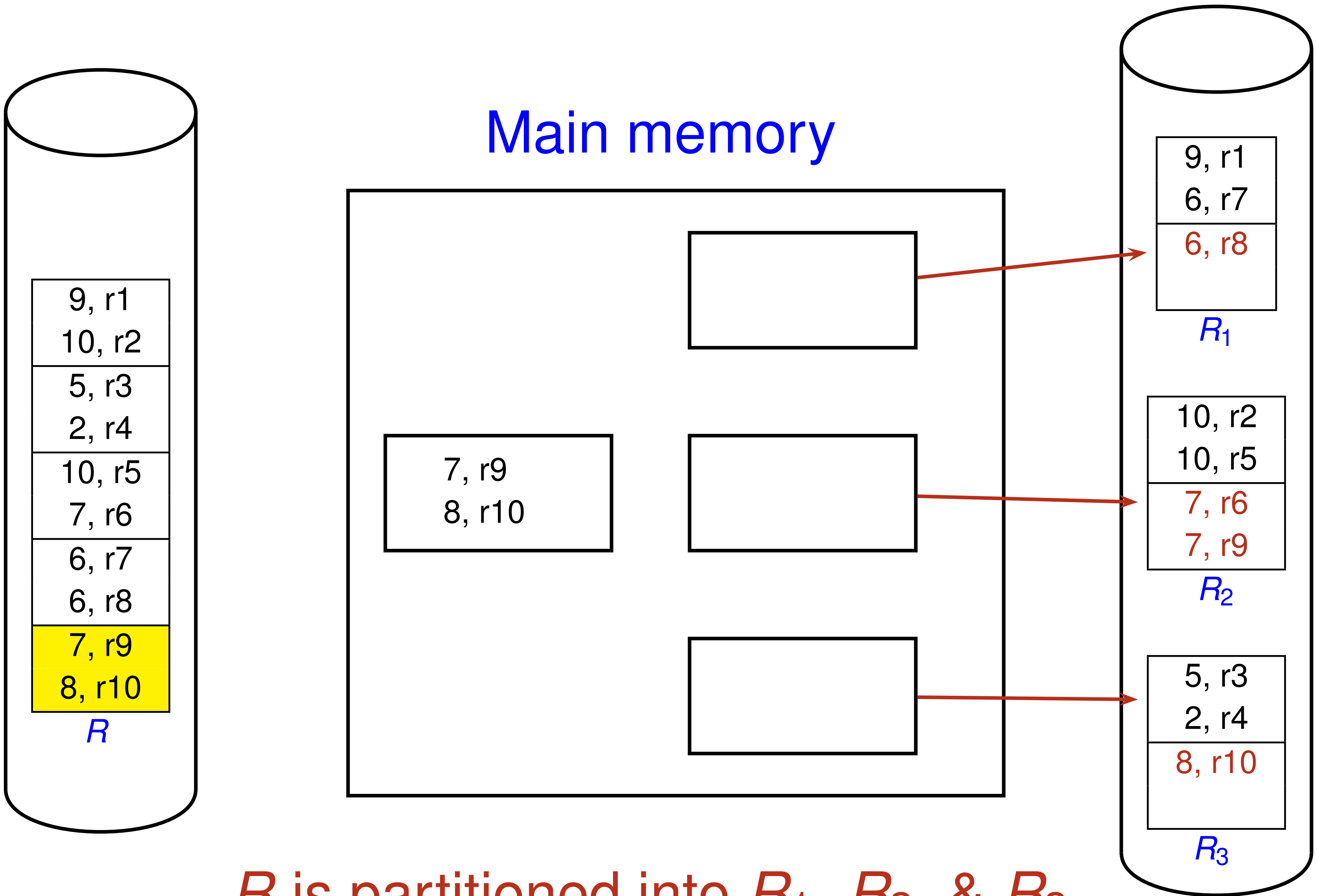
Grace Hash Join: Partitioning Relation R



Grace Hash Join: Partitioning Relation R

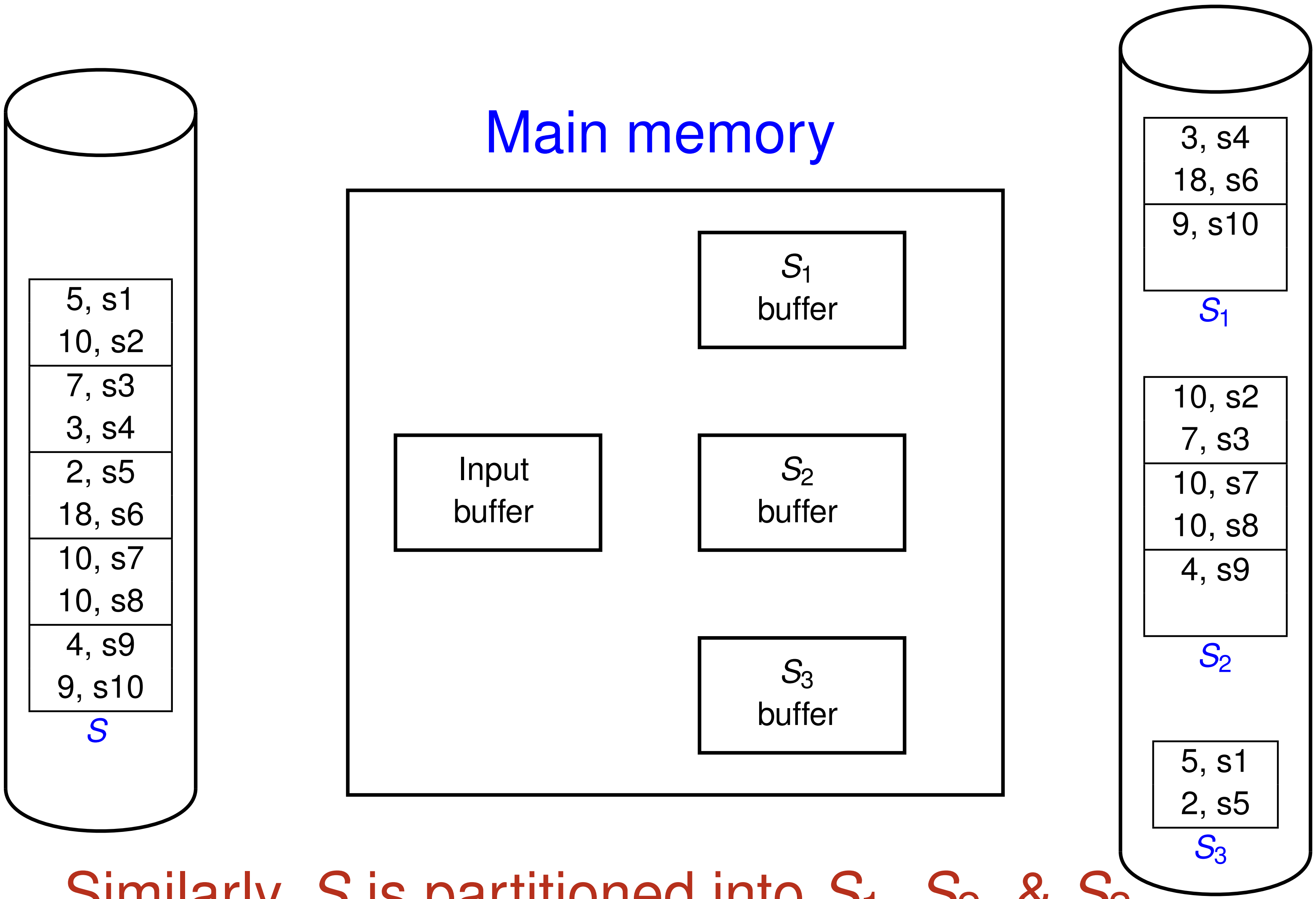


Grace Hash Join: Partitioning Relation R



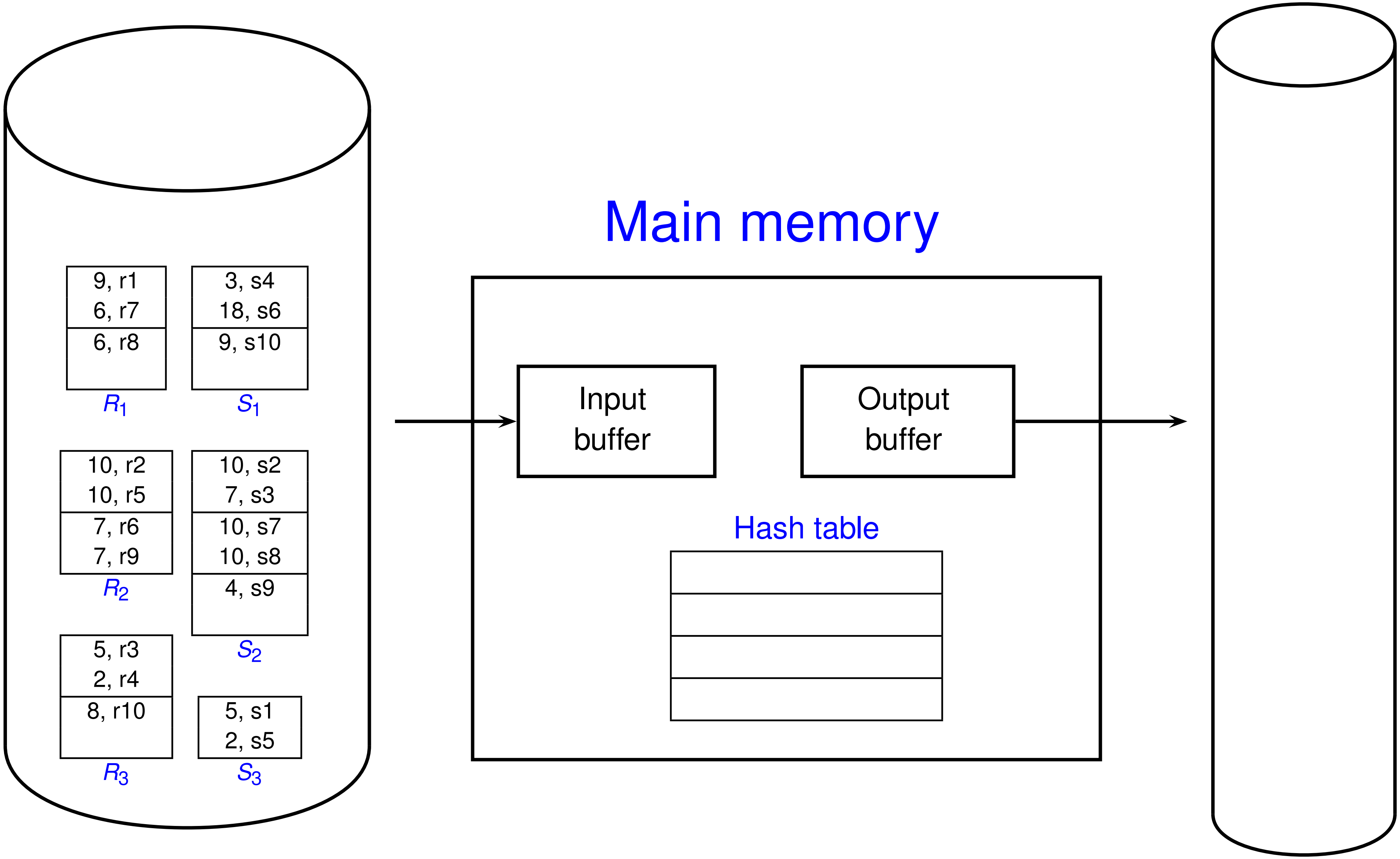
R is partitioned into R_1, R_2, & R_3

Grace Hash Join: Partitioning Relation S

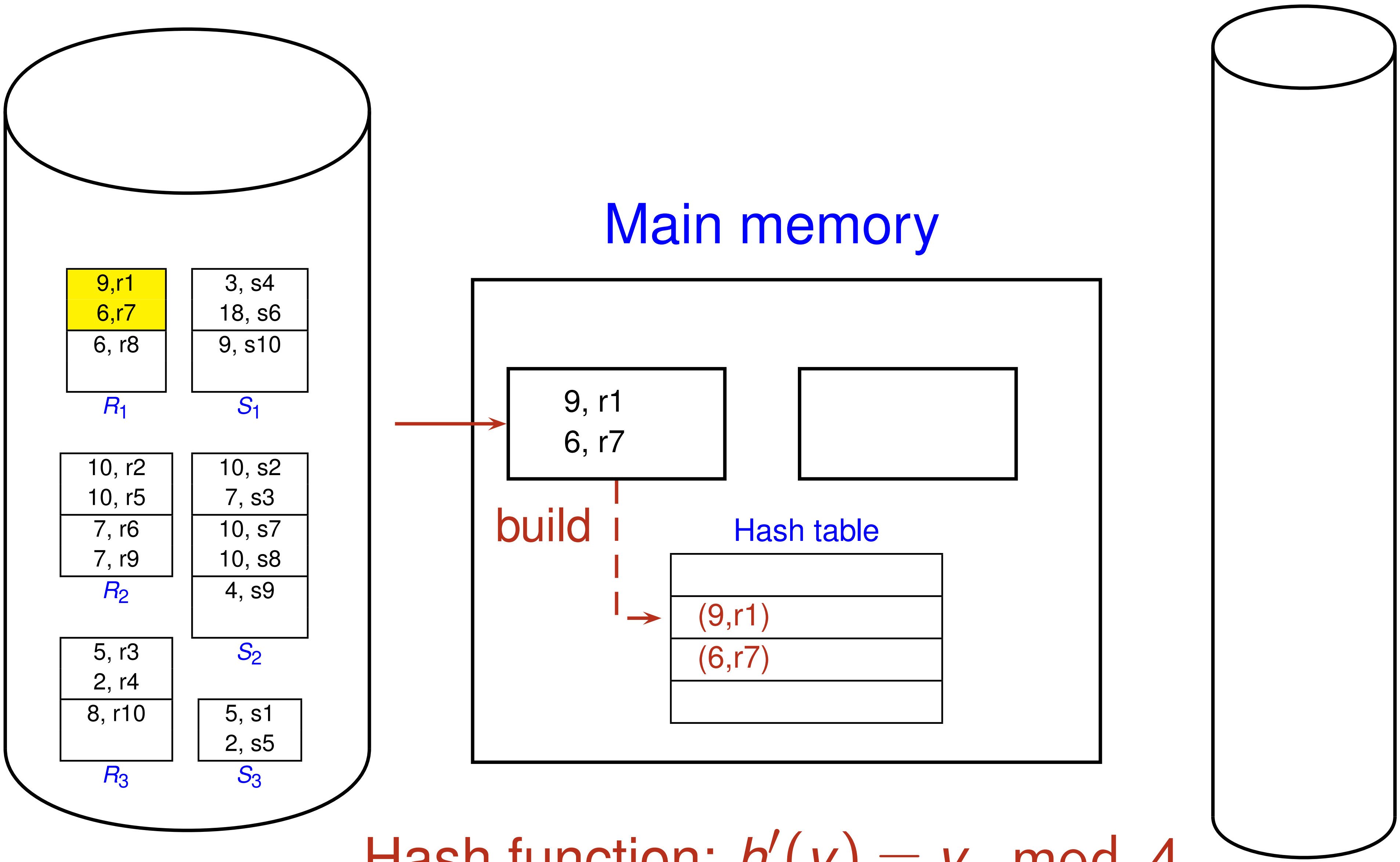


Similarly, S is partitioned into S_1, S_2, & S_3

Grace Hash Join: Probing Phase

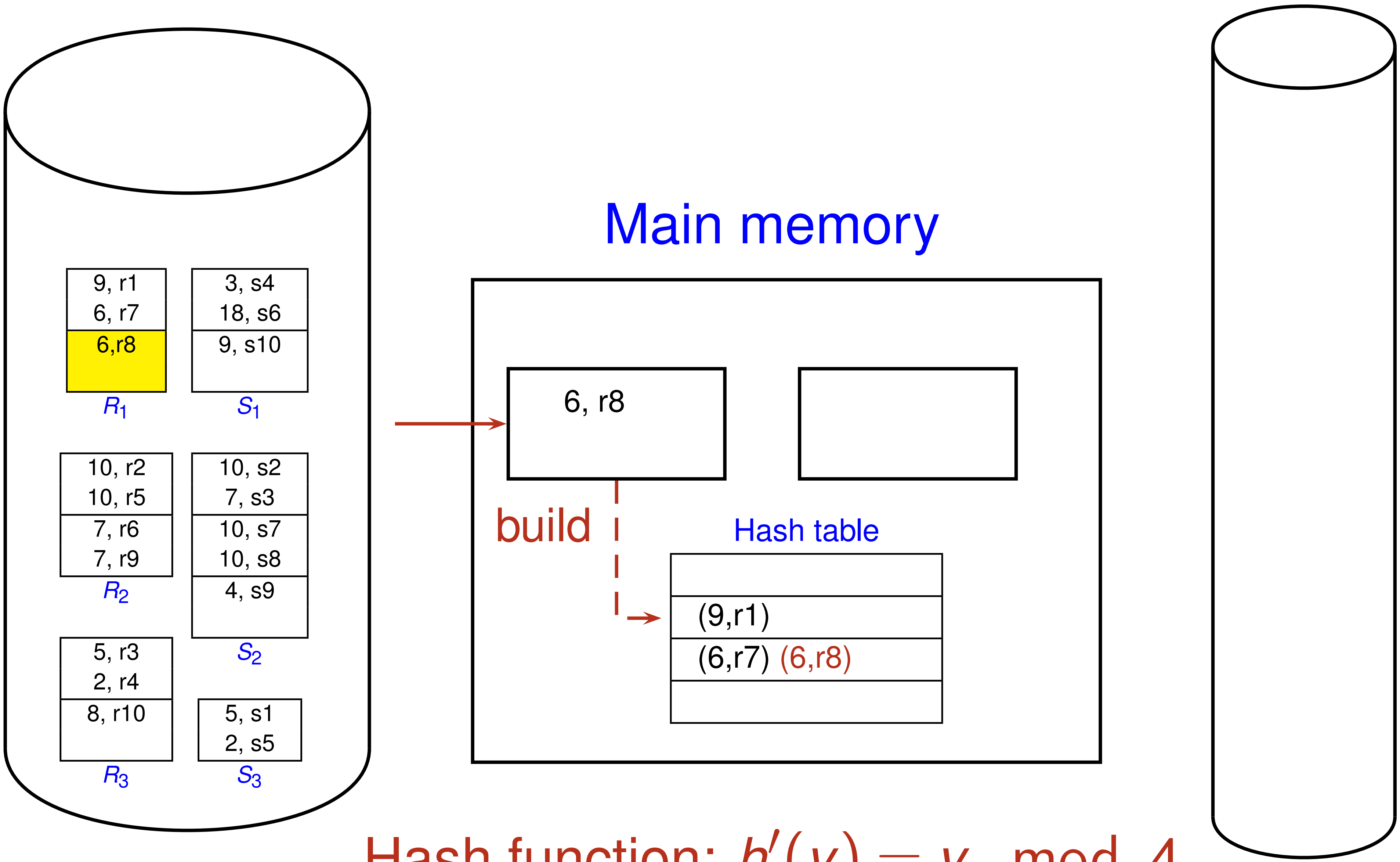


Grace Hash Join: Probing Phase



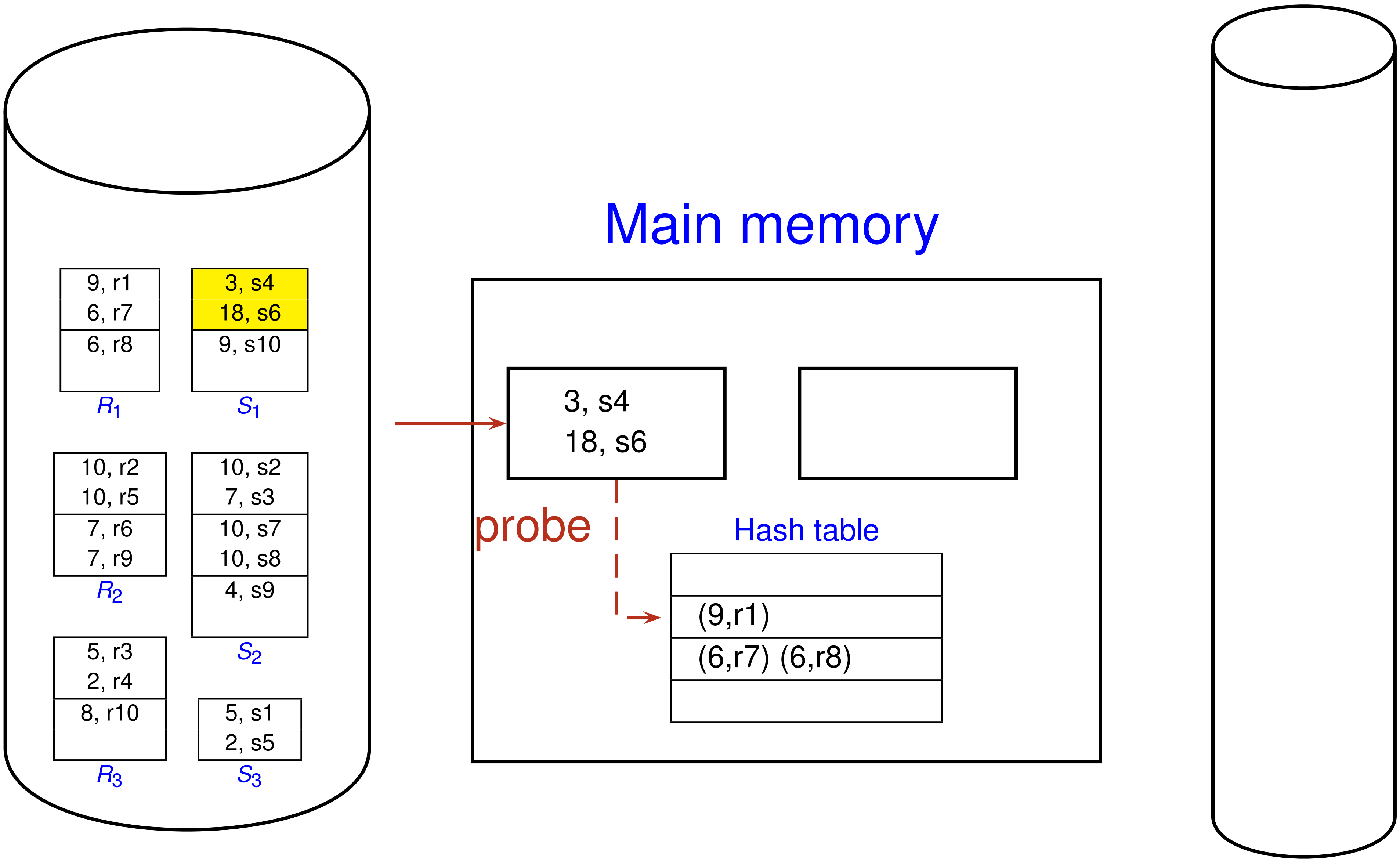
Hash function: $h'(v) = v \mod 4$

Grace Hash Join: Probing Phase

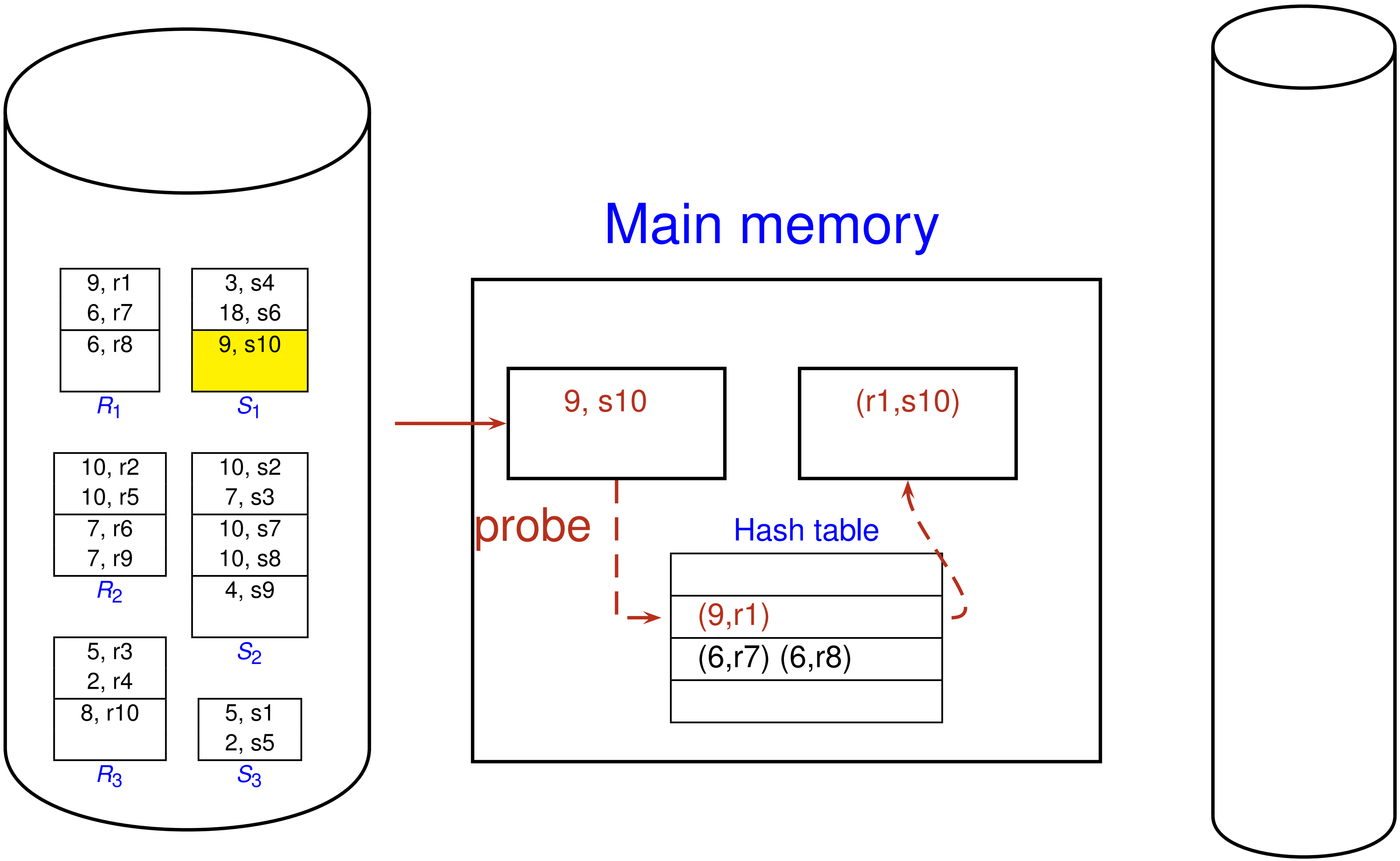


Hash function: $h'(v) = v \mod 4$

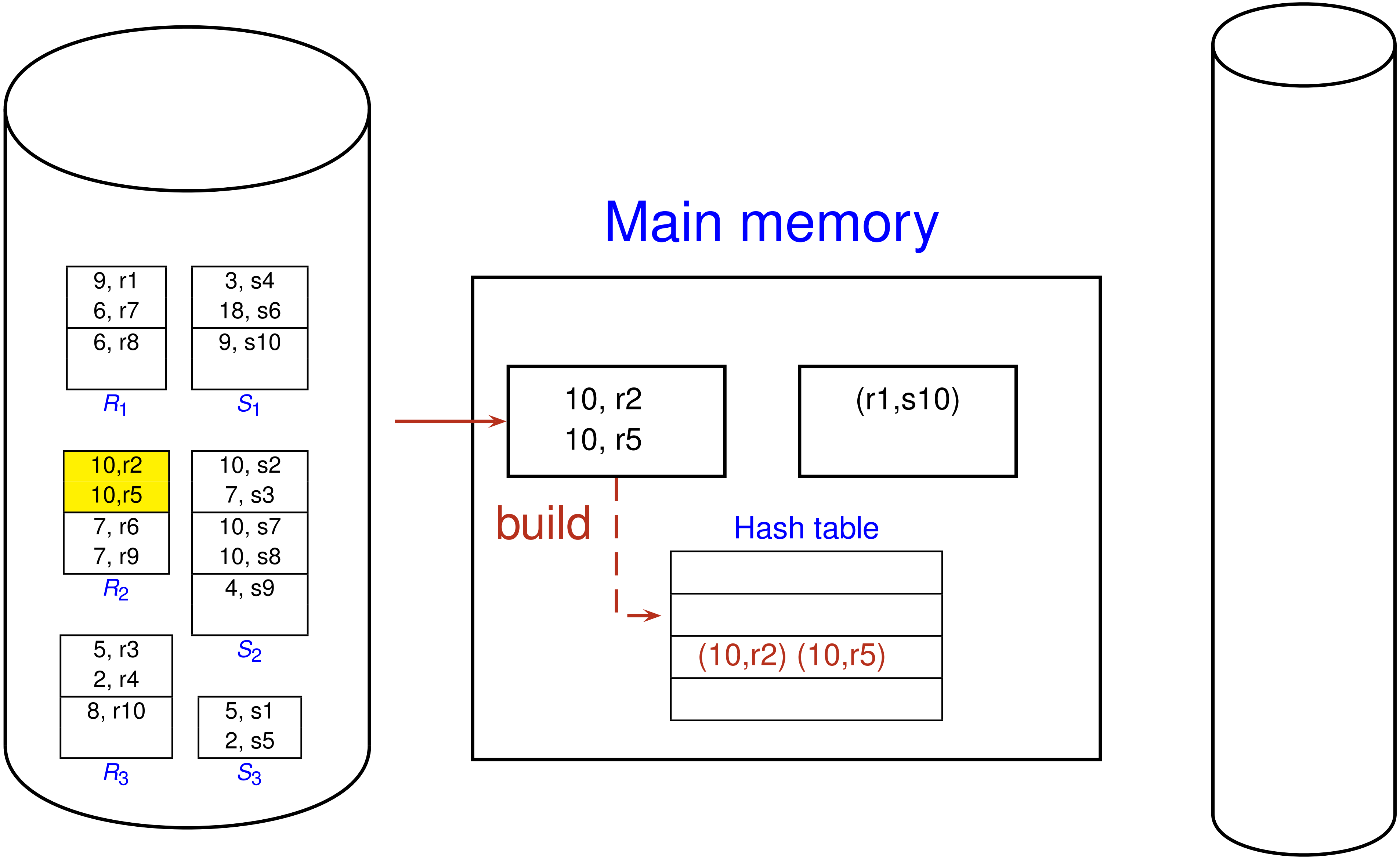
Grace Hash Join: Probing Phase



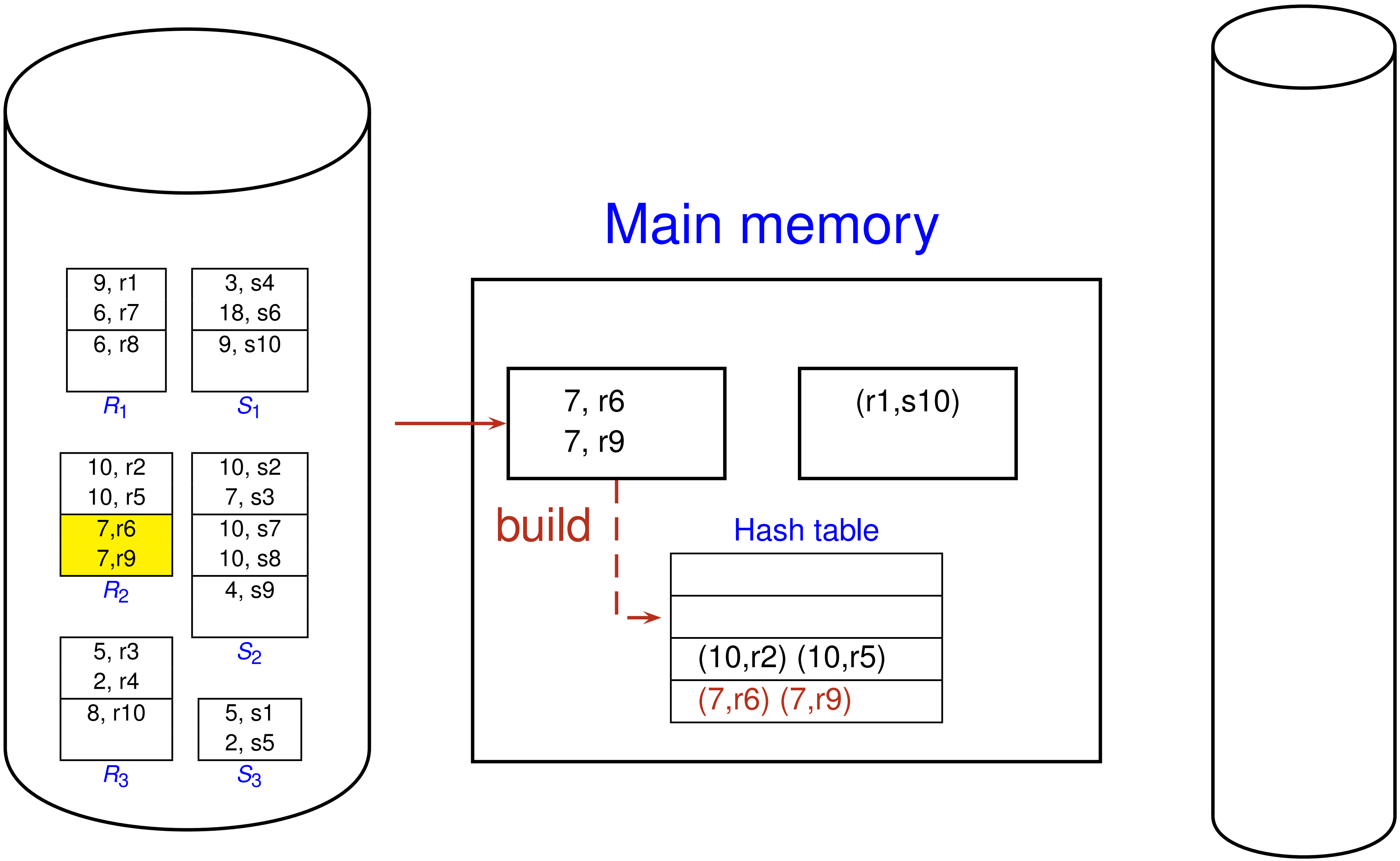
Grace Hash Join: Probing Phase



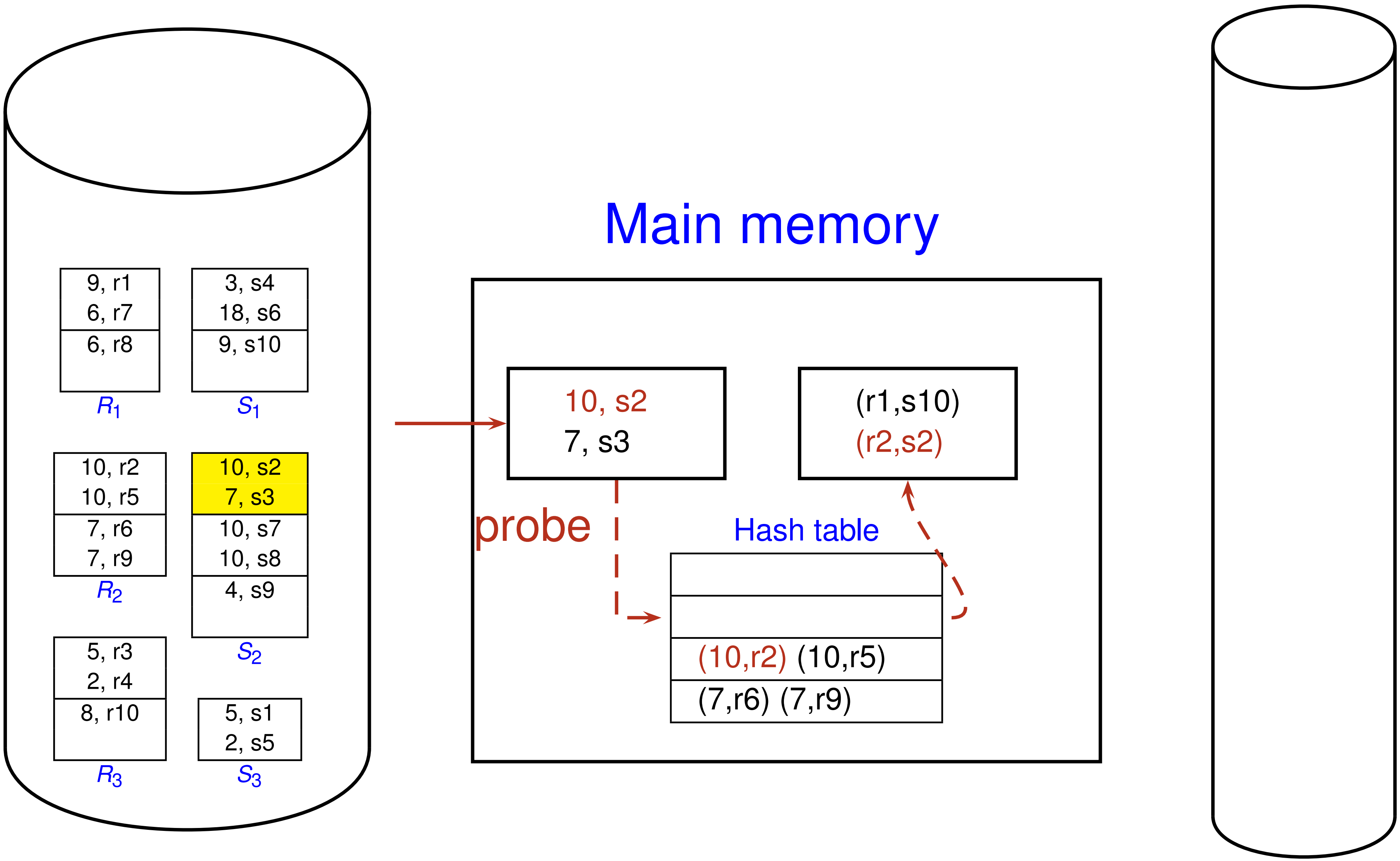
Grace Hash Join: Probing Phase



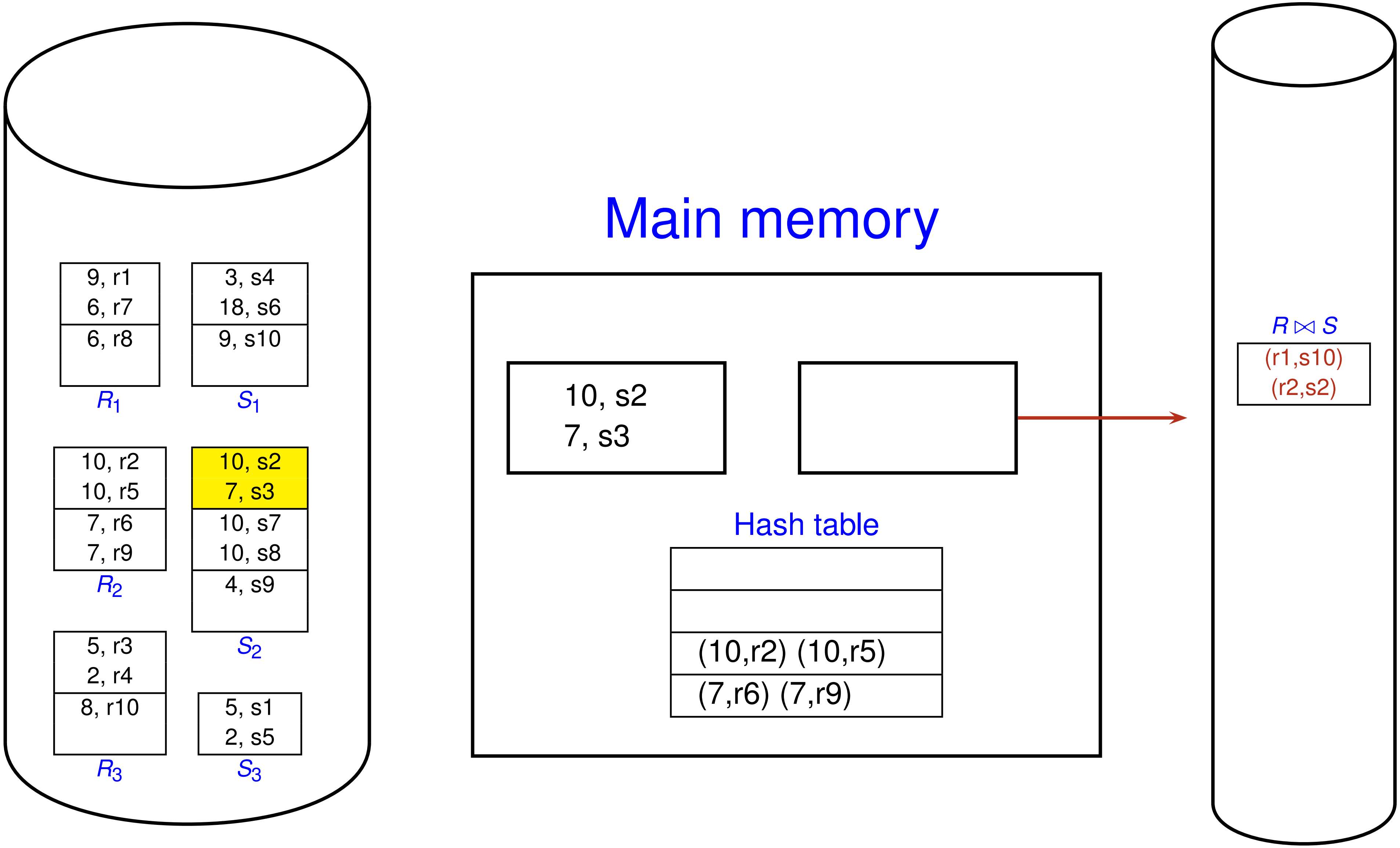
Grace Hash Join: Probing Phase



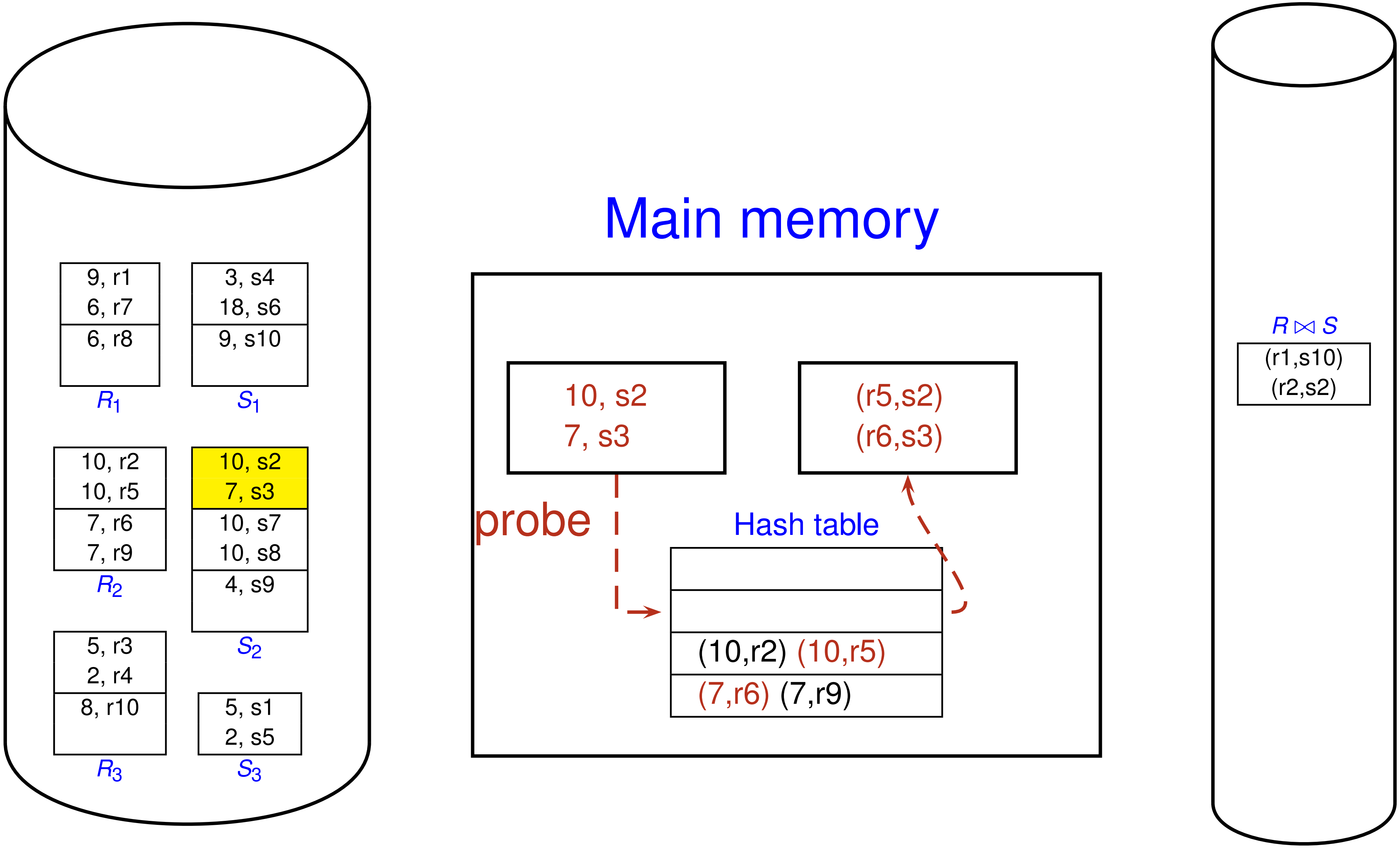
Grace Hash Join: Probing Phase



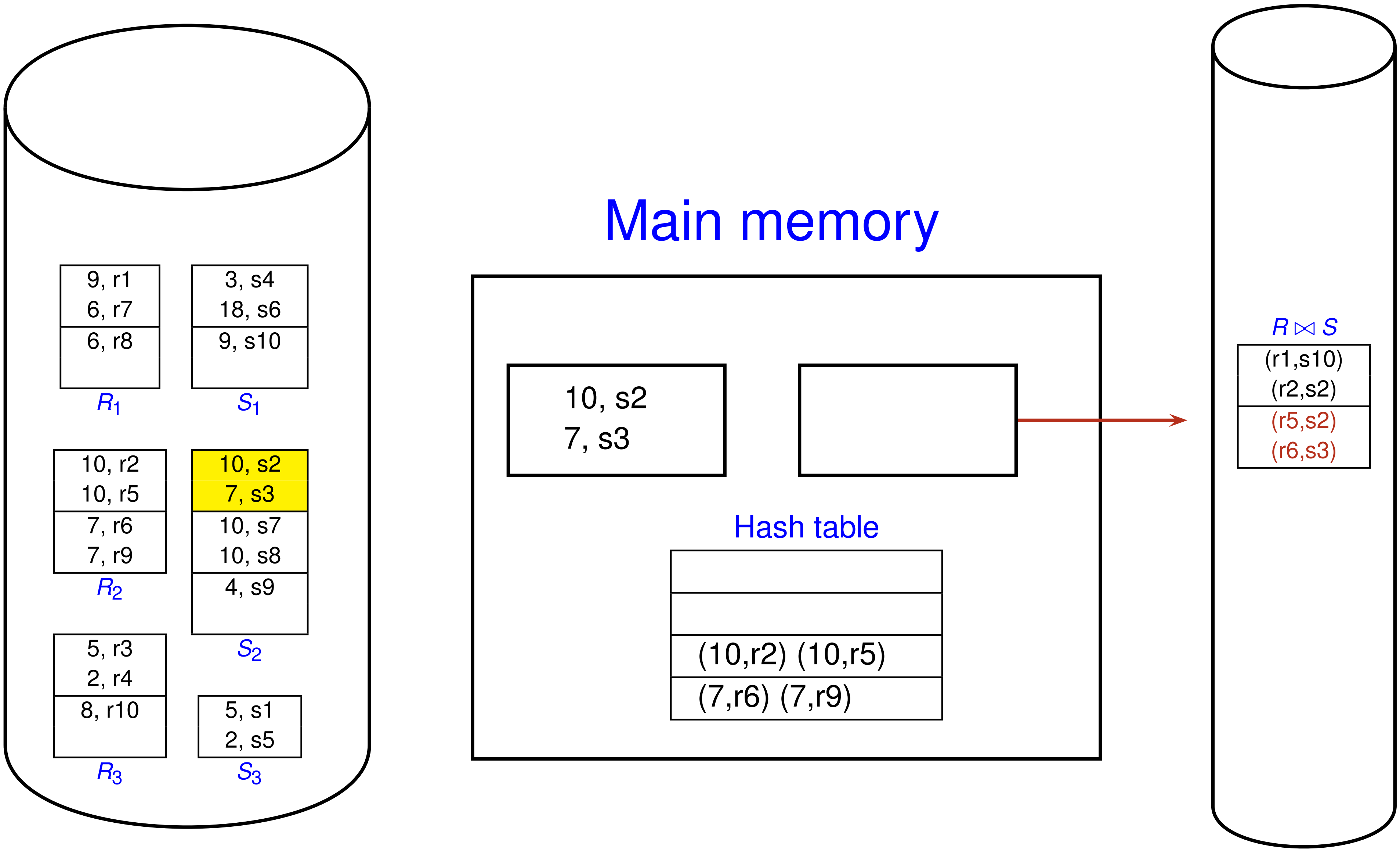
Grace Hash Join: Probing Phase



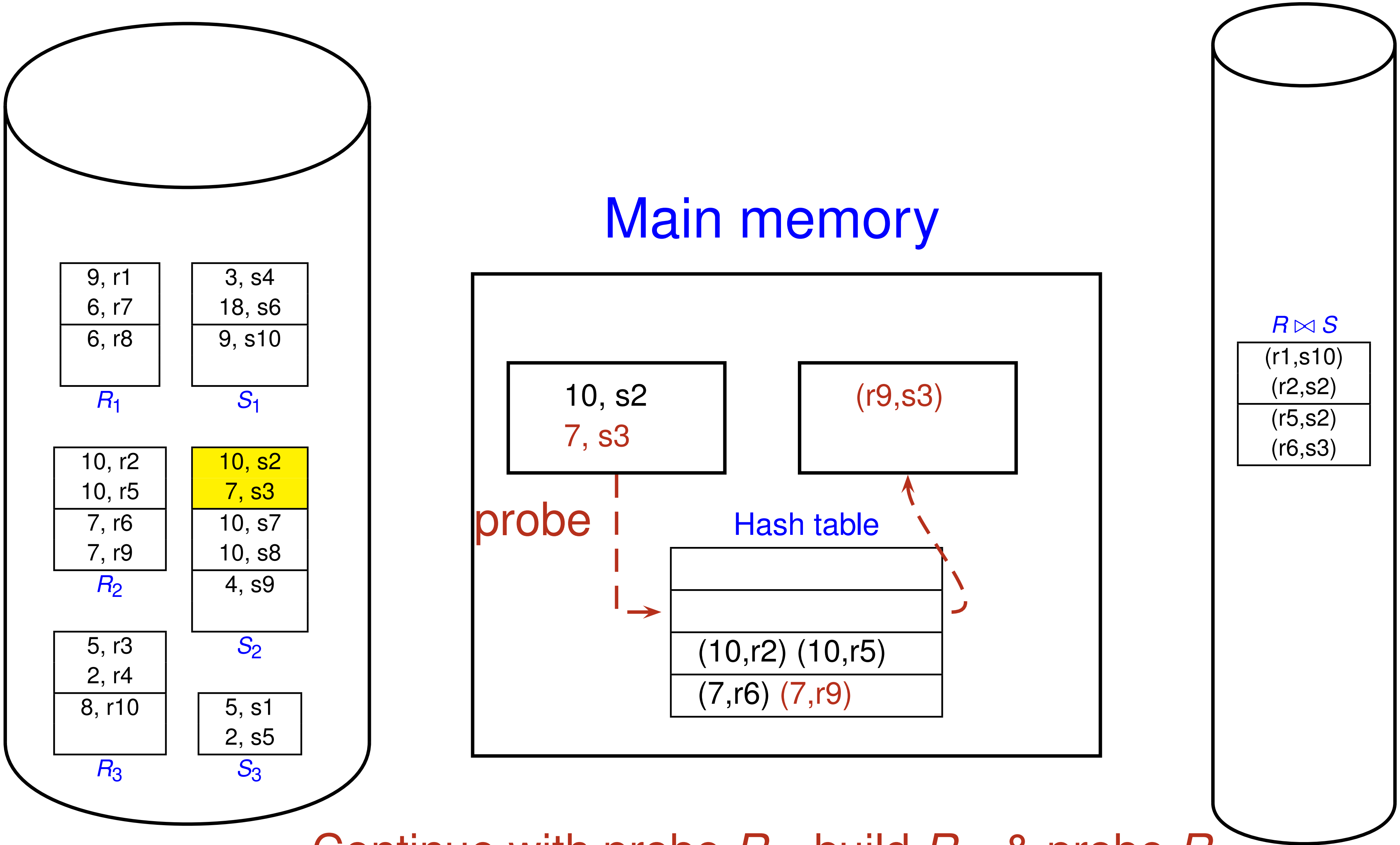
Grace Hash Join: Probing Phase



Grace Hash Join: Probing Phase



Grace Hash Join: Probing Phase



Continue with probe R_2, build R_3, & probe R_3

Grace Hash Join: Analysis

- To minimize size of each partition of R_i,
  - Set $k = B - 1$ given B buffer pages
- Assuming uniform hashing distribution,
  - size of each partition R_i is $\frac{|R|}{B-1}$
  - size of hash table for R_i is $\frac{f \times |R|}{B-1}$, where f is a fudge factor
  - During probing phase, $B > \frac{f \times |R|}{B-1} + 2$ (with one input buffer for S_i & one output buffer)
  - Approximately, $B > \sqrt{f \times |R|}$
- Partition overflow problem
  - Hash table for R_i does not fit in memory
  - Solution: recursively apply partitioning to overflow partitions
- I/O cost = Cost of partitioning phases + Cost of probing phase
  - I/O cost = $3(|R| + |S|)$ if there's no partition overflow problem

General Join Conditions

- Multiple equality-join conditions
  - Example: (R.A = S.A) `and` (R.B = S.B)
  - Algorithms:
    - **Index Nested Loop Join**: use index on all or some of join attributes
    - **Sort-Merge Join**: need to sort on combination of attributes
    - Other algorithms essentially unchanged
- Inequality-join conditions
  - Example: (R.A $<$ S.A)
  - Algorithms:
    - **Index Nested Loop Join**: requires a B^+-tree index
    - **Sort-Merge Join**: not applicable
    - **Hash-based Joins**: not applicable
    - Other algorithms essentially unchanged