Relational Operators

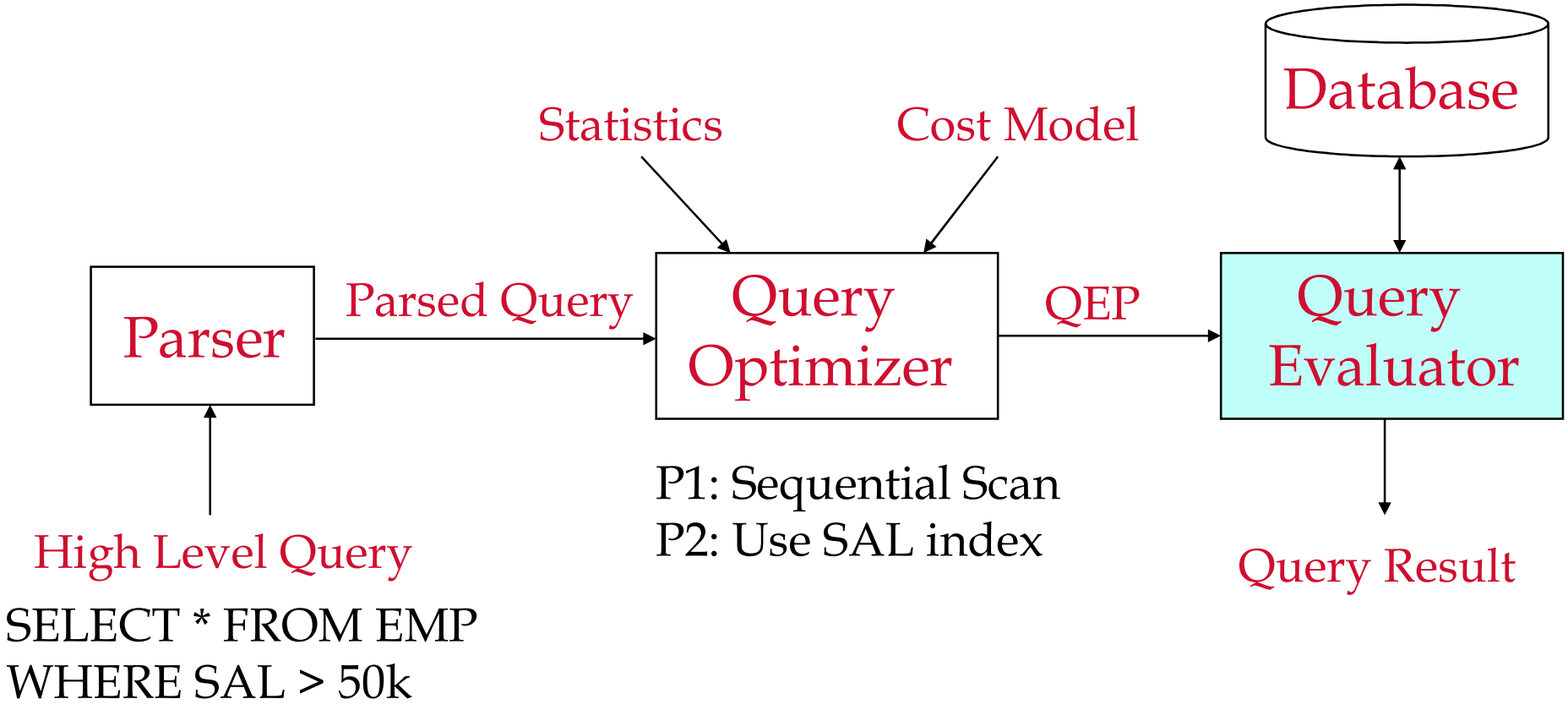
First comes thought; then organization of that thought, into ideas and plans; then transformation of those plans into reality. The beginning, as you will observe, is in your imagination.

Napolean Hill

Introduction

- We've covered the basic underlying storage, buffering, and indexing technology
  - Now we are ready to move on to query processing
- Some database operations are EXPENSIVE
- Can greatly improve performance by being "smart"
  - e.g., can speed up 1,000,000x over naïve approach
- Main approaches are:
  - clever implementation techniques for operators
  - exploit "equivalences" of relational operators
  - use statistics and cost models to choose among these

Steps of processing a high-level query



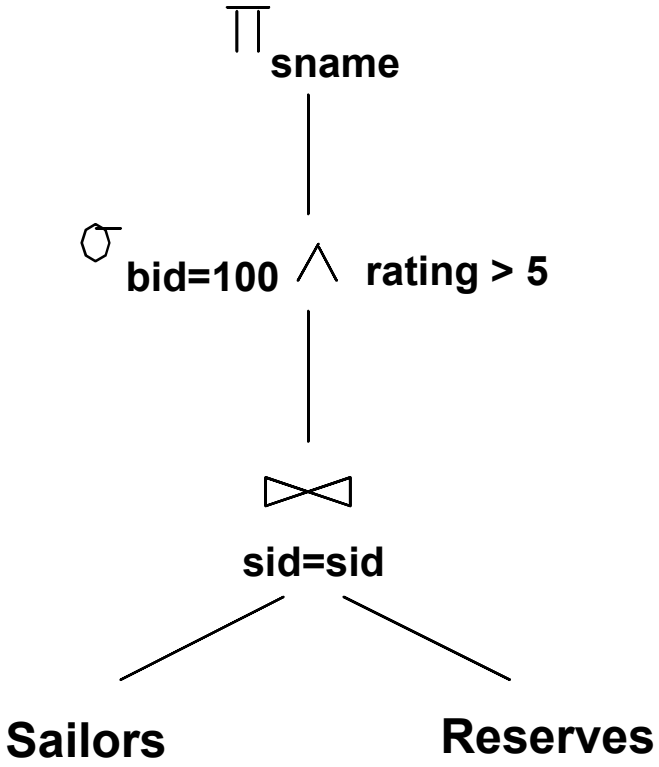
Relational Operations

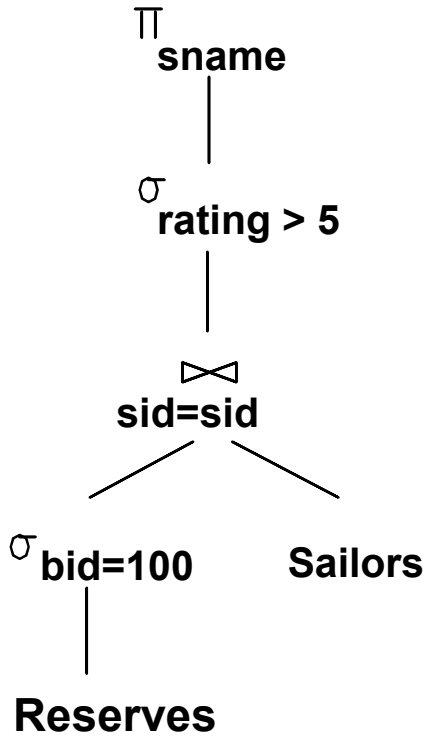
- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (Π) Deletes unwanted columns from relation.
  - *Join* (⋈) Allows us to combine two relations.
  - *Set-difference* (-) Tuples in reln. 1, but not in reln. 2.
  - *Union* (∪) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY

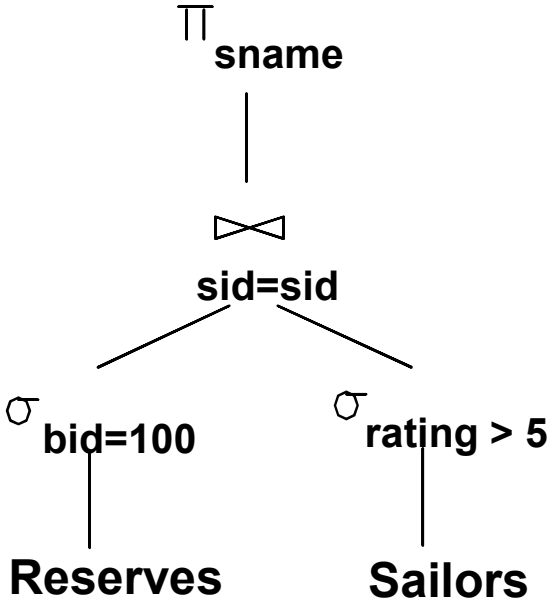
Since each op returns a relation, ops can be *composed*!
Queries that require multiple ops to be composed may be composed in different ways - thus *optimization* is necessary for good performance

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

Example







Paradigm

- Iteration-based
- Index
  - B+-tree, Hash
    - assume index entries to be (rid,pointer) pair
  - Clustered, Unclustered
- Sort
- Hash

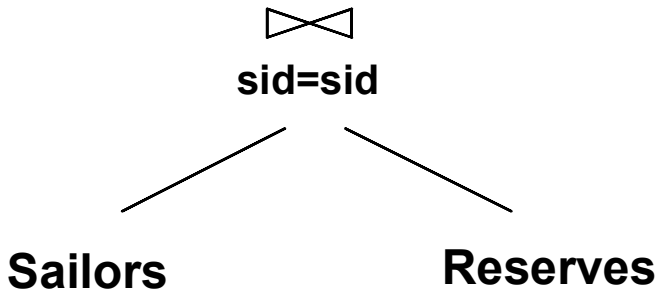
Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves (R):
  - ||R|| - number of tuples
  - |R| - number of pages
  - p_R tuples per page, |R| = M. Let p_R = 100, M = 1000, ||R|| = 100*1000
- Sailors (S):
  - p_S tuples per page, |S| = N. Let p_S = 80, N = 500, ||S|| = 80*500
- Cost metric: # of I/Os (pages)
  - We will ignore output costs in the following discussion

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid=S.sid
```



- In algebra: R ⋈ S
- Most frequently used operation; very costly operation
- ||R⋈S|| = ρ × (||R|| × ||S||) where ρ is the join selectivity

Join Example

Sailor

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Reserve

sid	bid	day	rname
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Query (join) output

sid	sname	rating	age	bid	day	rname
31	lubber	8	55.5	101	10/11/96	lubber
58	rusty	10	35.0	103	11/12/96	dustin

Join Algorithms

- Iteration-based
  - Block nested loop
- Index-based
  - Index nested loop
- Sort-based
  - Sort-merge join
- Partition-based
  - Hash join

Join Algorithms

- Things to consider when choosing an algorithm
  - Types of join predicates
    - Equality predicates (e.g., R.A = S.B)
    - Inequality predicates (e.g., R.A < S.B)
  - Sizes of join operands
  - Available buffer space
  - Available access methods

Simple (Tuple-based) Nested Loops Join

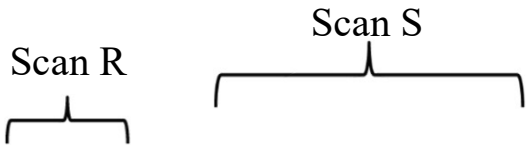
```
foreach tuple r in R do
        foreach tuple s in S do
                if r.sid == s.sid then add <r, s> to result
```

- For each tuple in the outer relation R, we scan the entire inner relation S
  - I/O Cost?
  - Memory?

Simple Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S do
                if r.sid == s.sid then add (r, s) to result
```

- For each tuple in the outer relation R, we scan the entire inner relation S



- Cost: |R| + ||R|| x |S|

$M + (p_R * M) * N = 1000 + 100*1000*500$ I/Os

Simple Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S do
                if r.sid == s.sid then add (r, s) to result
```

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: $M + (p_R * M) * N = 50{,}001{,}000$ I/Os
  - Memory: Only 3 buffers needed

Can we do better??

Page-based Nested Loop Join

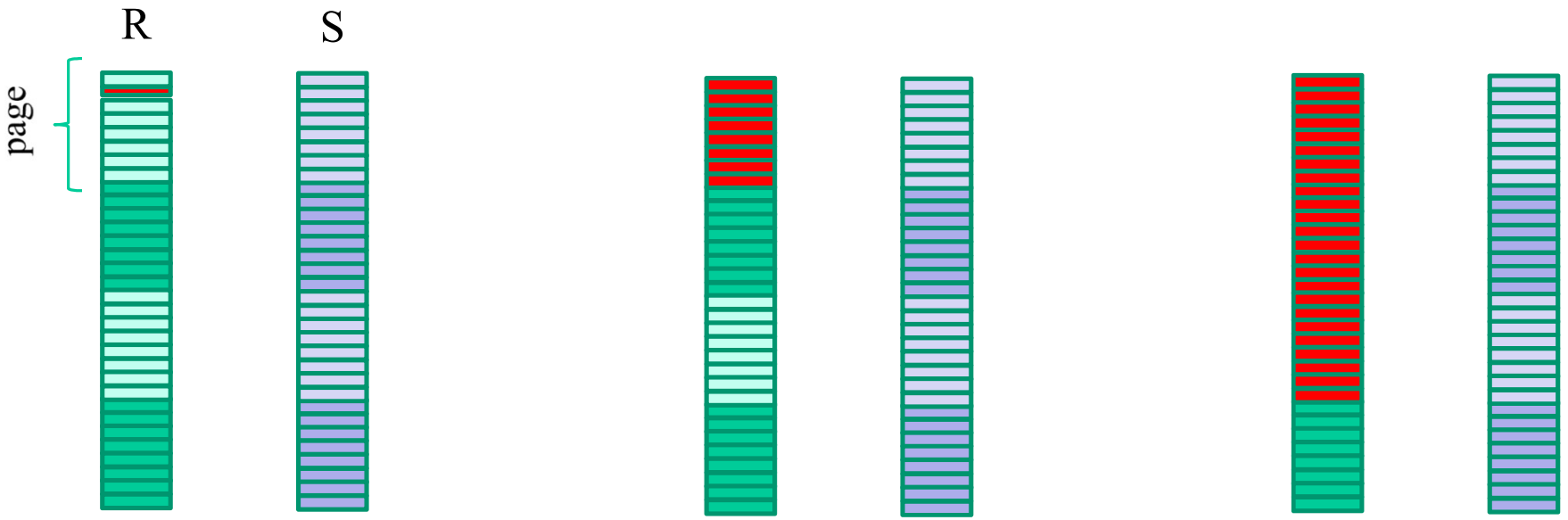
for each page P_R of R do
  for each page P_S of S do
    for each tuple r ∈ P_R do
      for each tuple s ∈ P_S do
        if (r.sid == s.sid) then output (r ,s) to result

- I/O cost = |R| + |R| x |S| = M + M * N = 1000 + 1000*500 I/Os
- Memory = 3 pages

Can we do better??

Block Nested Loops Join

- Motivation: How to better exploit buffer space to minimize number of I/Os?



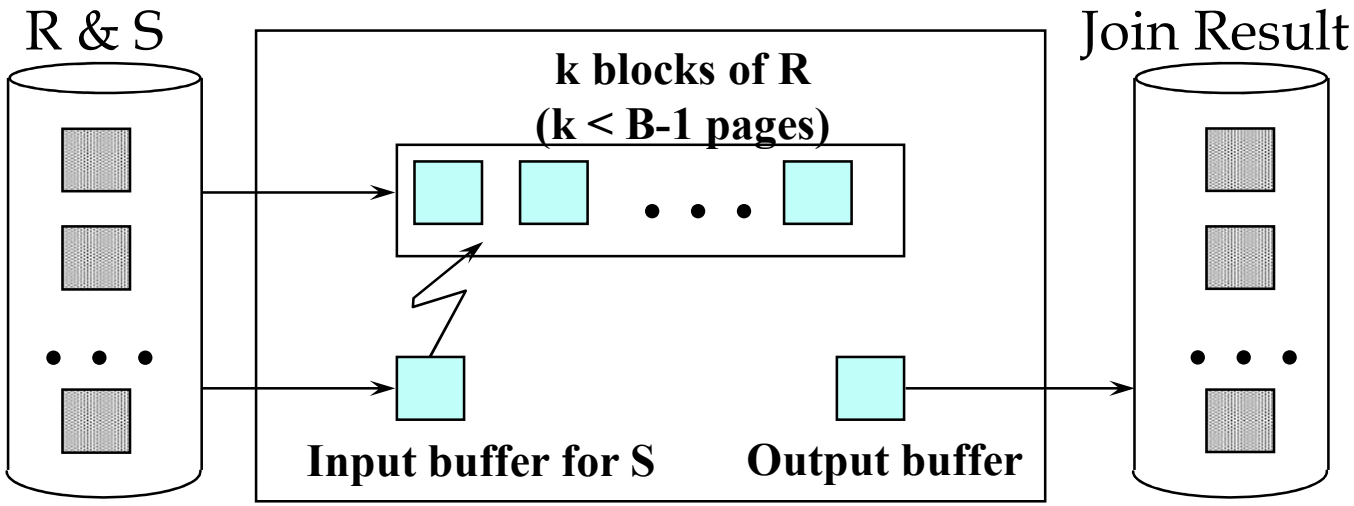
For each R tuple, scan S
(memory size = 3 pages)
Number of iterations of S: ||R||

For each page of R tuples,
scan S (memory = 3 pages)
Number of iterations: |R|

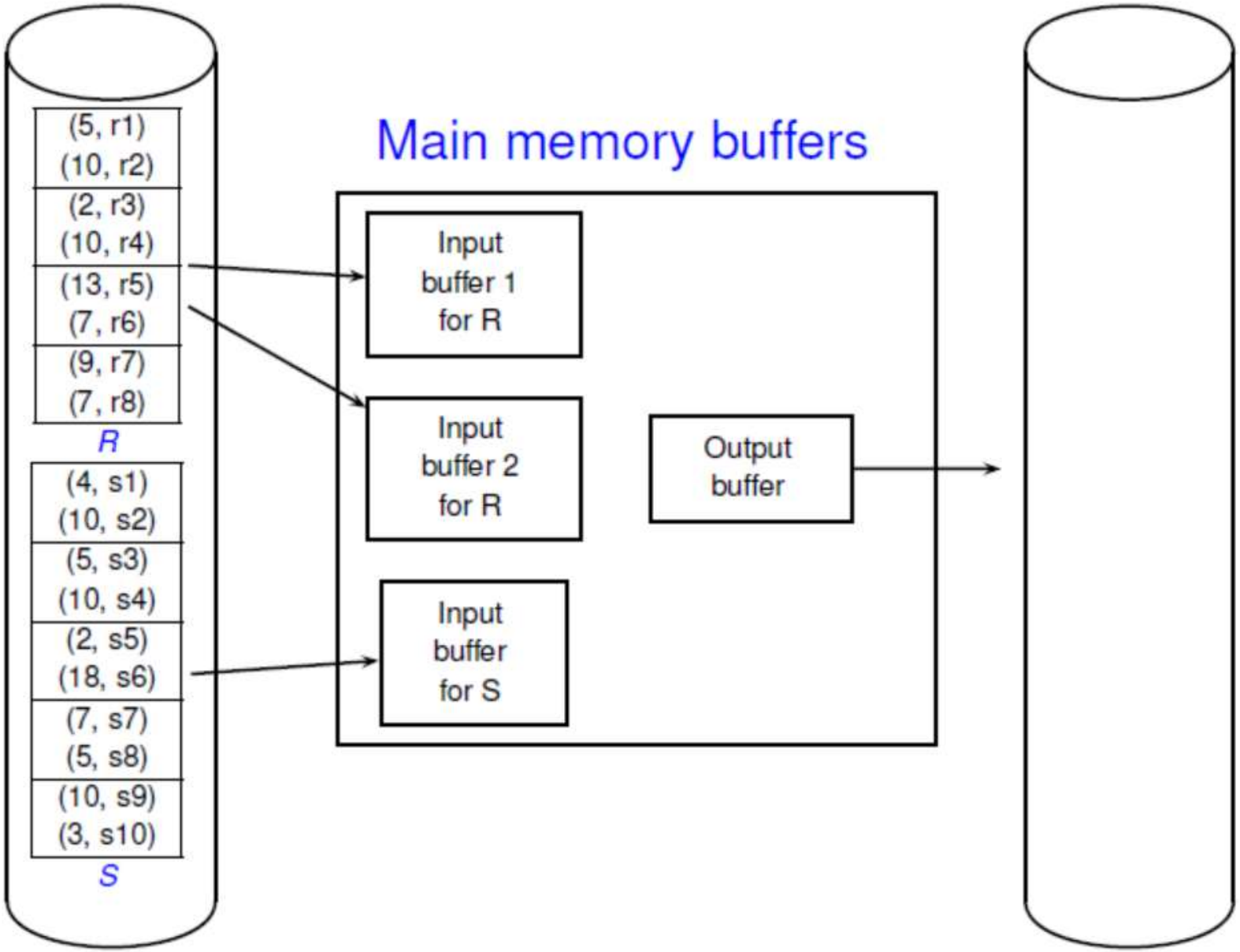
For each block of R tuples,
scan S (memory > 3 pages)
block size (B): buffer size – 2
Number of iterations: |R|/B

Block Nested Loops Join

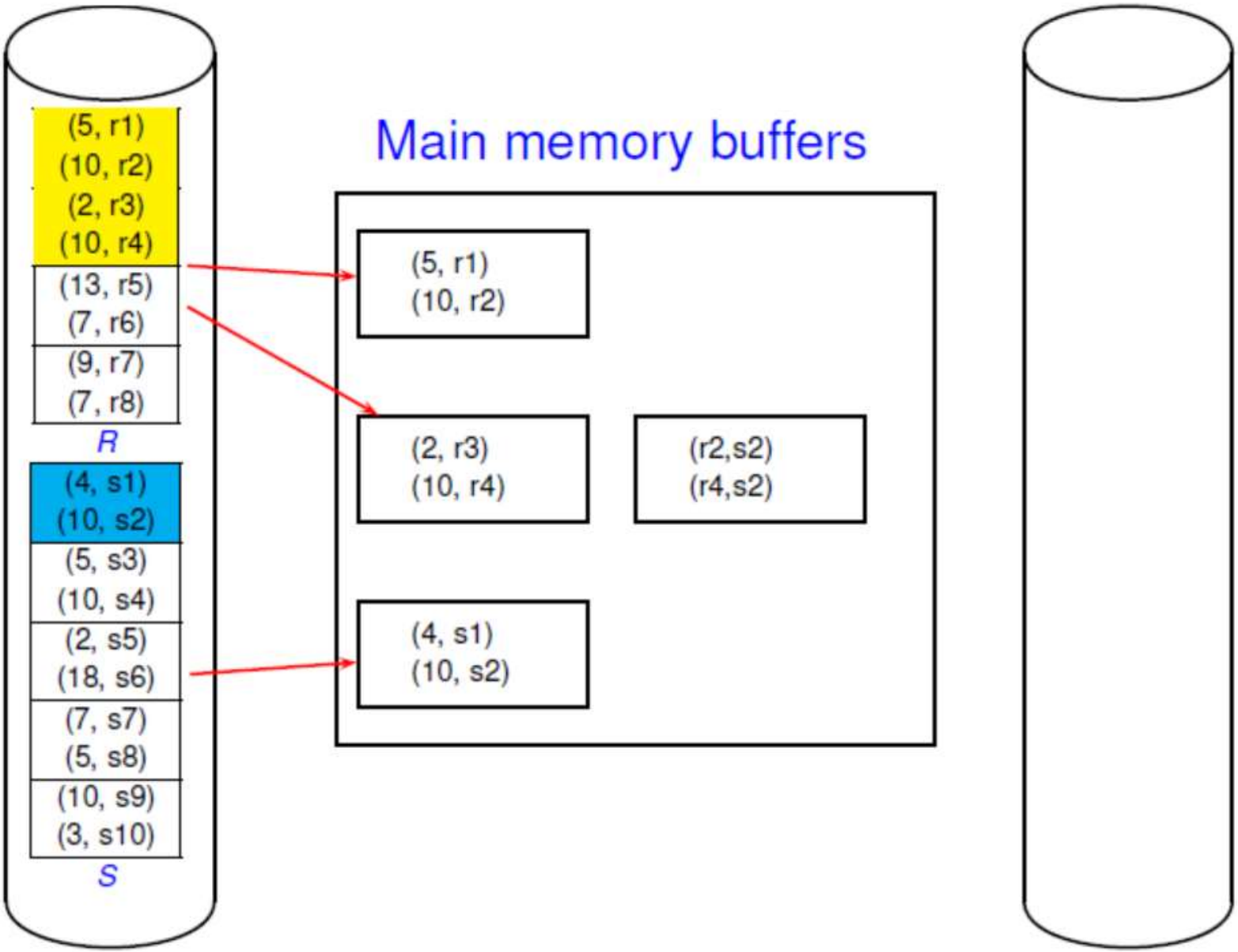
- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages (B -2) to hold "block" of outer R
  - For each matching tuple r in R-block, s in S-page, add (r, s) to result. Then read next R-block, scan S, etc



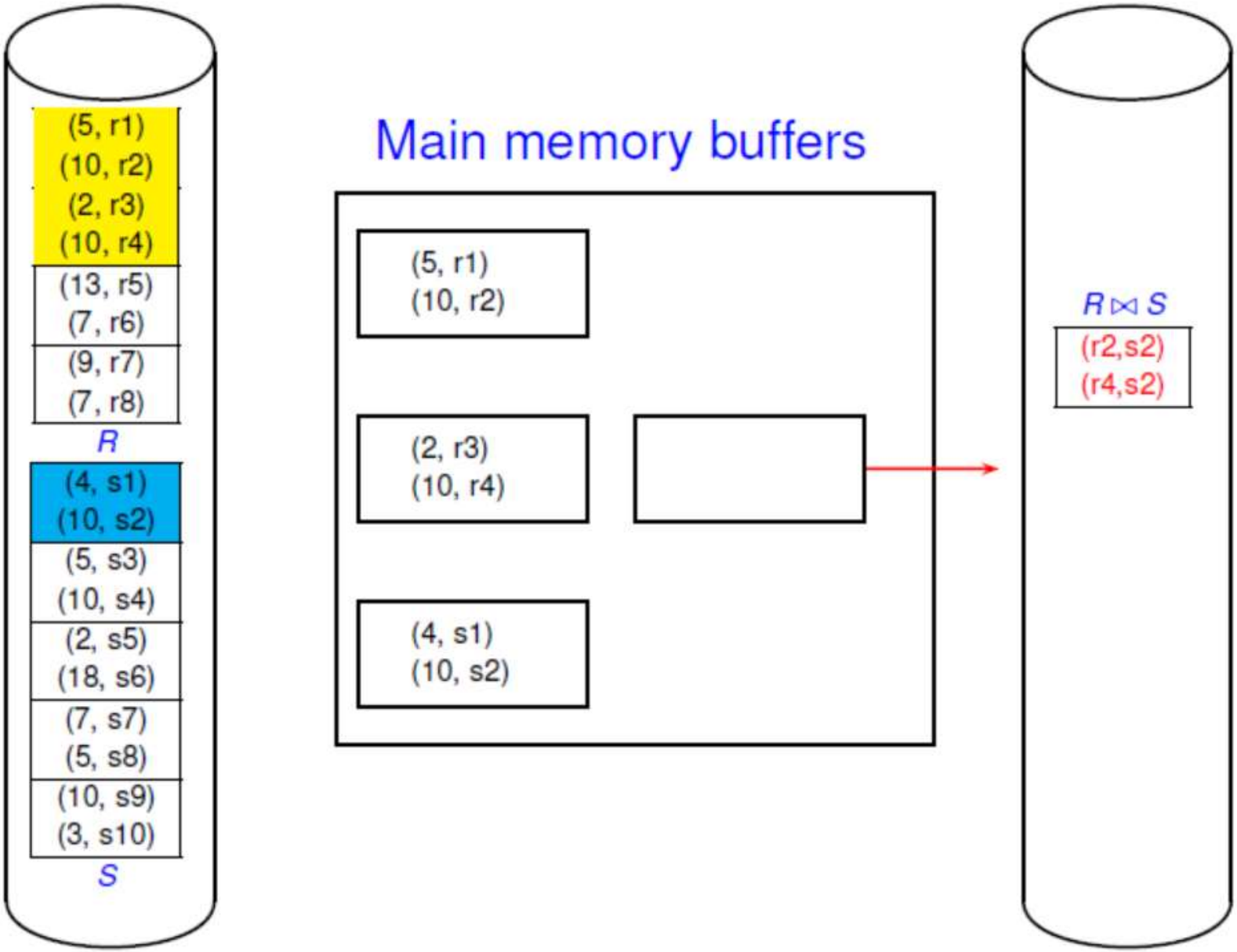
Block Nested Loop Join: Example



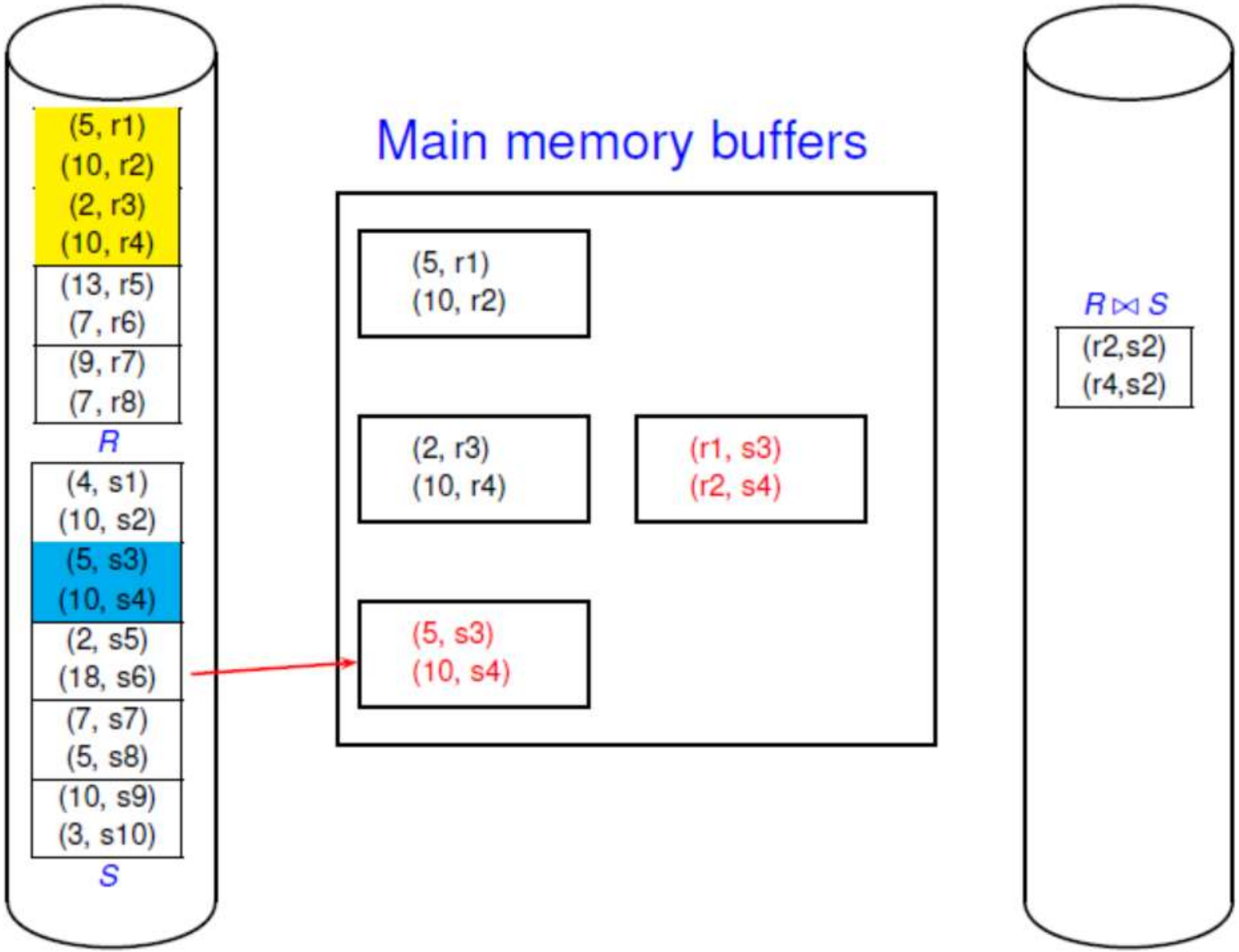
Block Nested Loop Join: Example



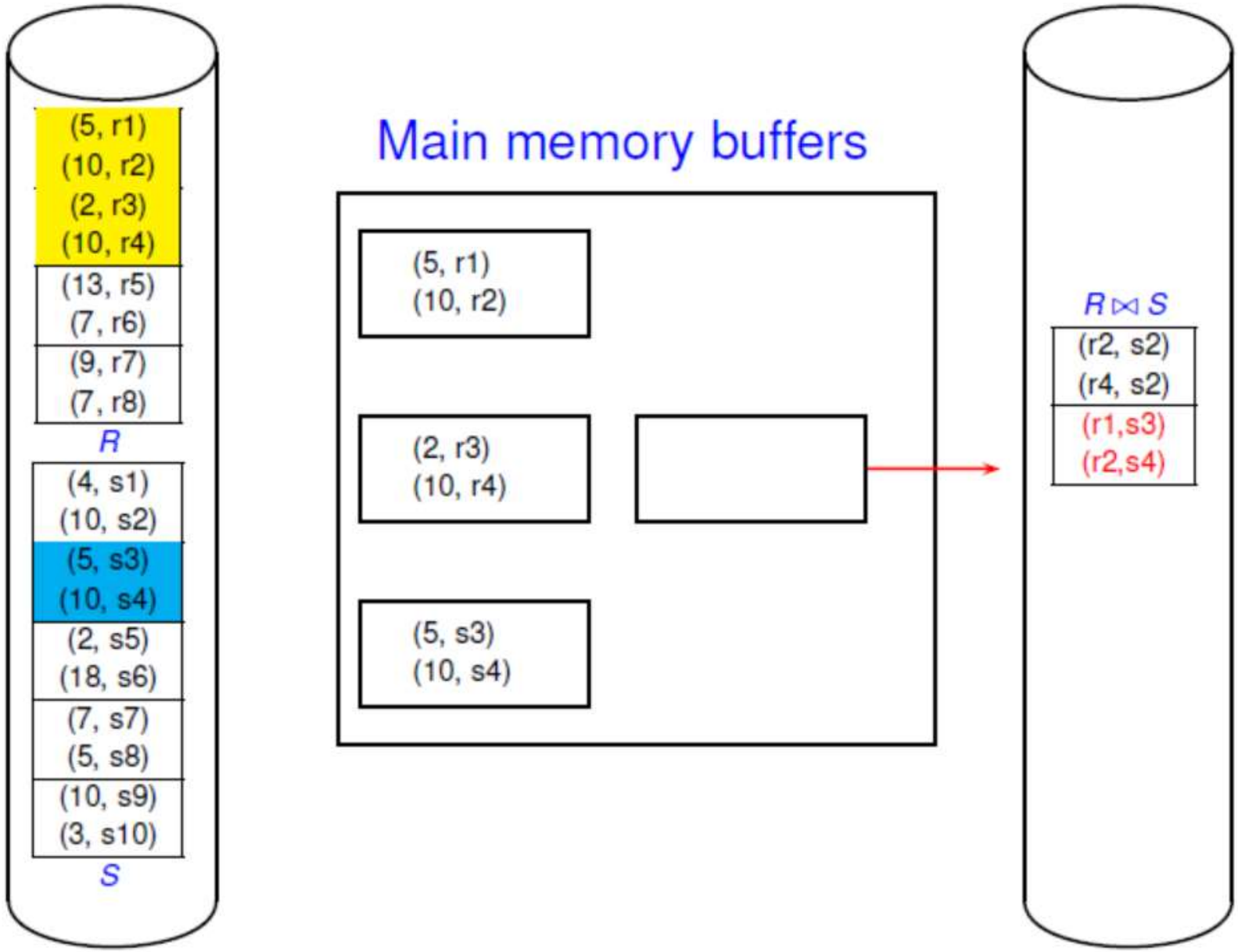
Block Nested Loop Join: Example



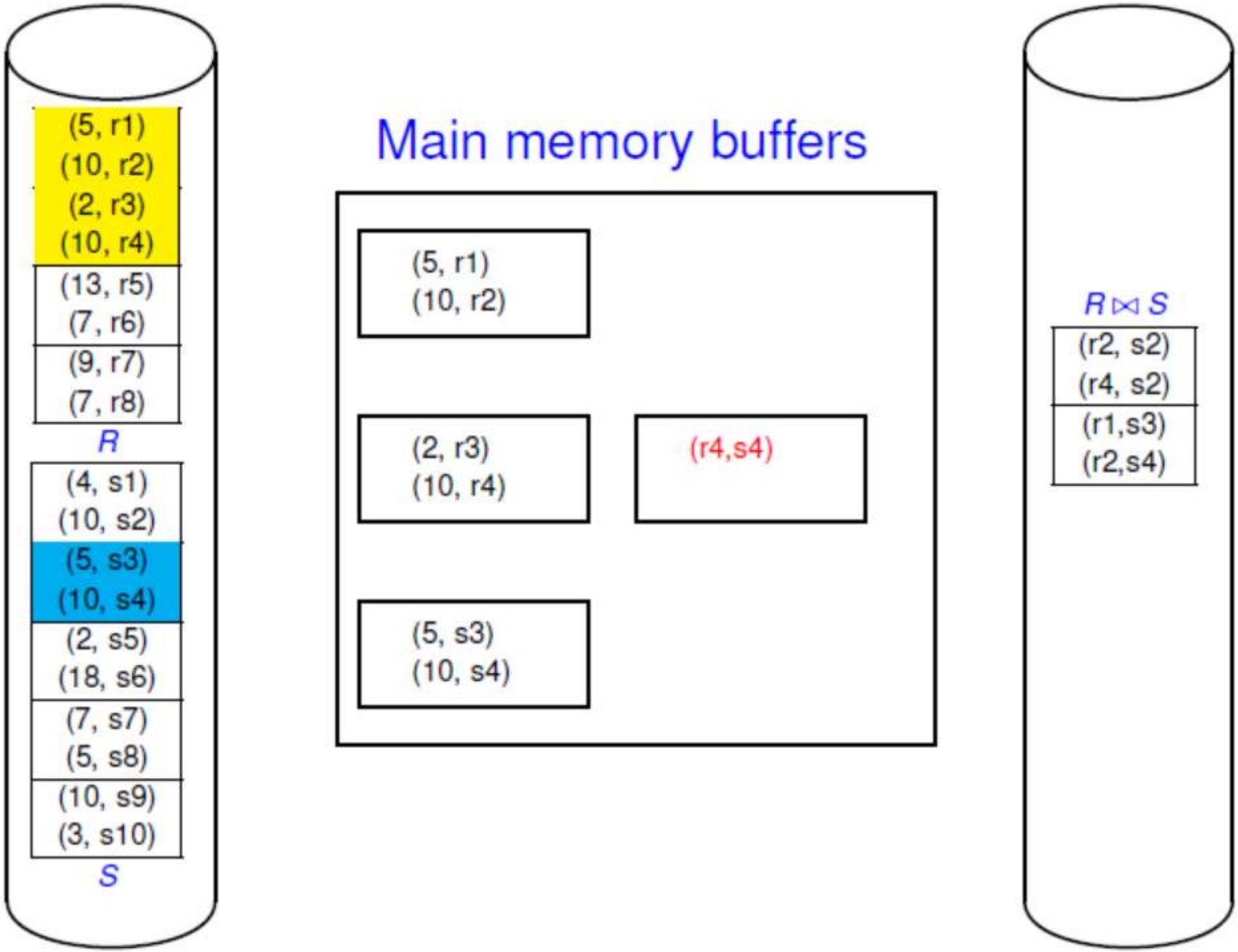
Block Nested Loop Join: Example



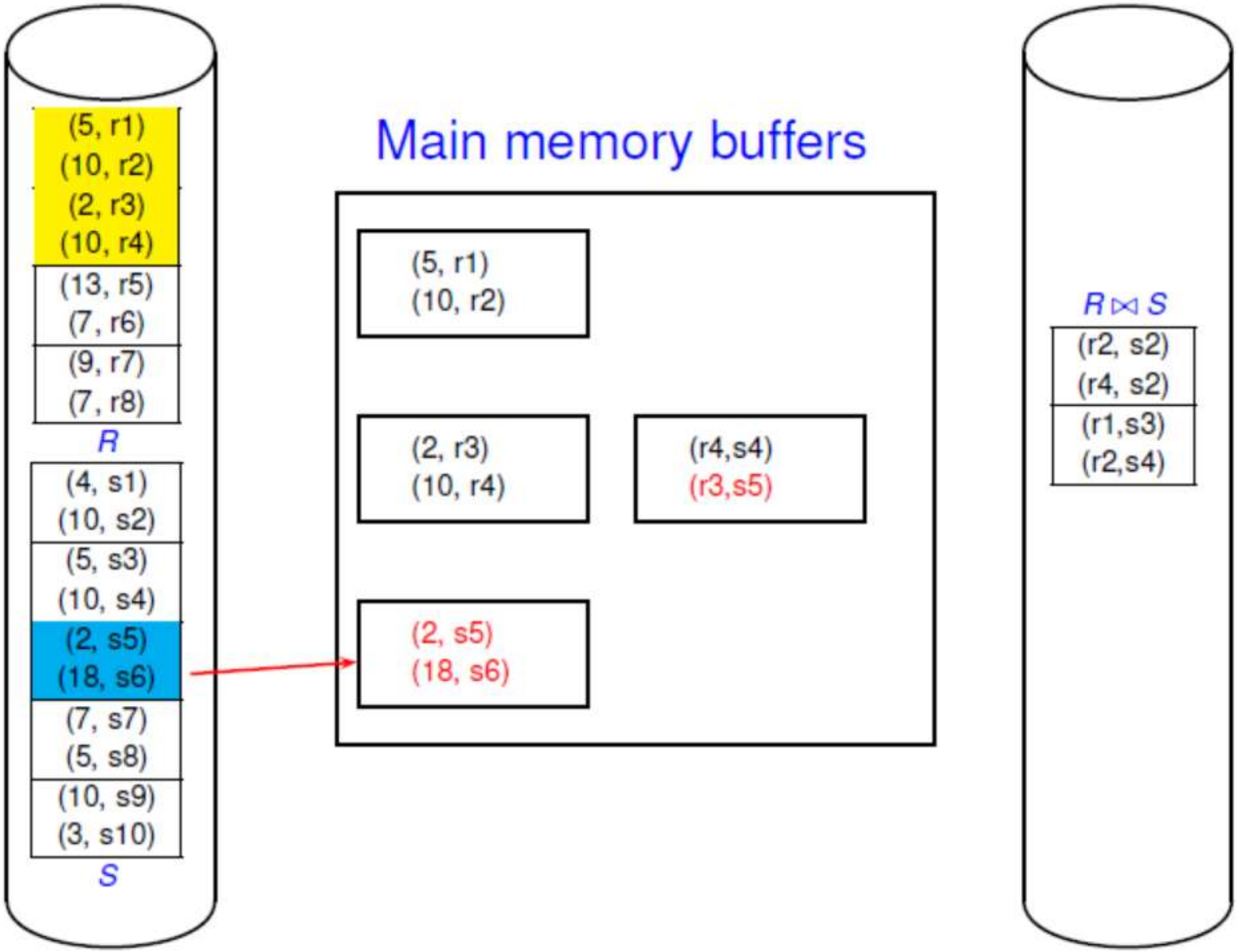
Block Nested Loop Join: Example



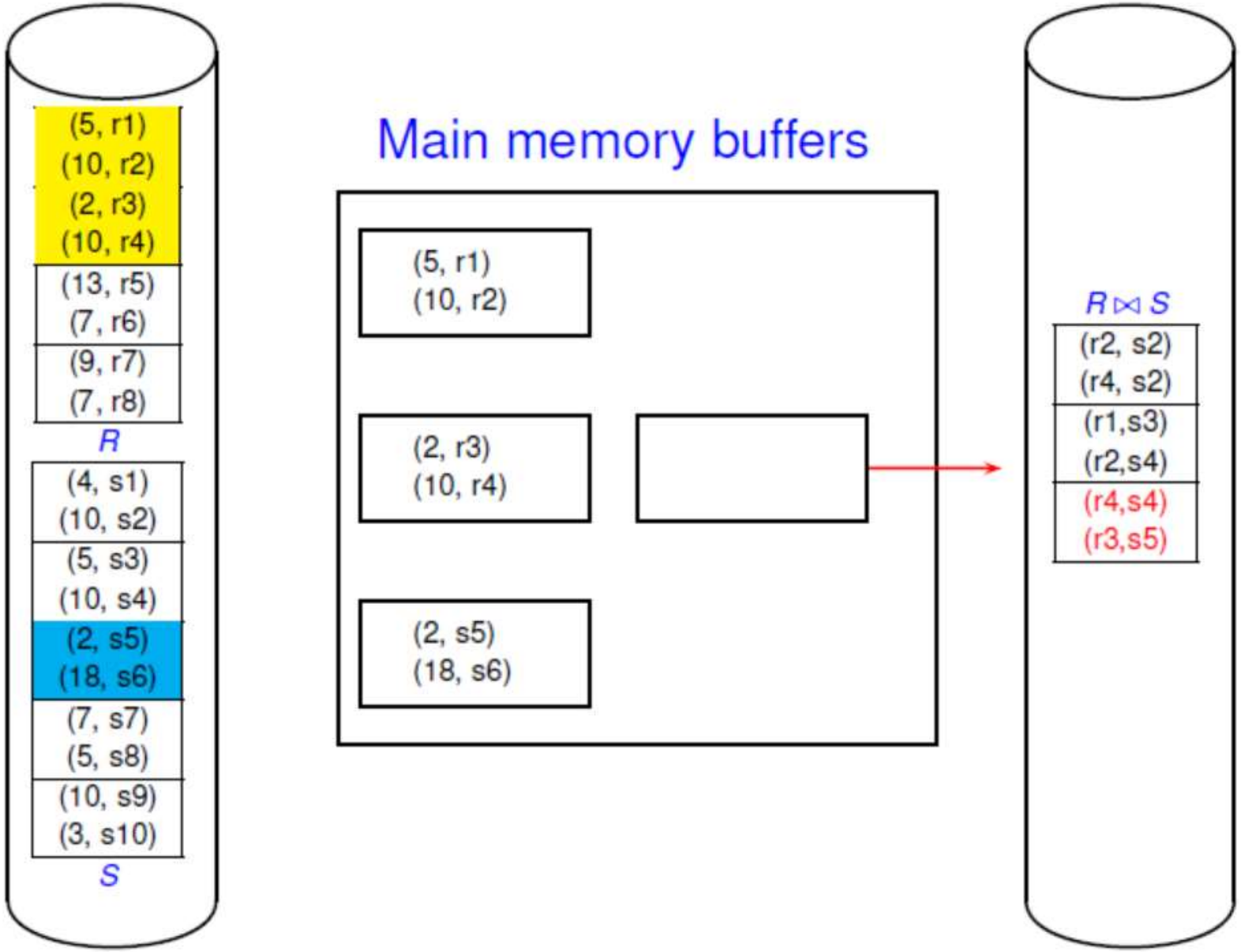
Block Nested Loop Join: Example



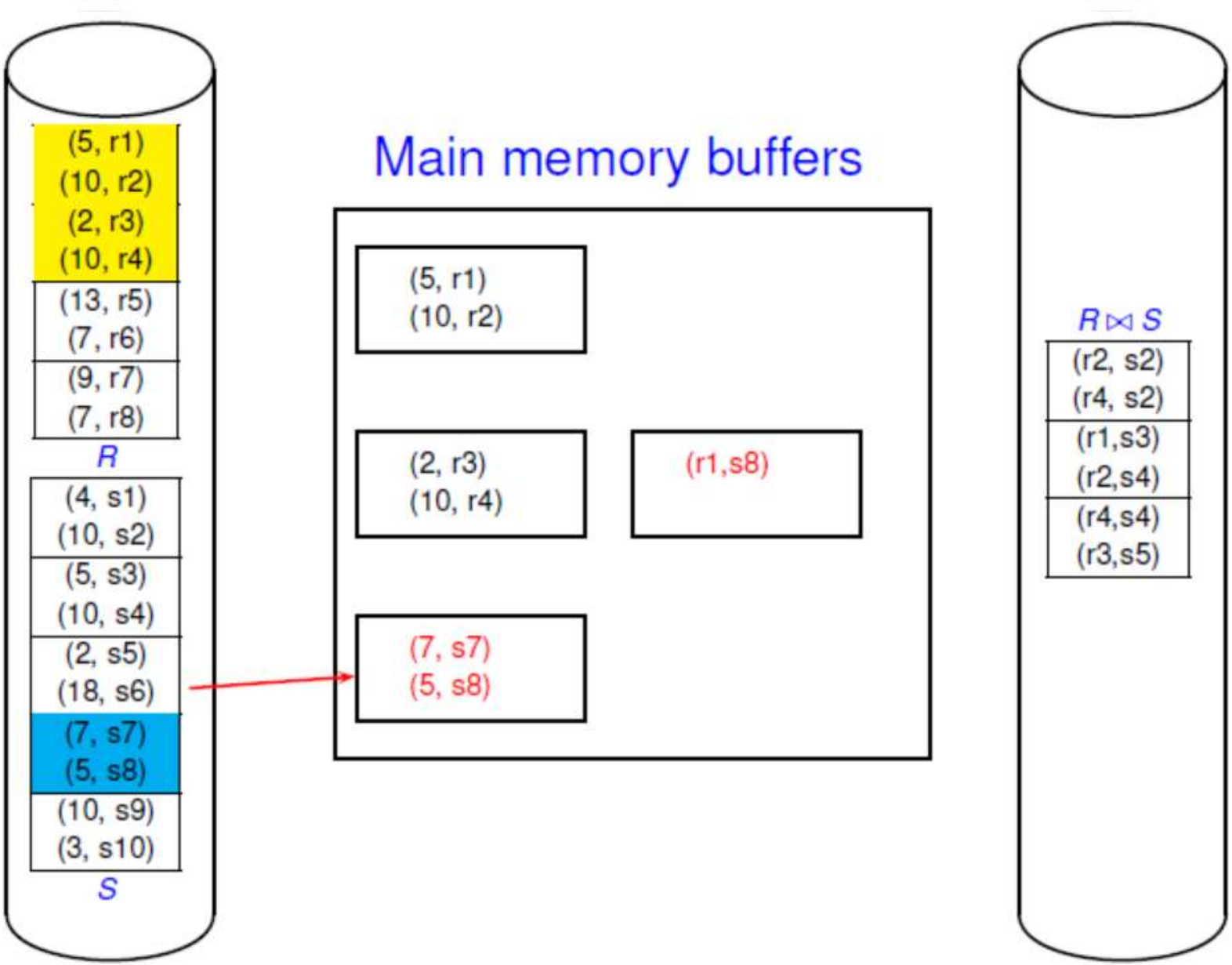
Block Nested Loop Join: Example



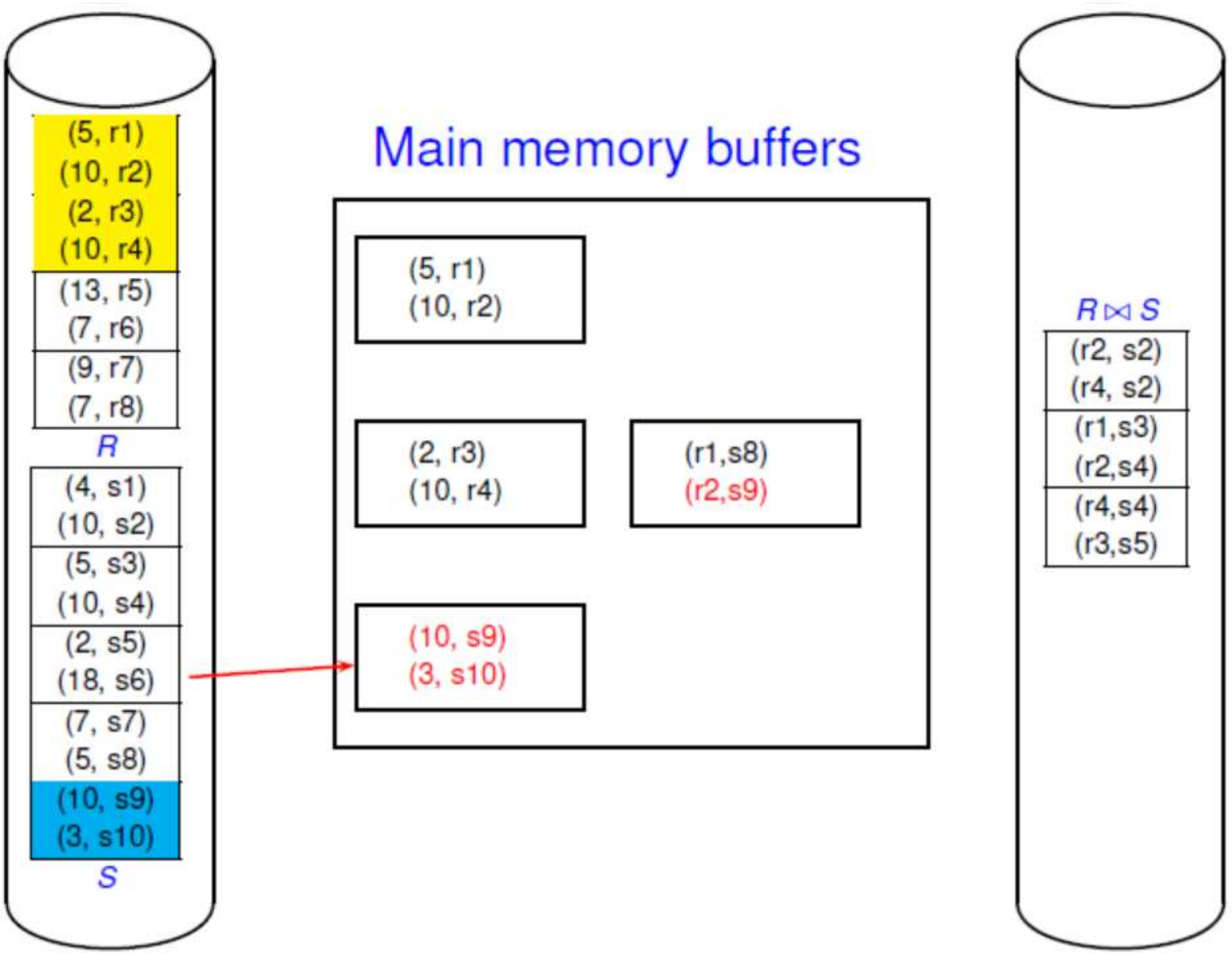
Block Nested Loop Join: Example



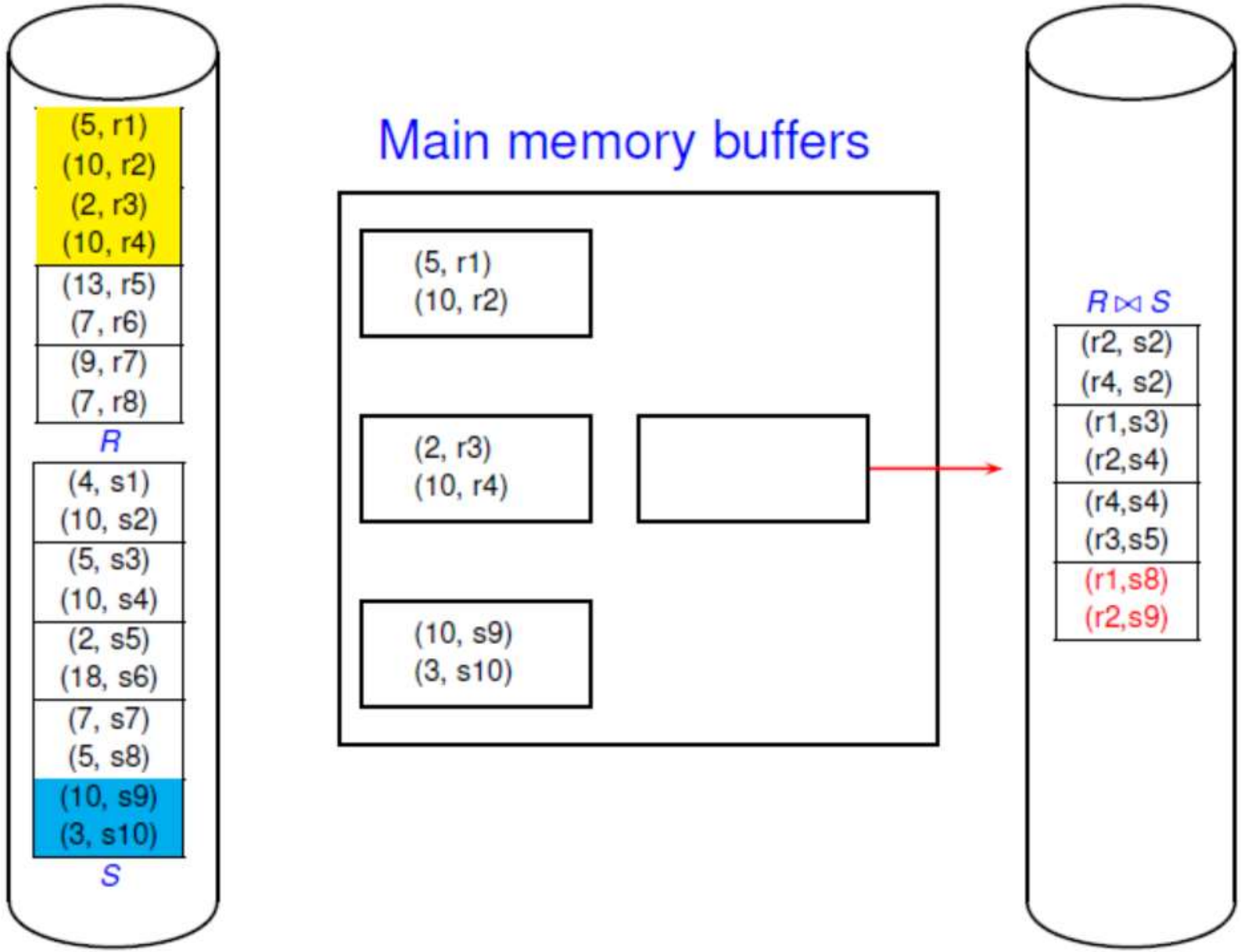
Block Nested Loop Join: Example



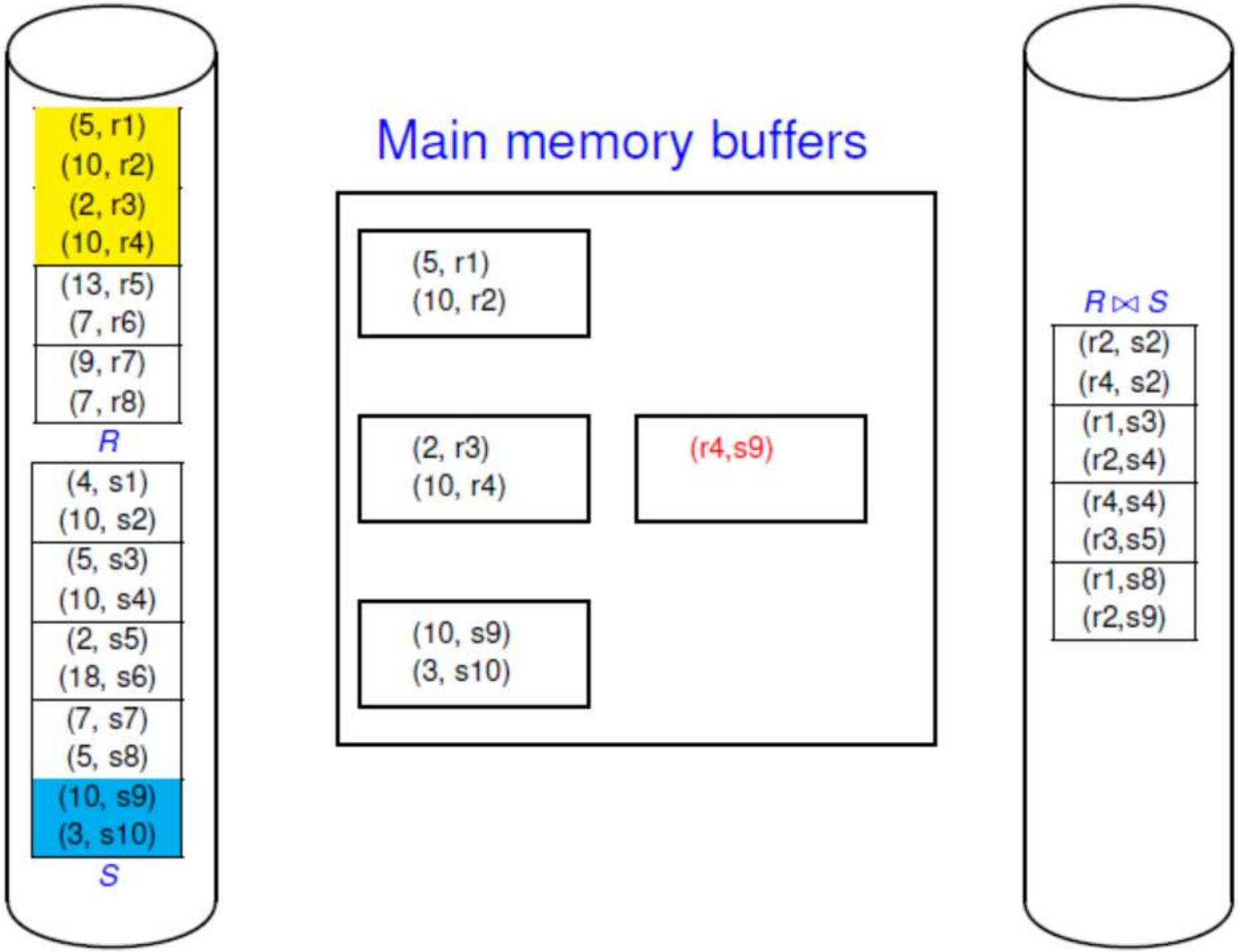
Block Nested Loop Join: Example



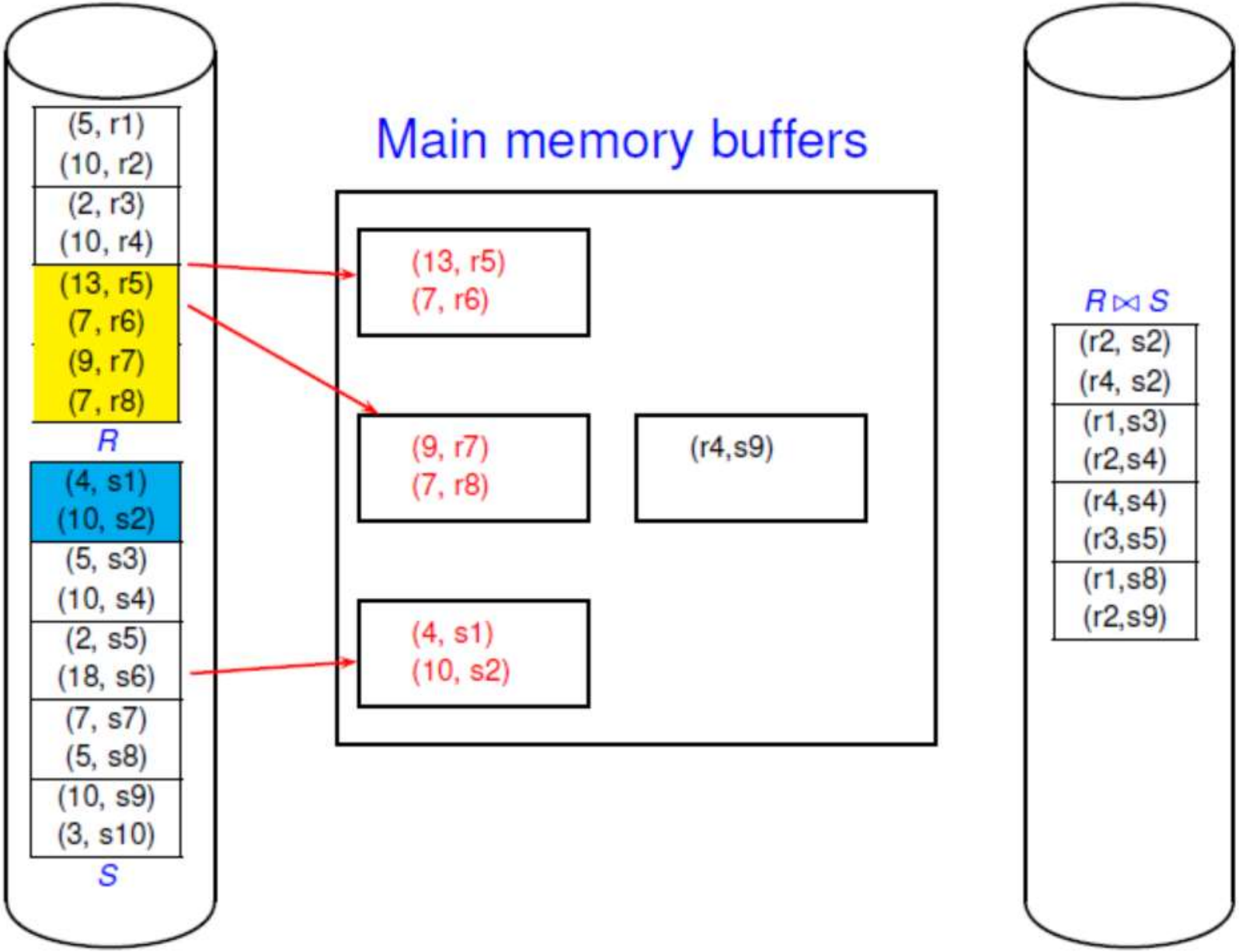
Block Nested Loop Join: Example



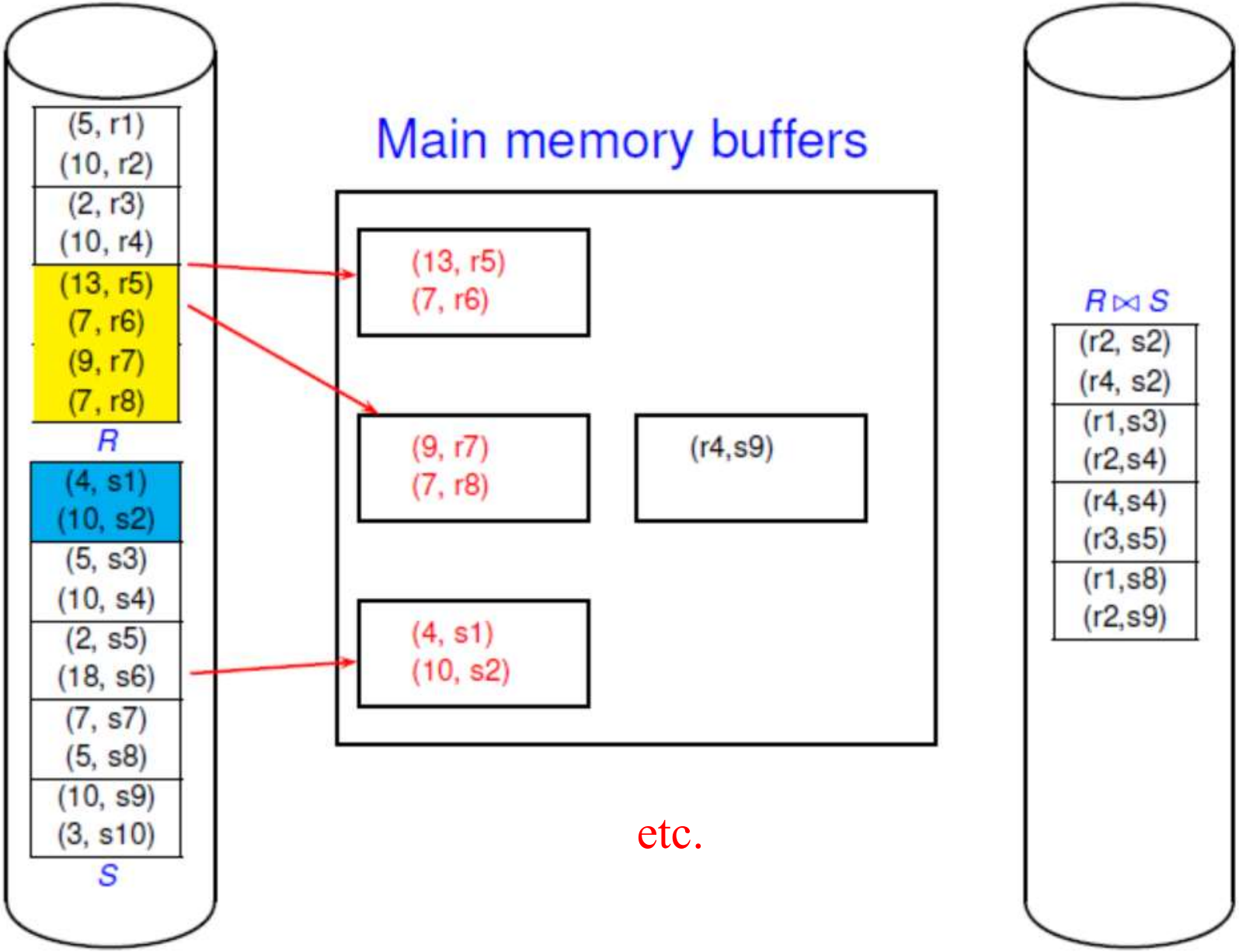
Block Nested Loop Join: Example



Block Nested Loop Join: Example



Block Nested Loop Join: Example



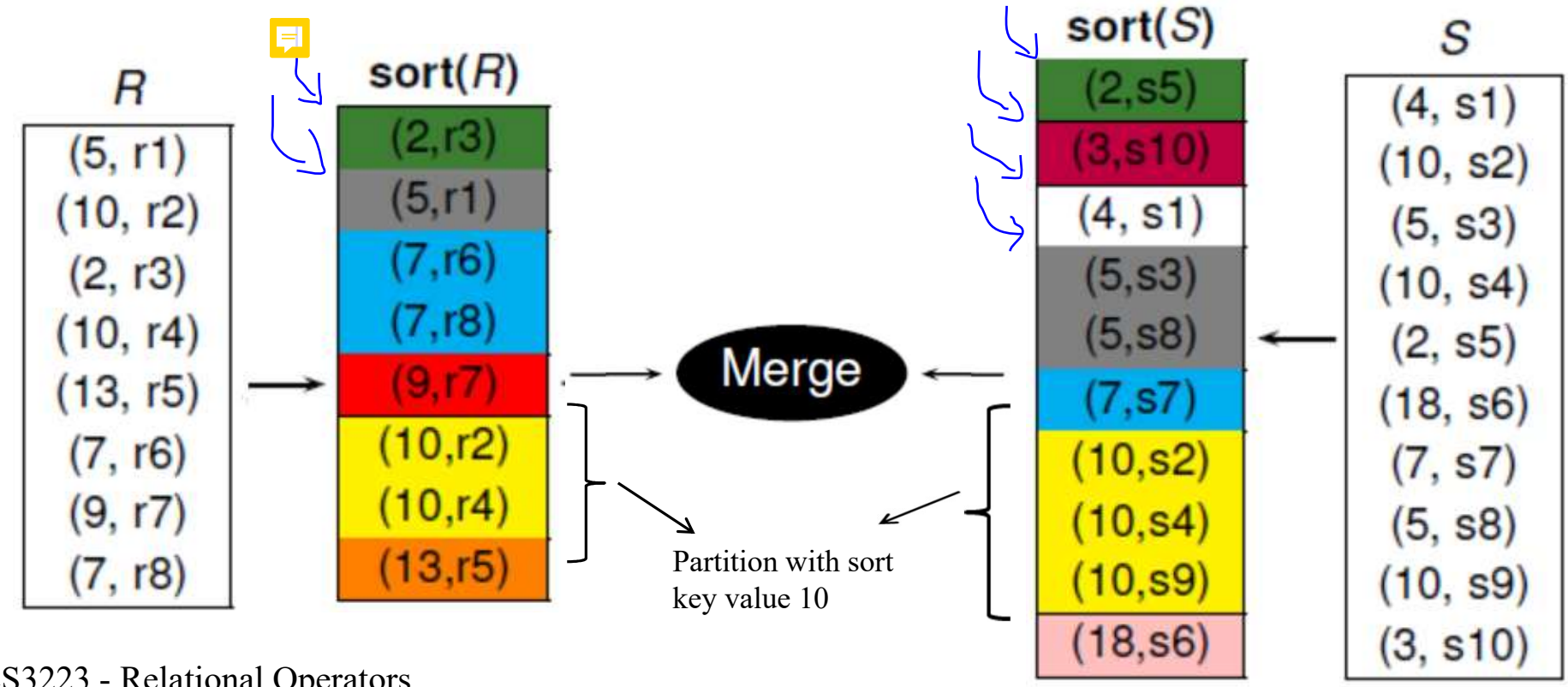
Examples of Block Nested Loops

- Cost: size of outer + #outer blocks * size of inner
  - #outer blocks = ⌈no. of pages in outer relation / block size⌉
- With R as outer, block size of 100 pages (buffer size = 102):
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks
  - Per block of R, we scan S; 10*500 I/Os
  - Join cost = 6000 I/Os
  - If block size for just 90 pages of R, scan S 12 times
- With 100-page block of S as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks
  - Per block of S, we scan R; 5*1000 I/Os
  - Join cost = 5500 I/Os

Ordering of inner/outer relations affects performance!

Sort-Merge Join

- Idea:
  - Sort R and S on the join column
    - A sorted relation R consists of (implicit) partitions of Ri of records where r, r' $\in R_i$ iff r and r' have the same values for the join attribute(s)
  - Scan them to do a "merge" (on join col.), and output result tuples



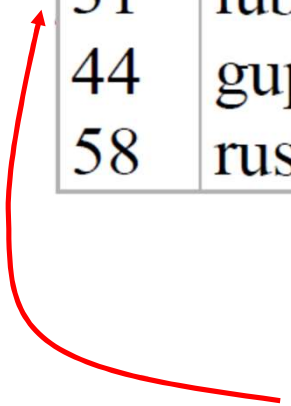
Sort-Merge Join

- Idea: Sort R and S on the join column, then scan them to do a "merge" (on join col.), and output result tuples
  - Advance scan of R until current R-tuple's sort key $\geq$ current S tuple's sort key, then advance scan of S until current S-tuple's key $\geq$ current R tuple's key; do this until current R tuple's key = current S tuple's key
  - At this point, all R tuples with same value in R_i (current R partition) and all S tuples with same value in S_j (current S partition) match; output (r, s) for all pairs of such tuples
  - Then resume scanning R and S
- R is scanned once; each S partition is scanned once per matching R tuple (Multiple scans of an S partition are likely to find needed pages in buffer)

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin



What if we have another record with sid = 31 (if sid is not the key), e.g, (31, jonny, 9, 37)

Cost of Sort-Merge Join

- I/O Cost = Cost of sorting R + Cost of sorting S + Merging cost
- Cost to sort R = $2|R|\ (\ 1 + \lceil \log_{B-1} \lceil |R|/B \rceil \rceil\ )$ for external merge sort
  - B = number of buffers
- Cost to sort S has a similar expression
- If each S partition is scanned at most once during merging
  - Merging cost = |R| + |S|
- Best case?
- Worst case?
  - Occurs when each tuple of R requires scanning entire S!
  - Merging cost = |R| + |R| x |S| (can be reduced using block-nested loops)

Cost of Sort-Merge Join (Example)

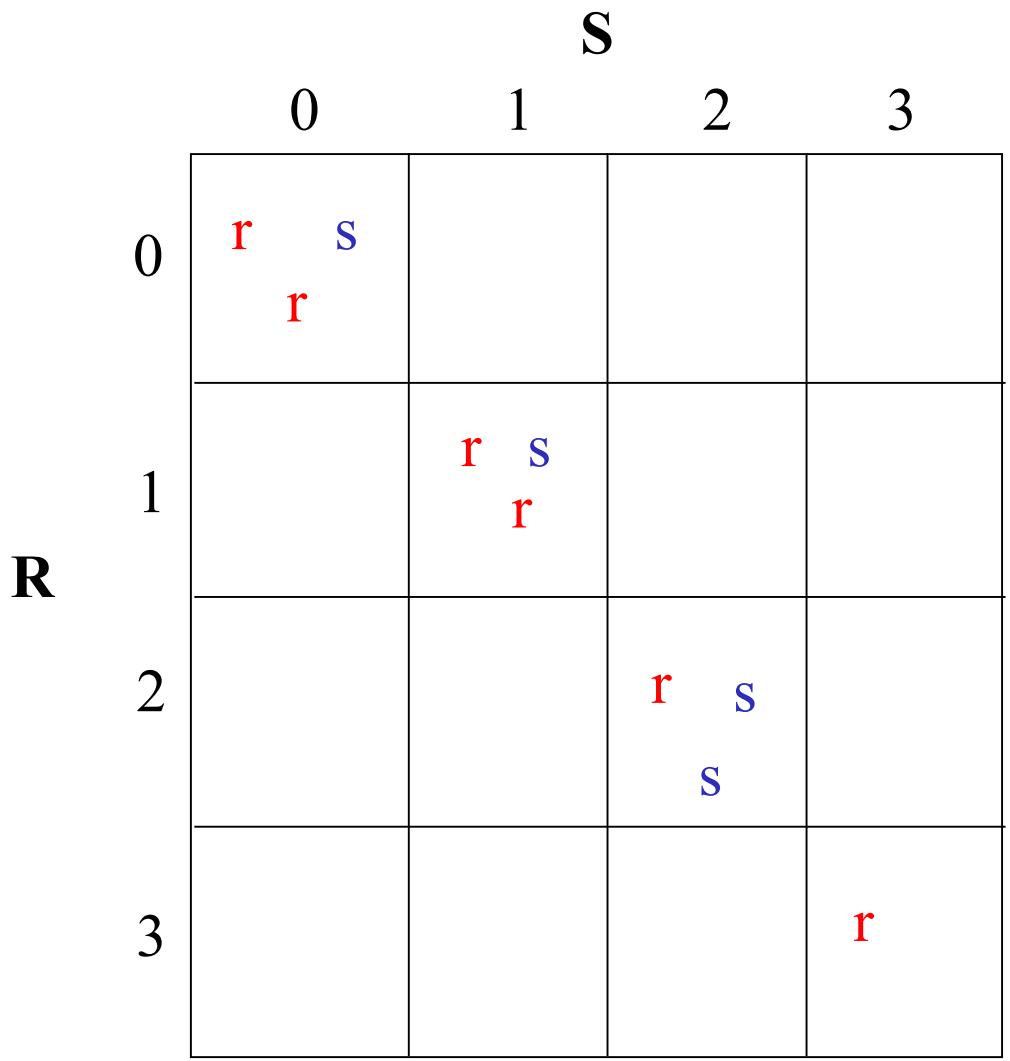
- Reserves (R):
  - p_R tuples per page, M pages. p_R = 100. M = 1000
- Sailors (S):
  - p_S tuples per page, N pages. p_S = 80. N = 500

- Cost: $2M*K_1 + 2N*K_2 + (M+N)$
  - K_1 and K_2 are the number of passes to sort R and S respectively
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both R and S can be sorted in 2 passes
  - total join cost = 2*1000*2 + 2*500*2 + 1000 + 500 = 7500

(BNL cost: 2500 to 15000 I/Os)

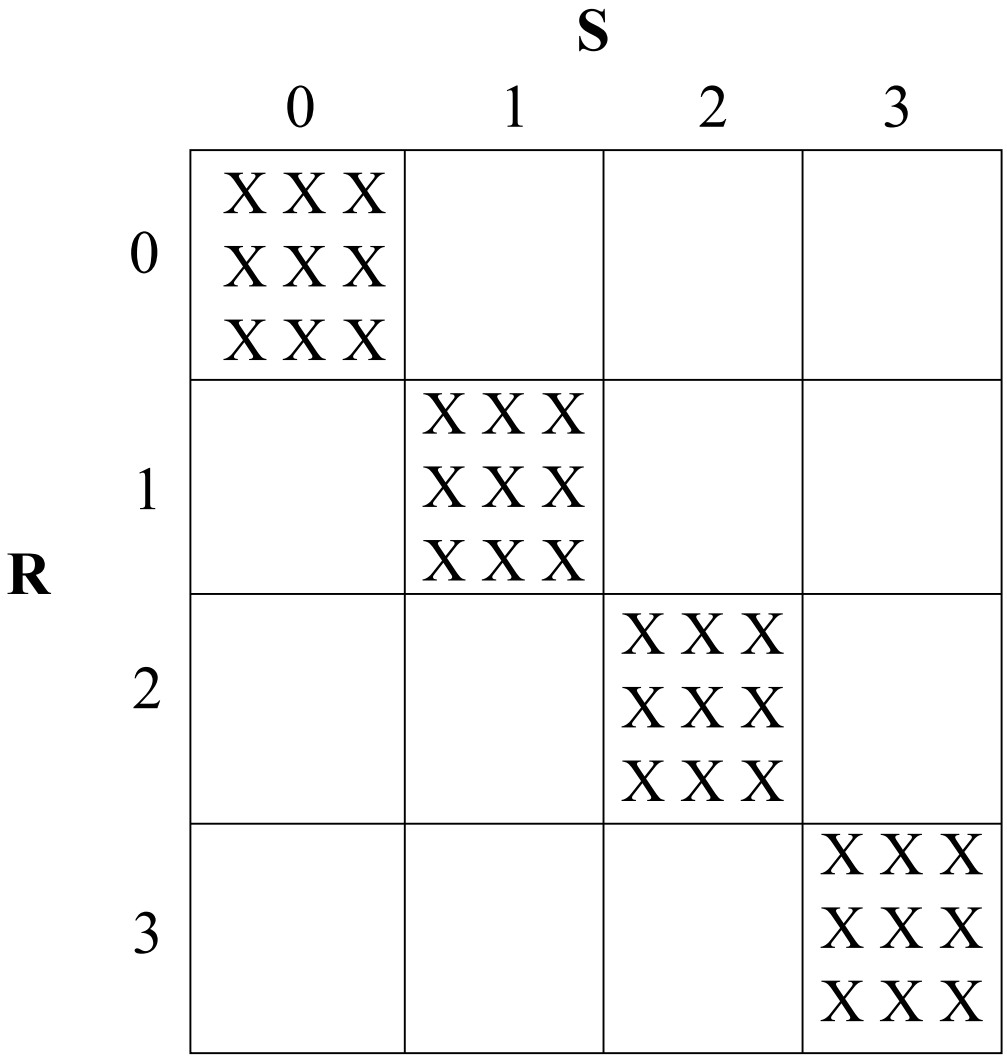
GRACE Hash-Join

bucketID = X mod 4



GRACE Hash-Join

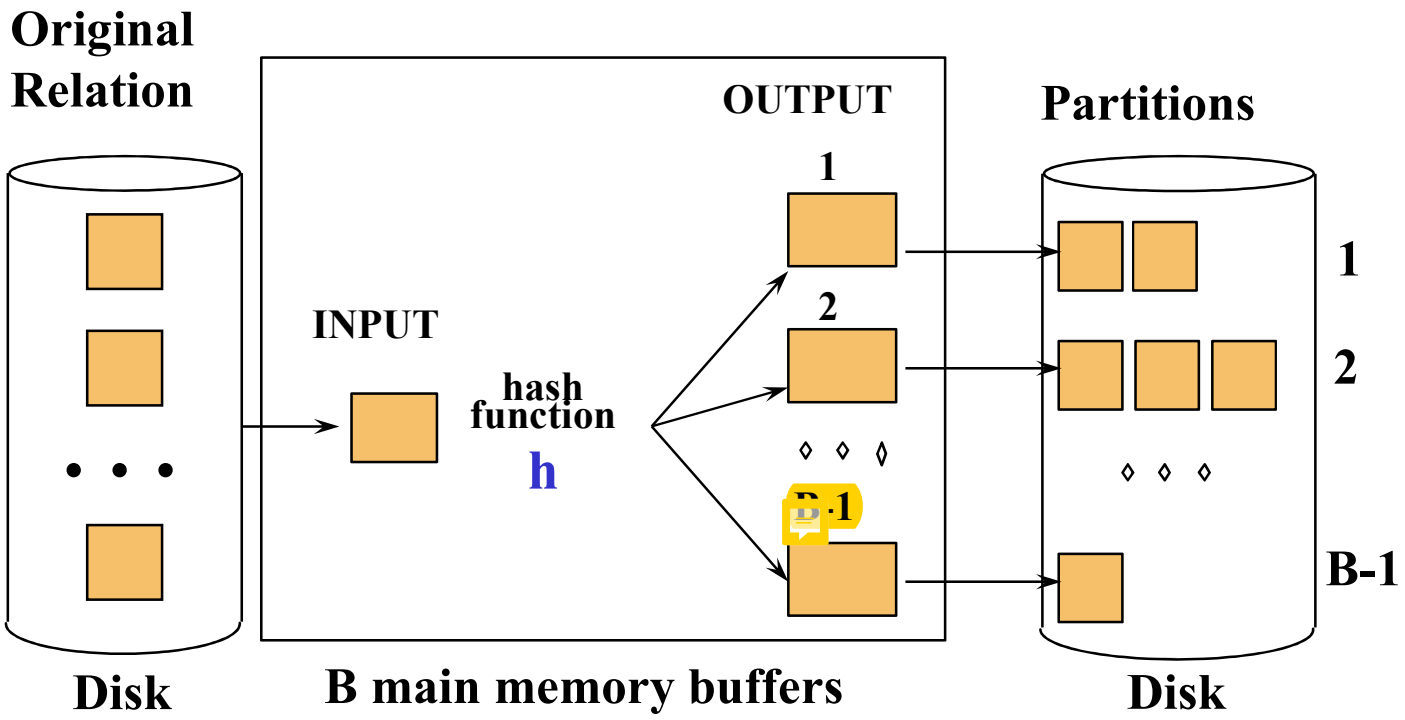
bucketID = X mod 4



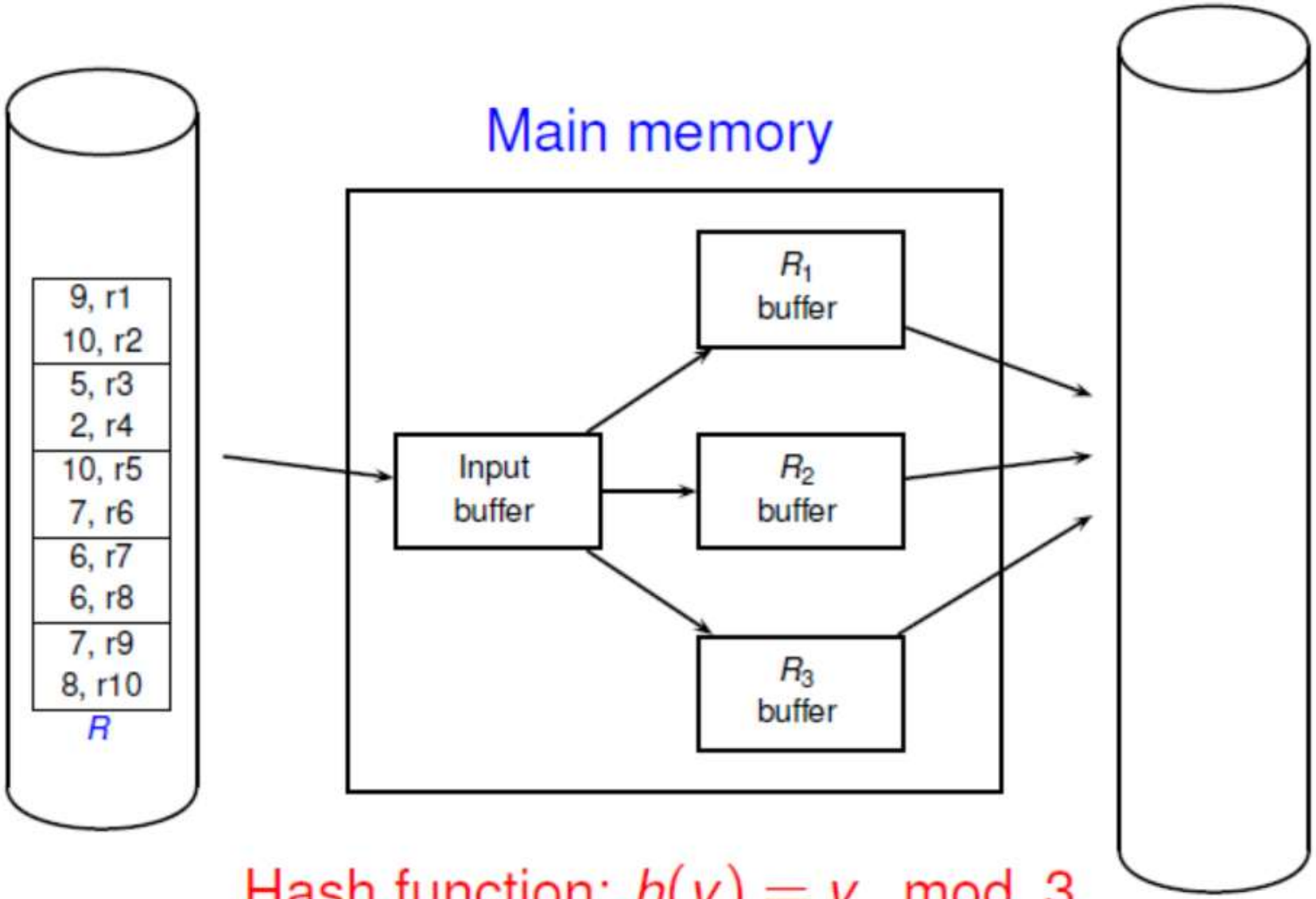
GRACE Hash-Join

- Operates in two phases:
  - Partition phase
    - Partition relation R (on join attribute) using hash fn h
    - Partition relation S using the same hash fn h
    - R tuples in partition i will only match S tuples in partition i
  - Join phase
    - Read in a partition of R
    - Build a hash table for the partition using hash fn h2 (<> h!)
    - Scan matching partition of S, search for matches
      - Using hash fn h2
- $R \bowtie S = \bigcup (R_i \bowtie S_i)$

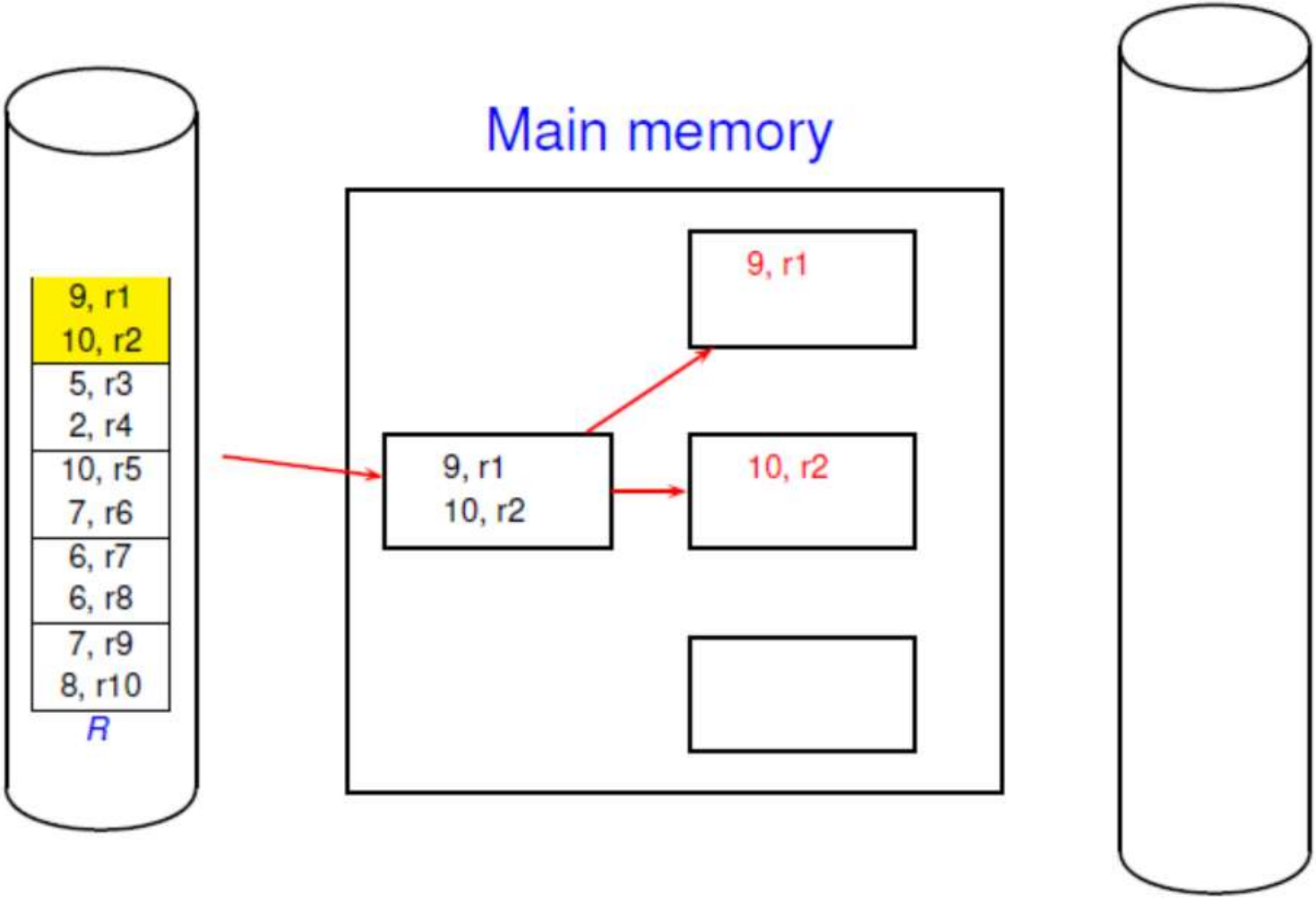
Partitioning Phase



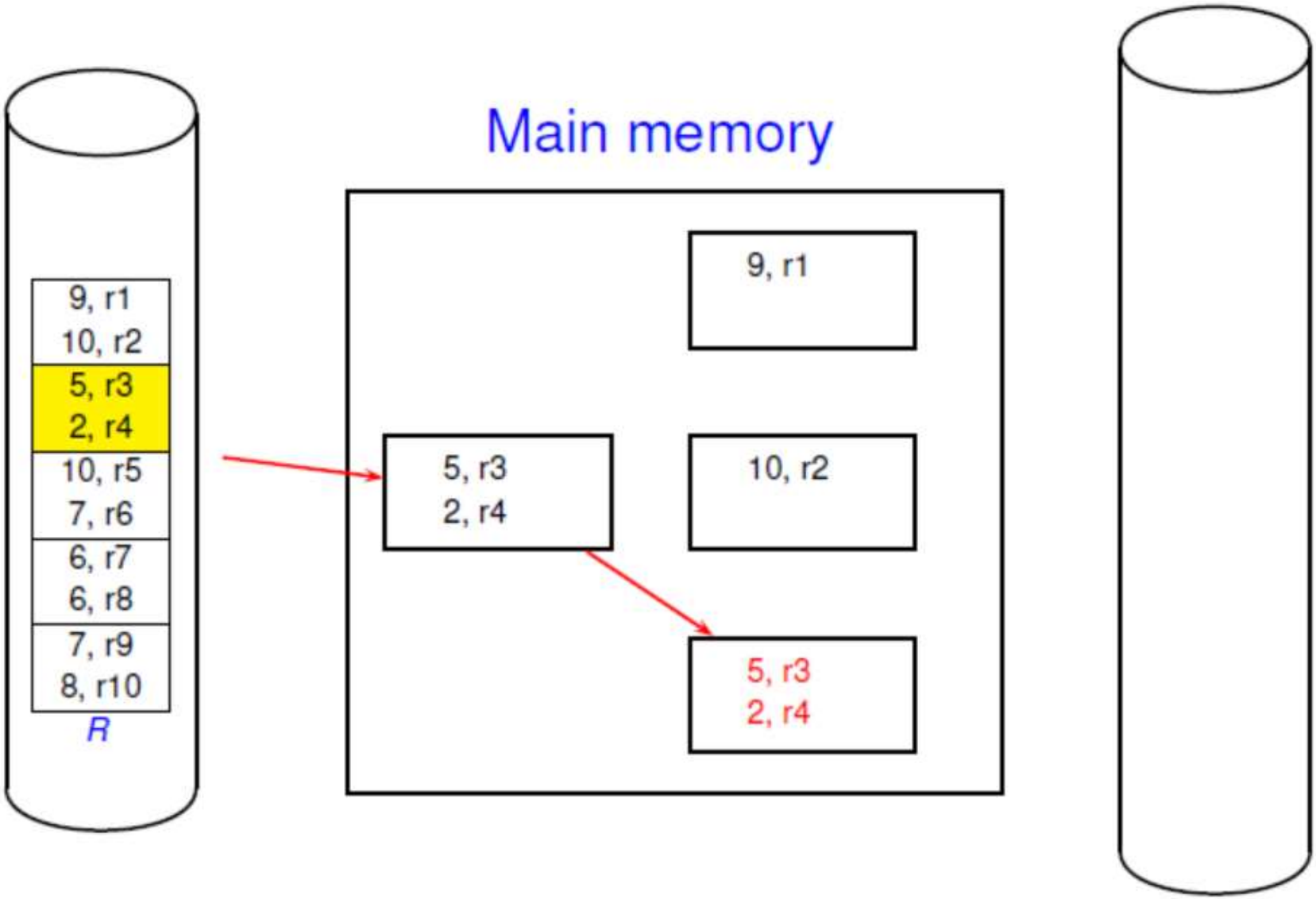
Grace Hash Join: Partitioning Relation R



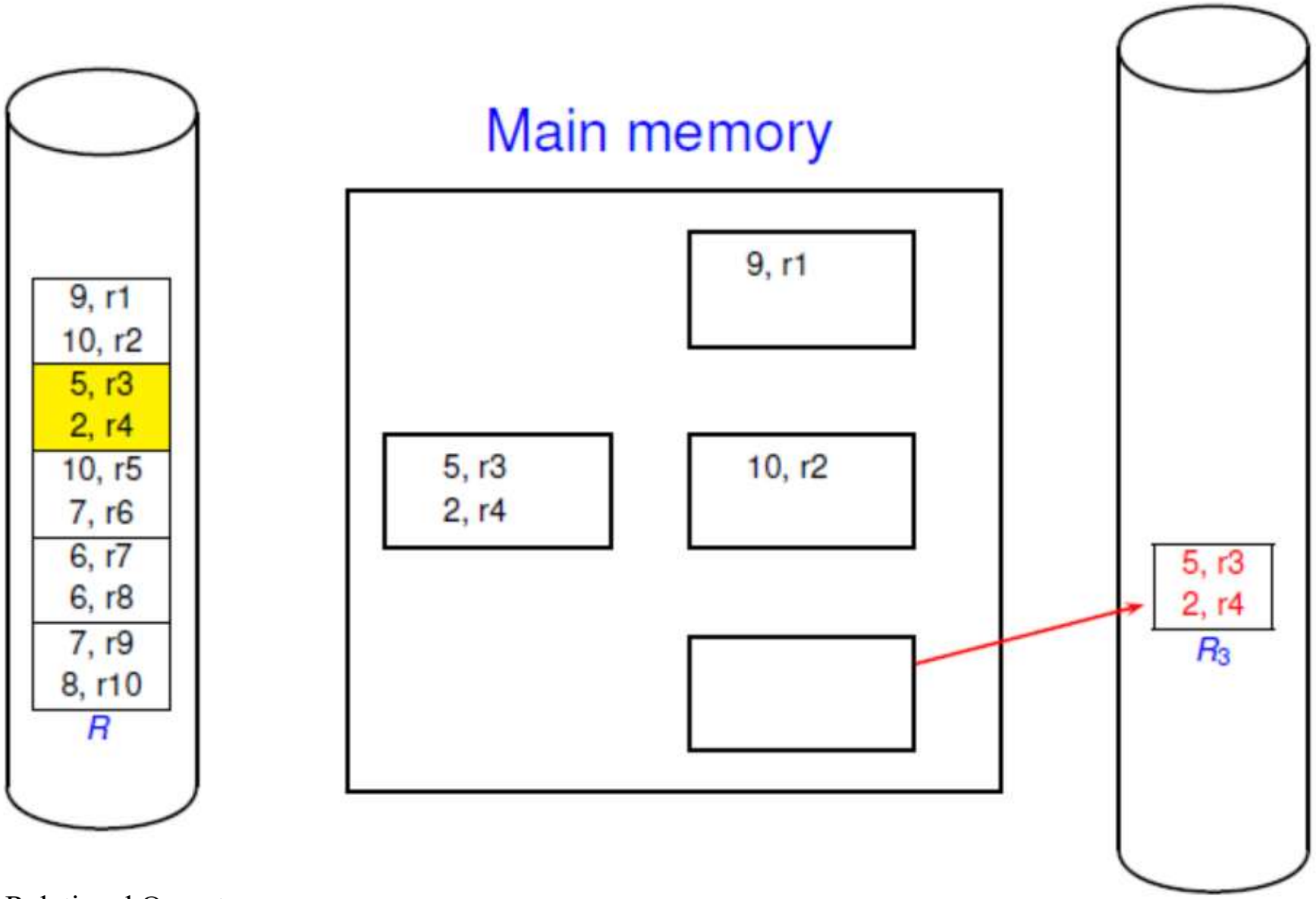
Grace Hash Join: Partitioning Relation R



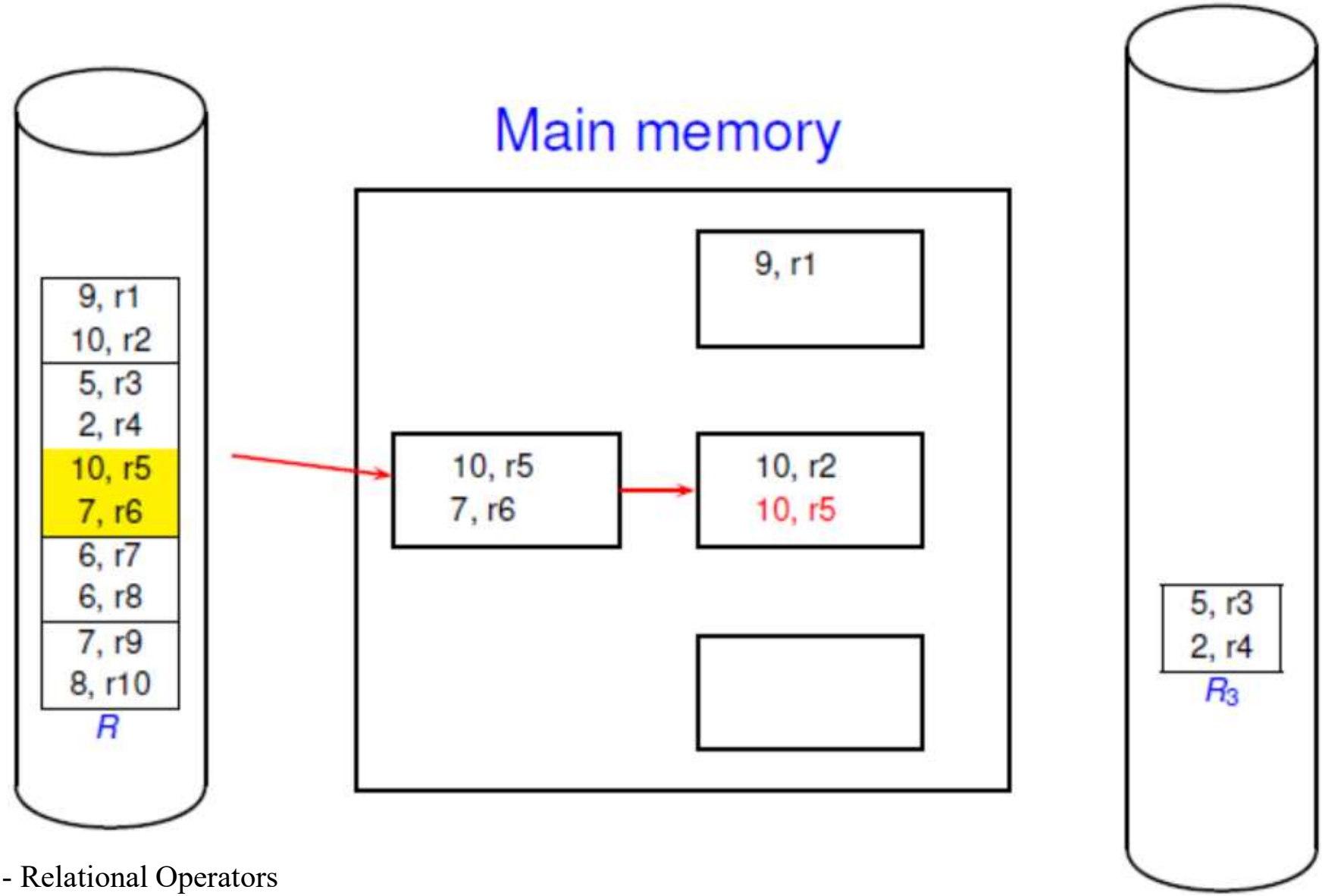
Grace Hash Join: Partitioning Relation R



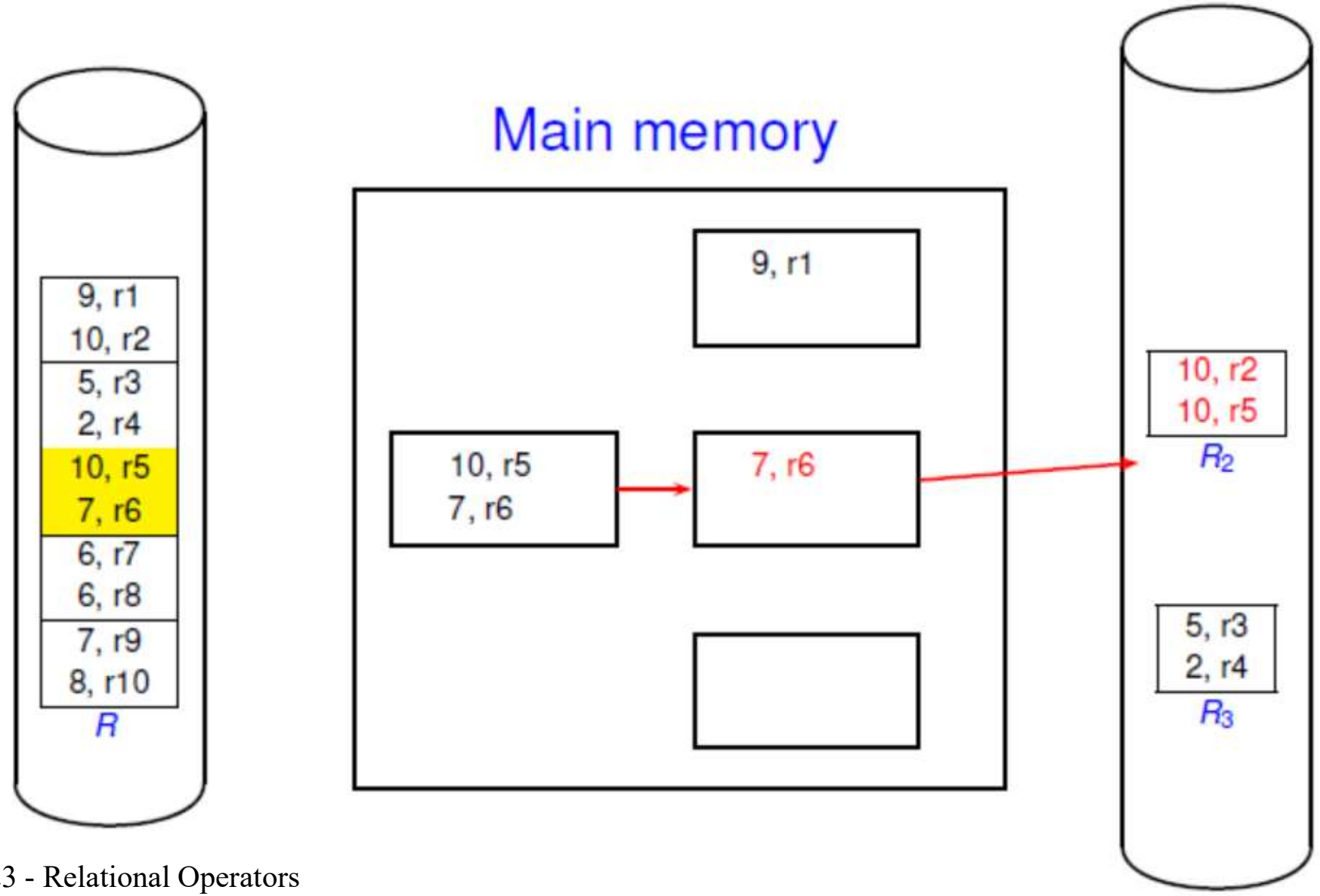
Grace Hash Join: Partitioning Relation R



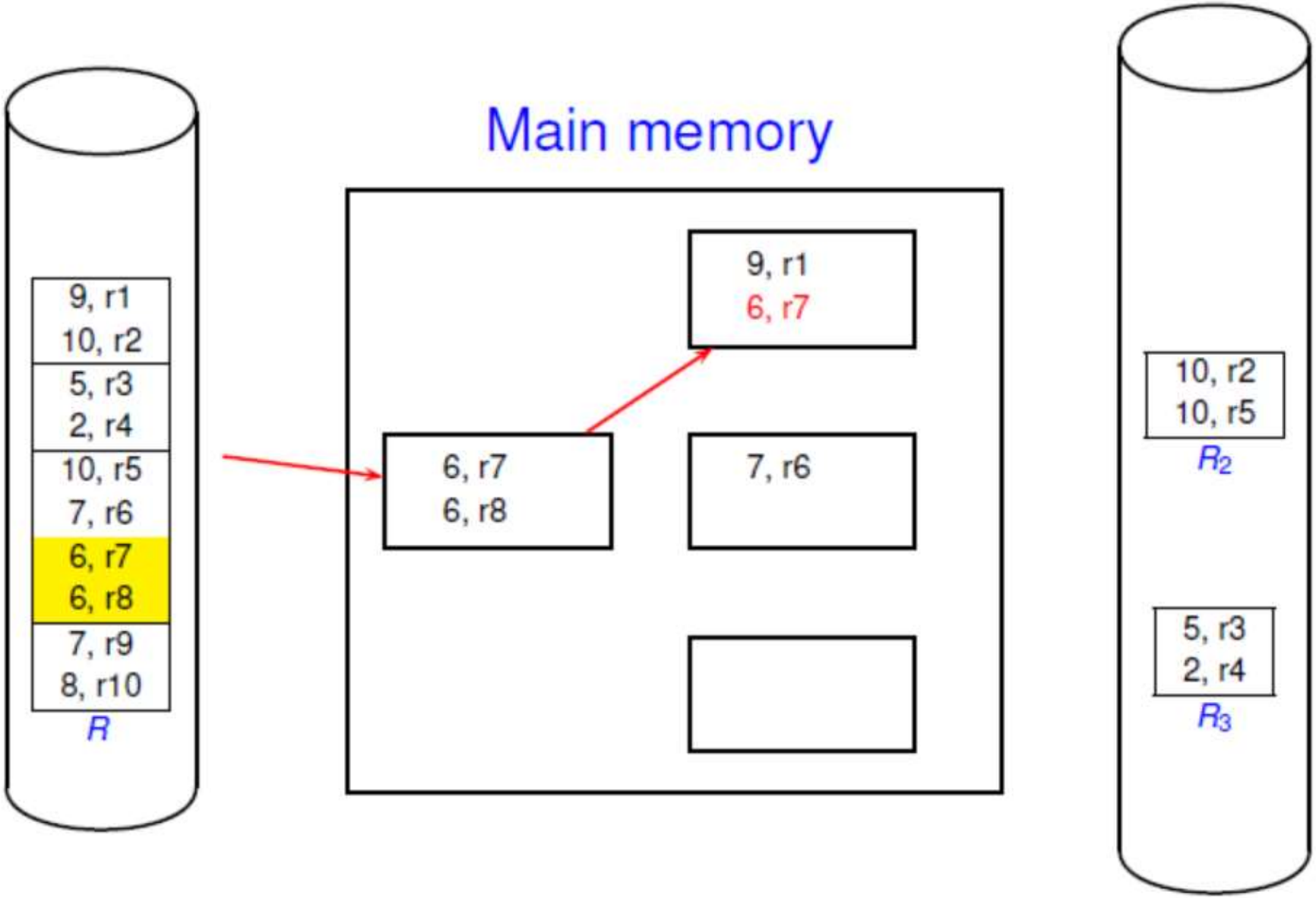
Grace Hash Join: Partitioning Relation R



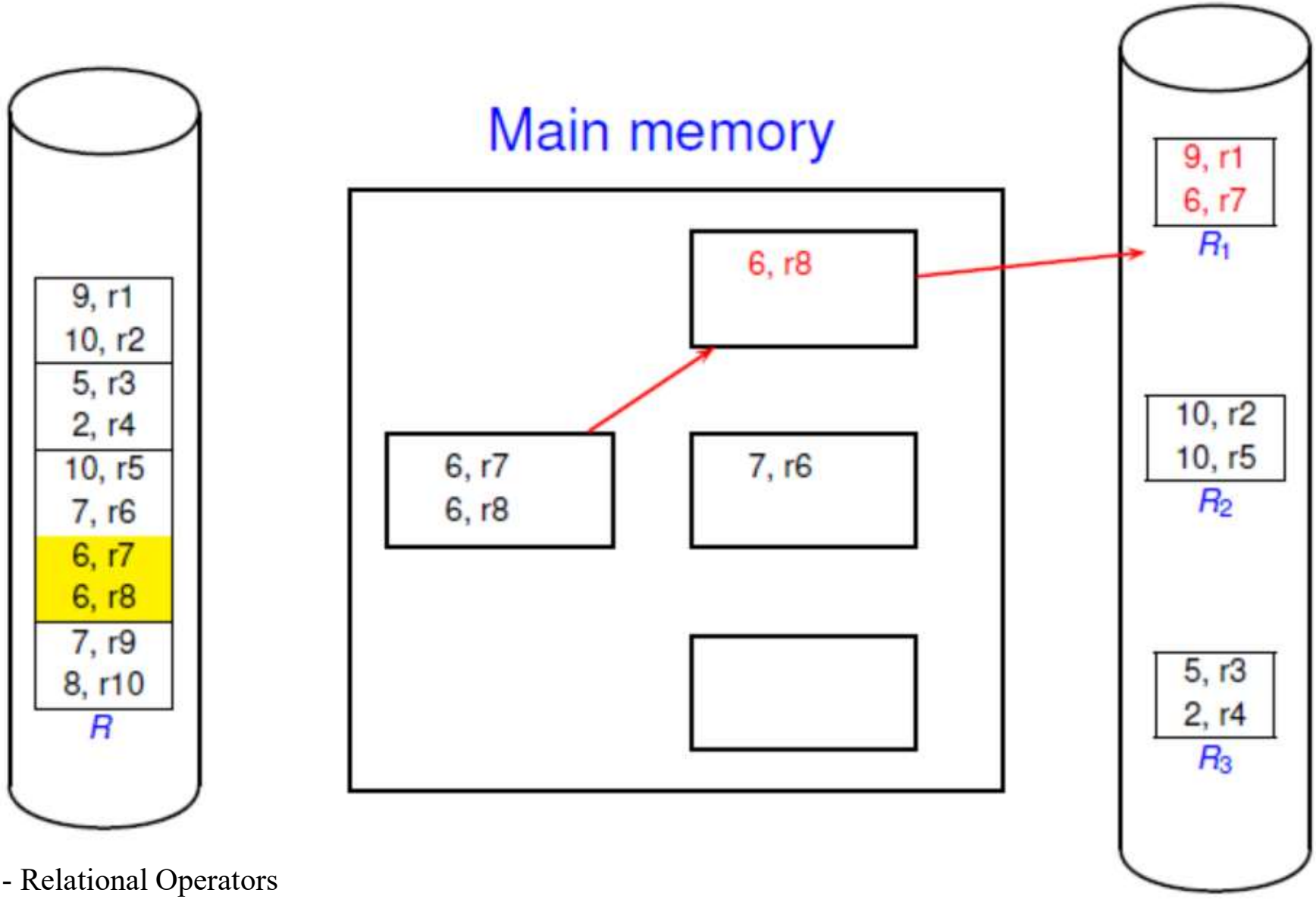
Grace Hash Join: Partitioning Relation R



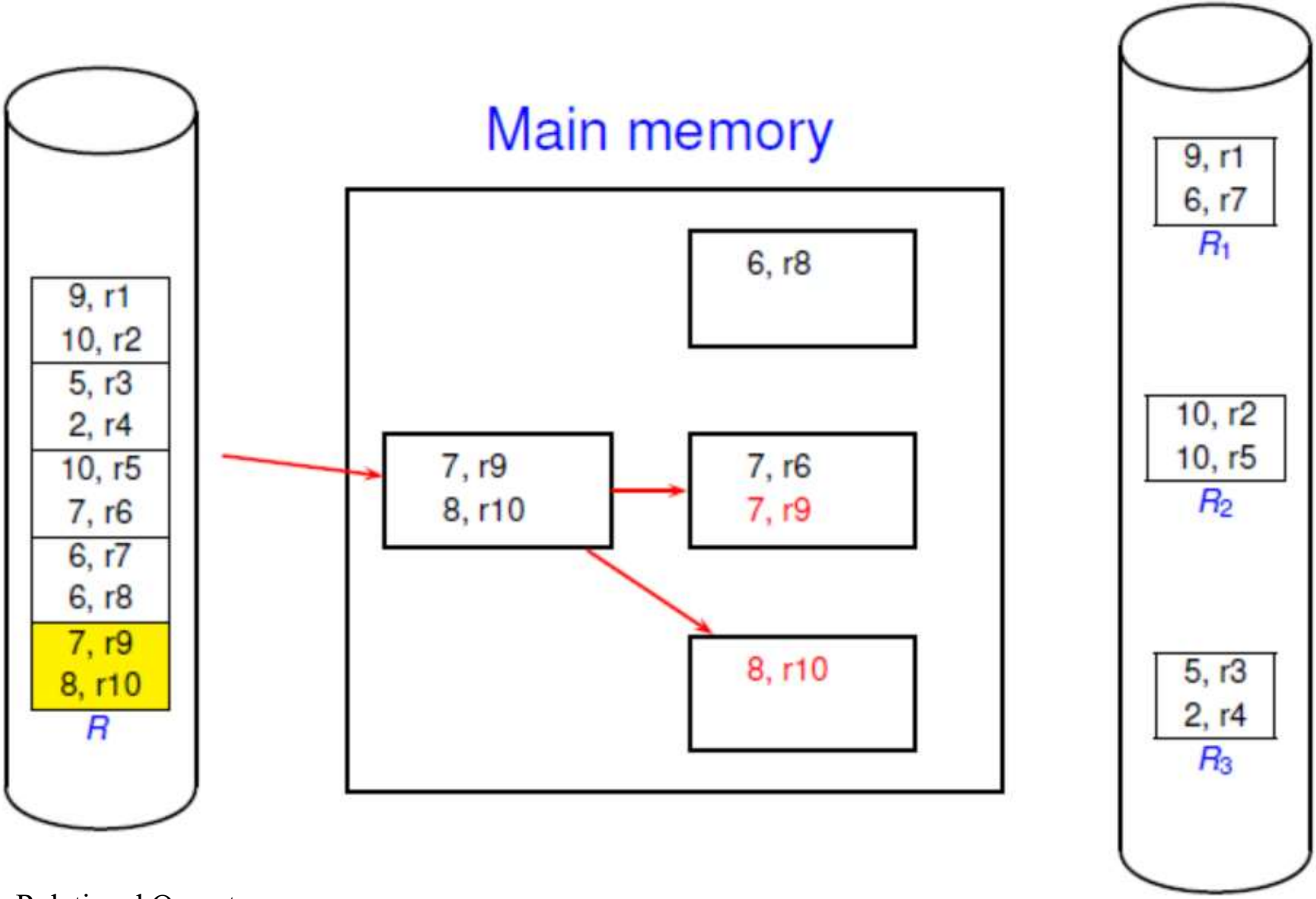
Grace Hash Join: Partitioning Relation R



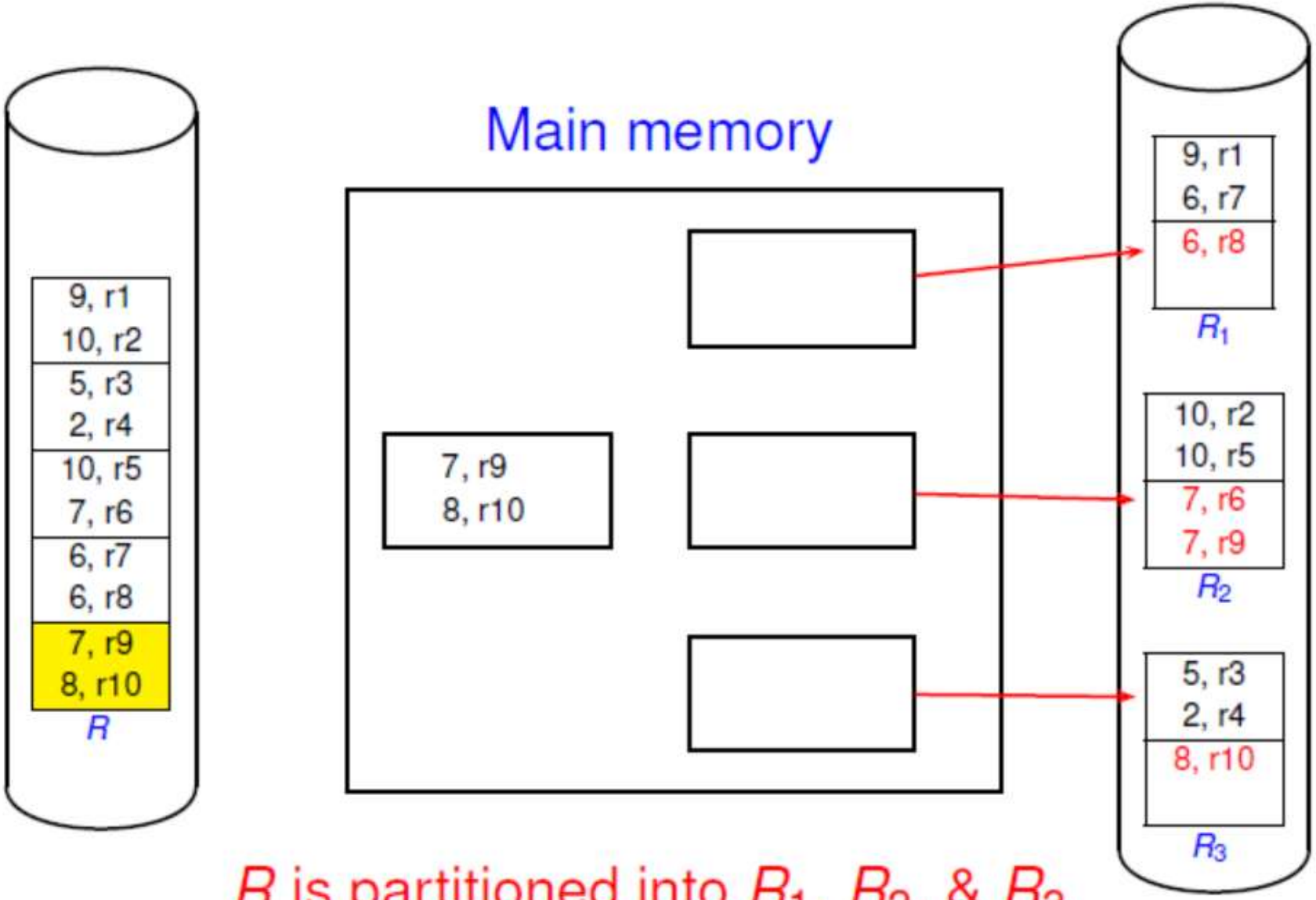
Grace Hash Join: Partitioning Relation R



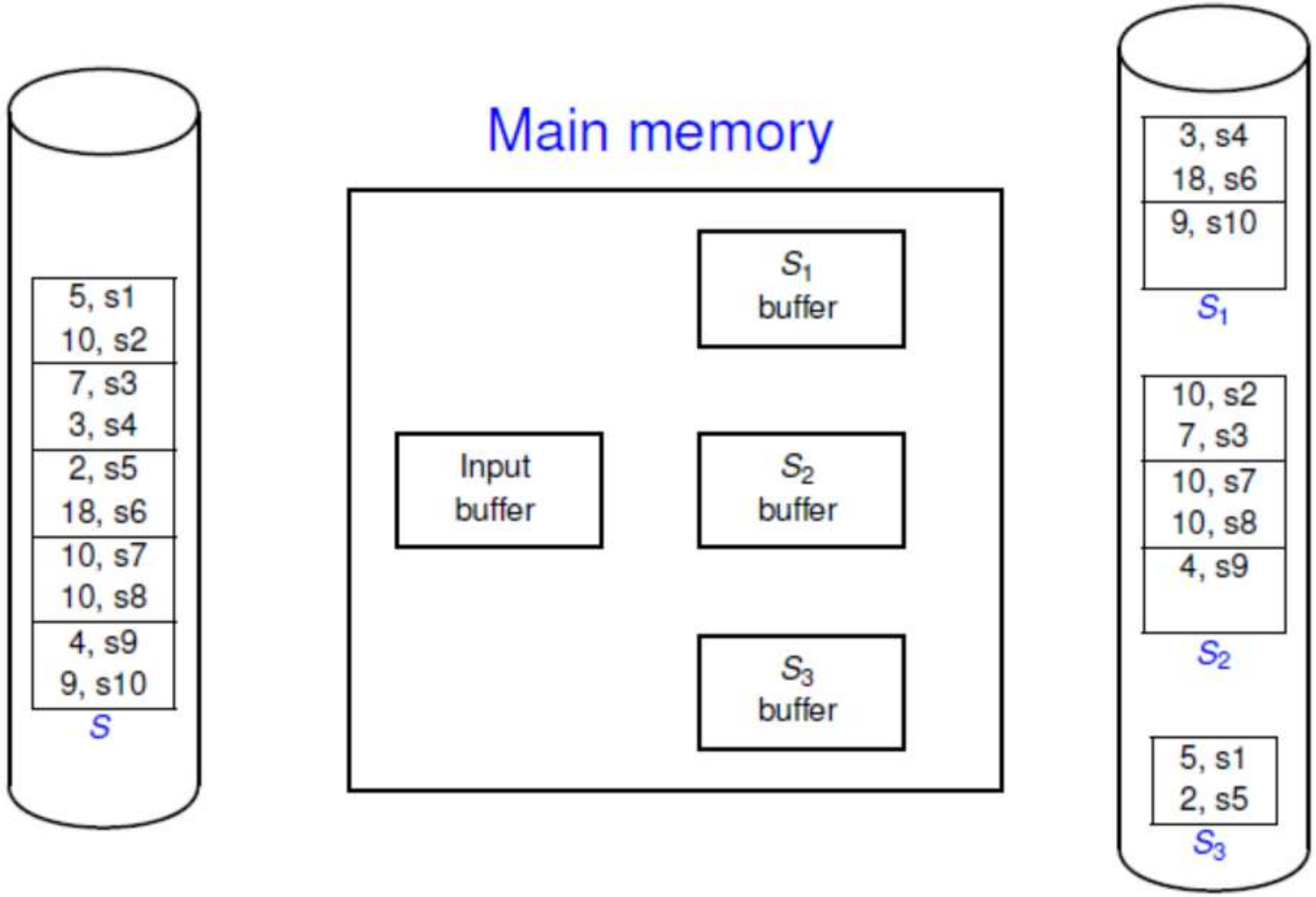
Grace Hash Join: Partitioning Relation R



Grace Hash Join: Partitioning Relation R

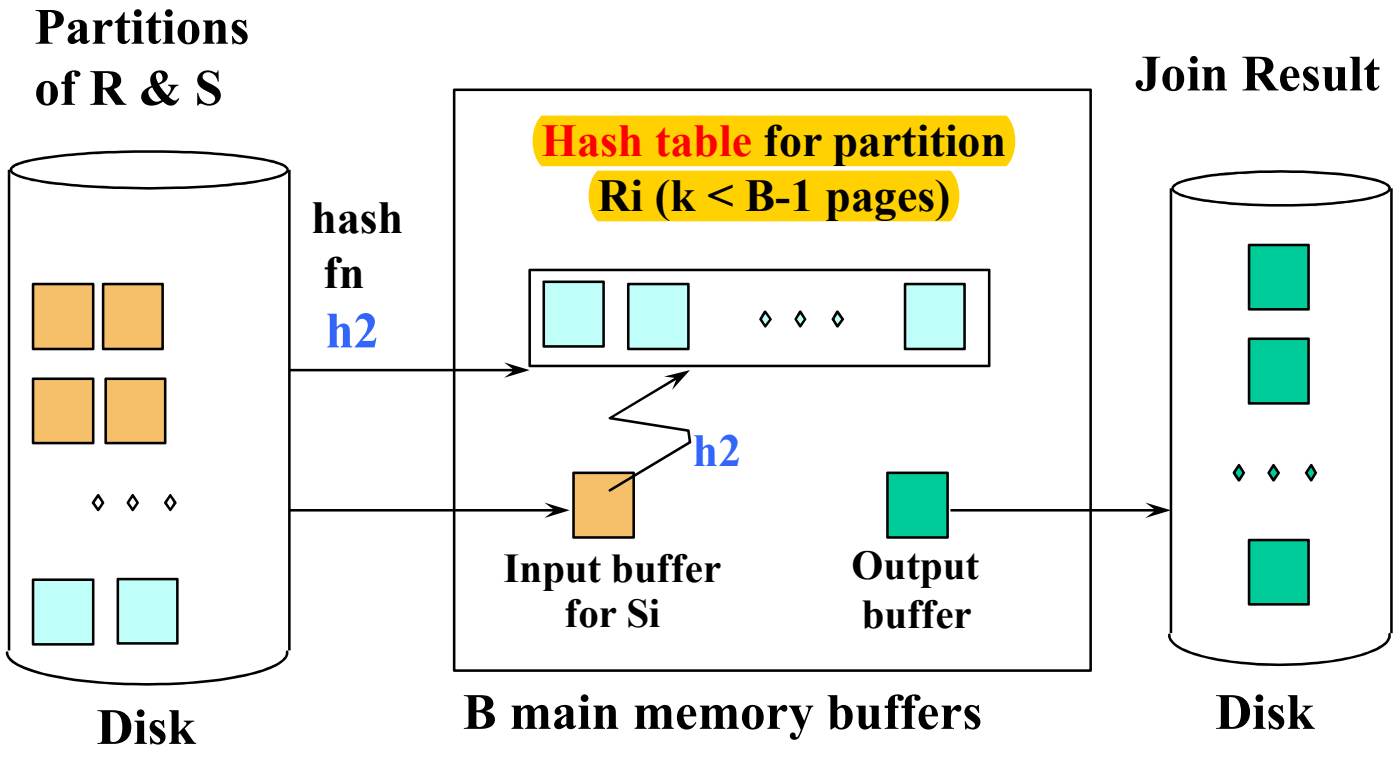


Grace Hash Join: Partitioning Relation S

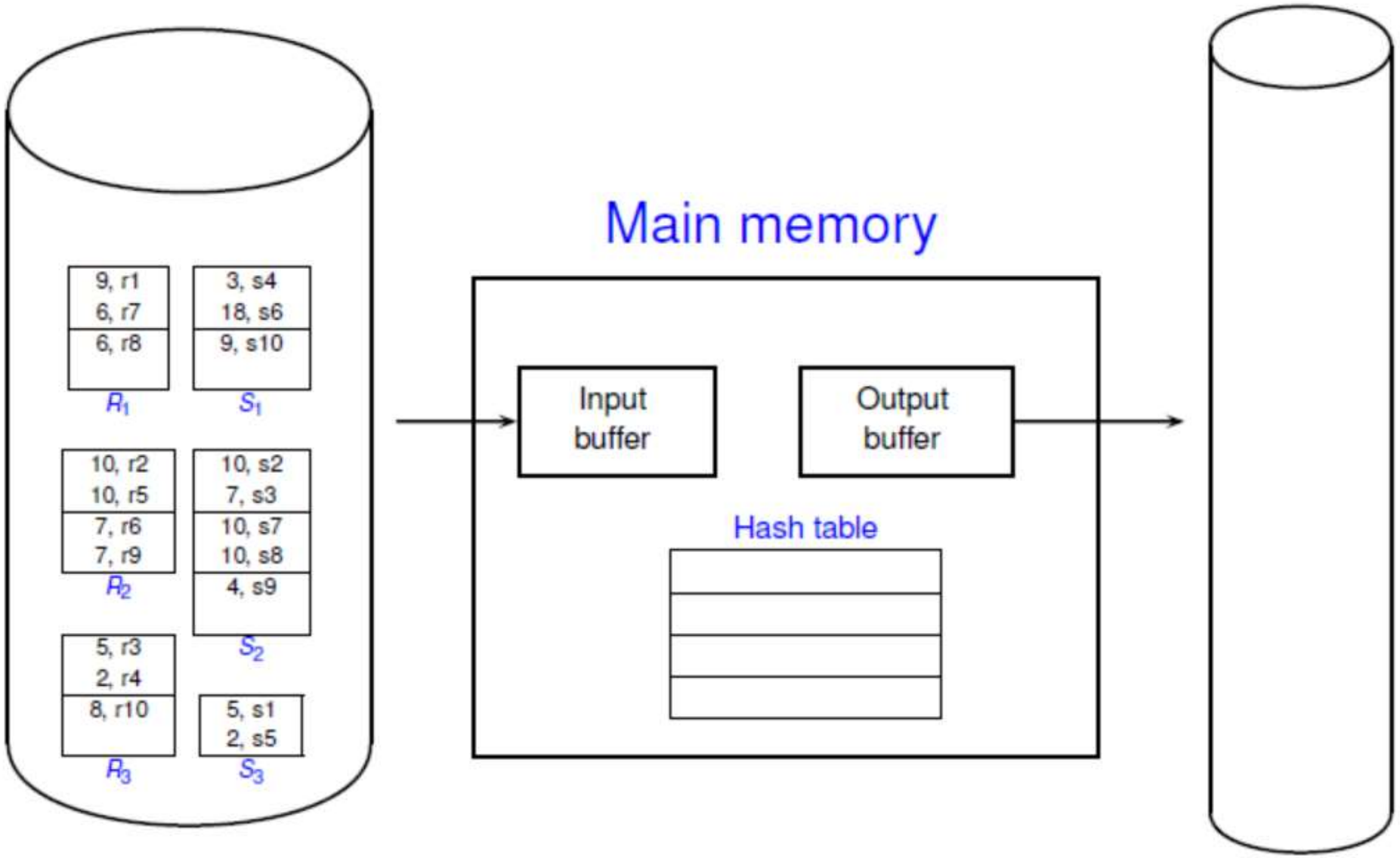


Similarly, S is partitioned into S_1, S_2, & S_3

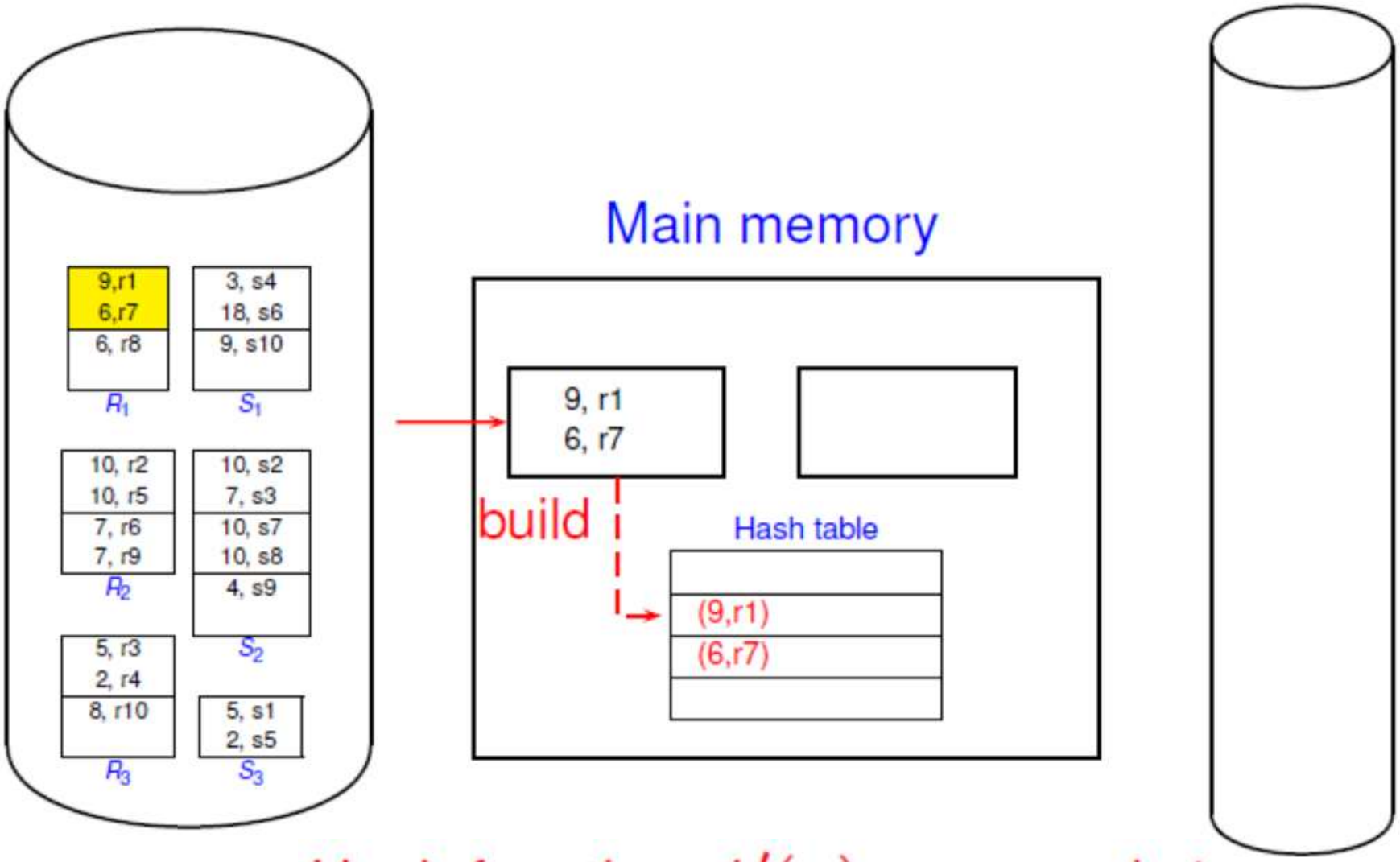
Joining Phase



Grace Hash Join: Probing/Joining Phase

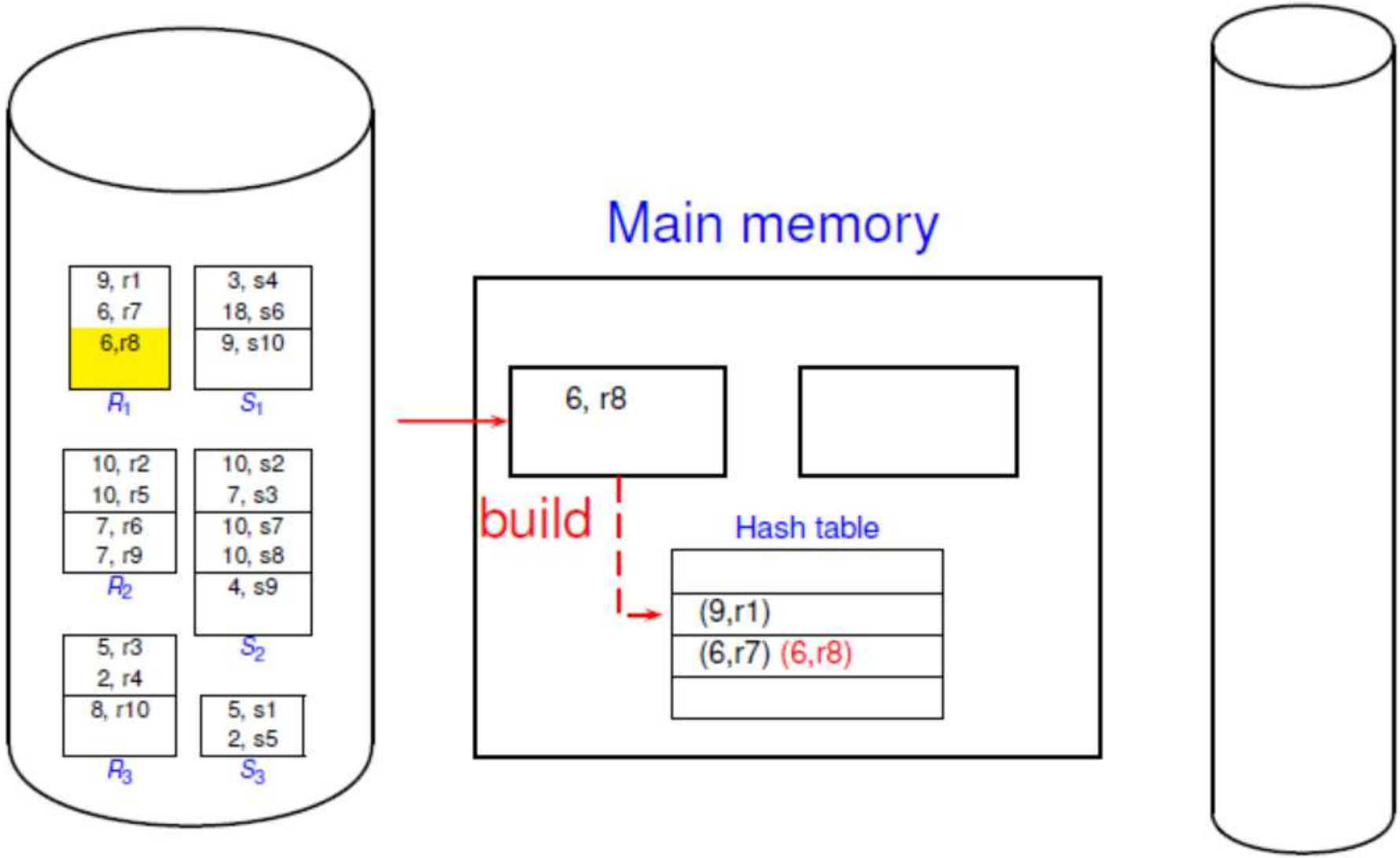


Grace Hash Join: Probing/Joining Phase

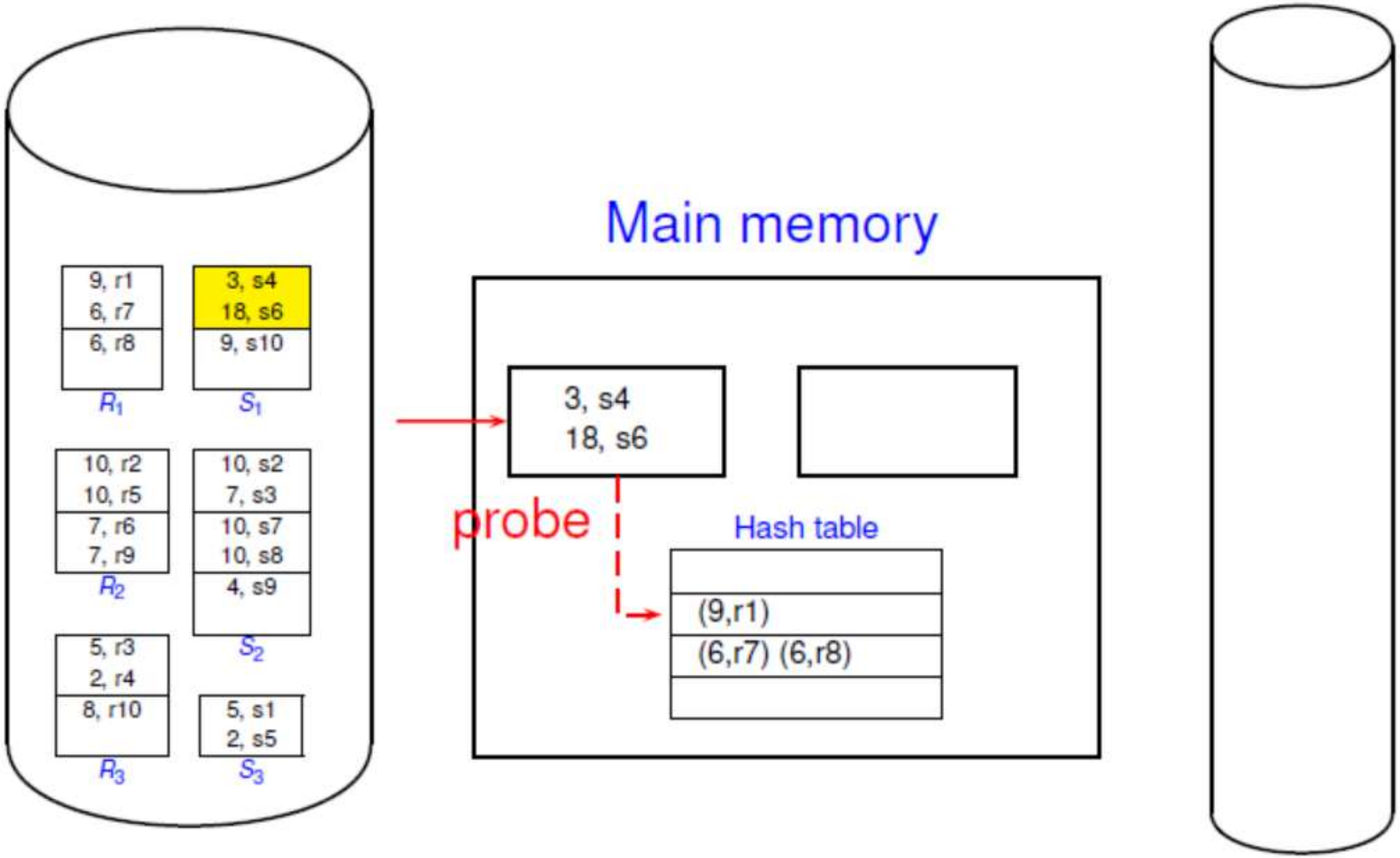


Hash function: $h'(v) = v \mod 4$

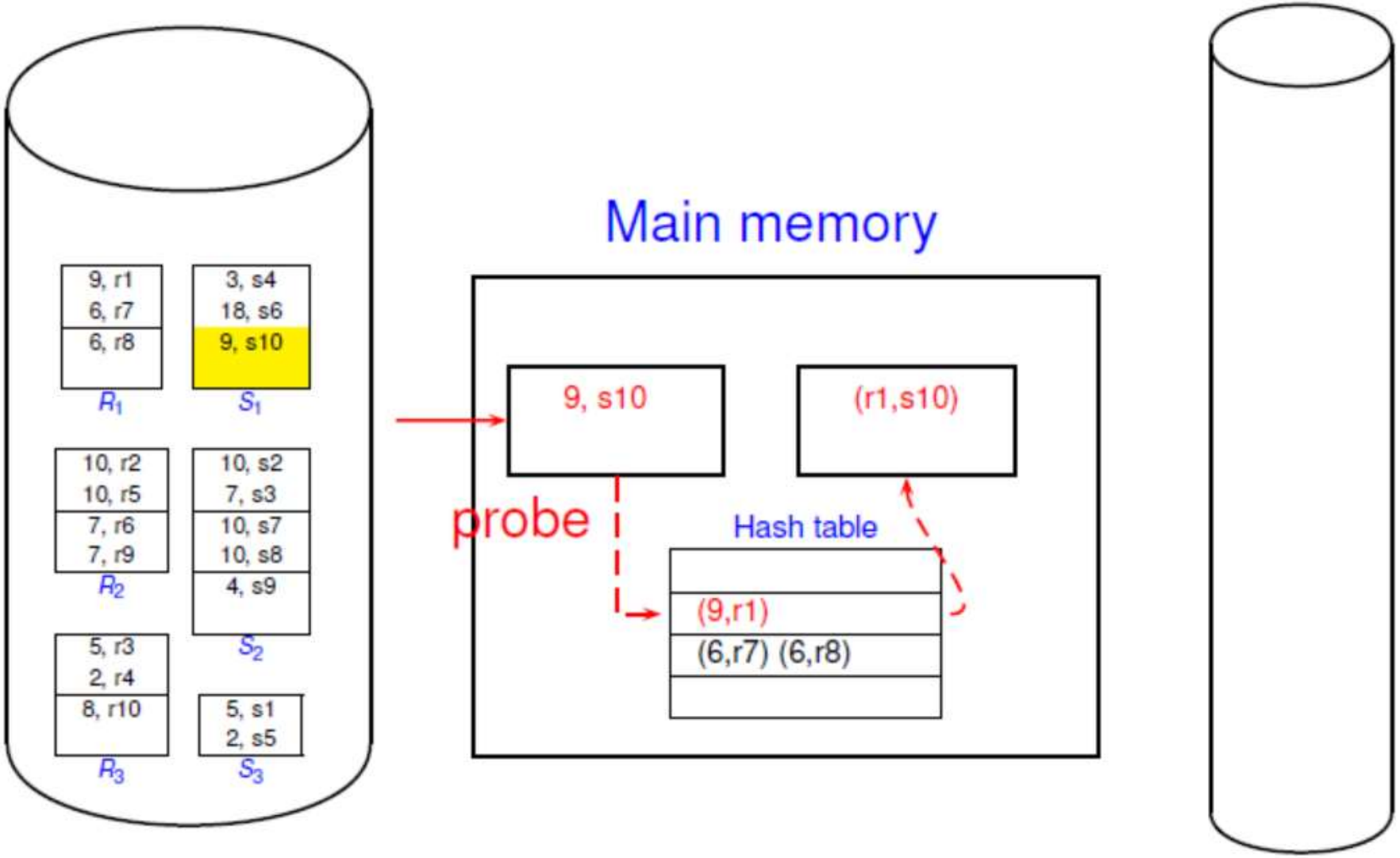
Grace Hash Join: Probing/Joining Phase



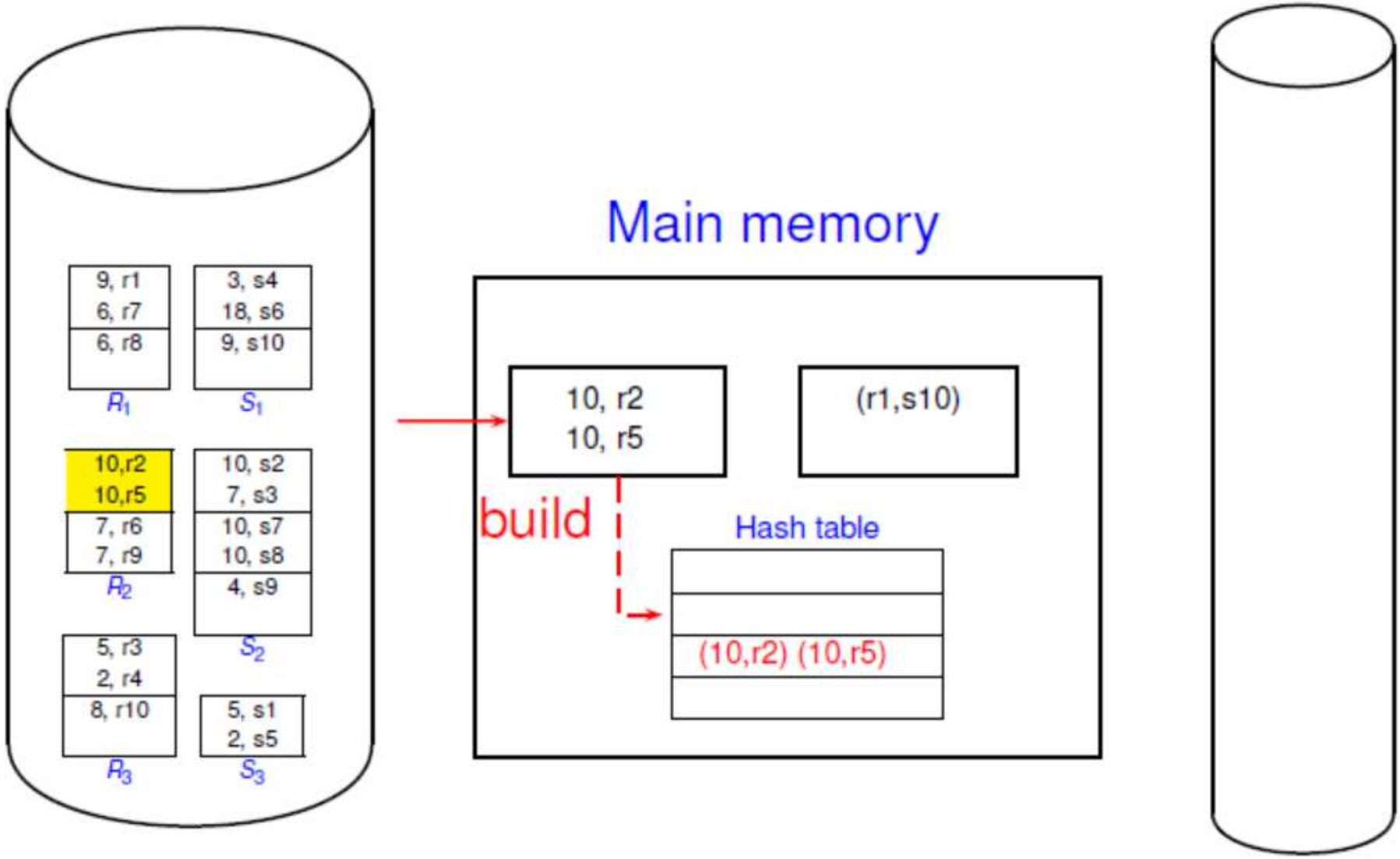
Grace Hash Join: Probing/Joining Phase



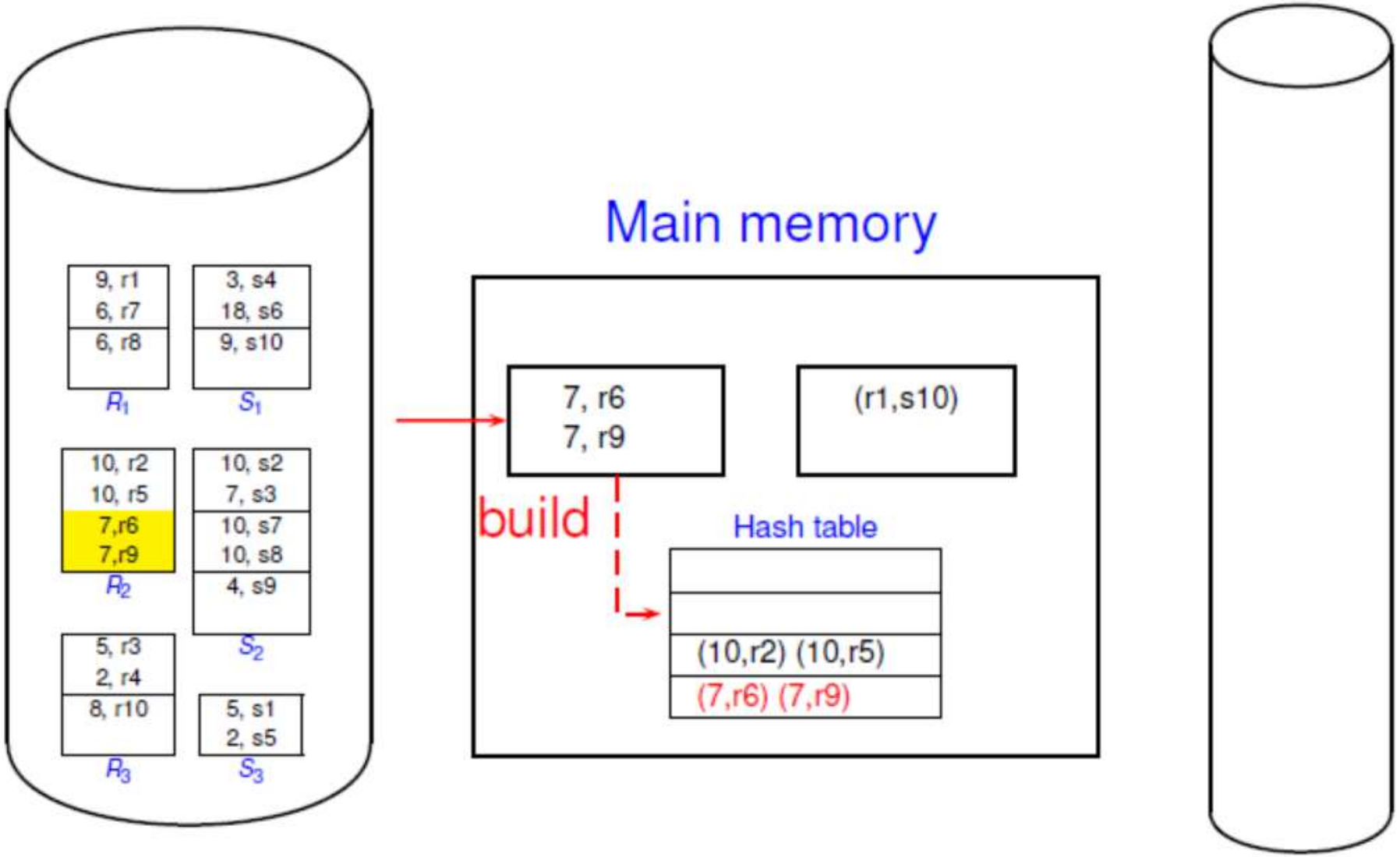
Grace Hash Join: Probing/Joining Phase



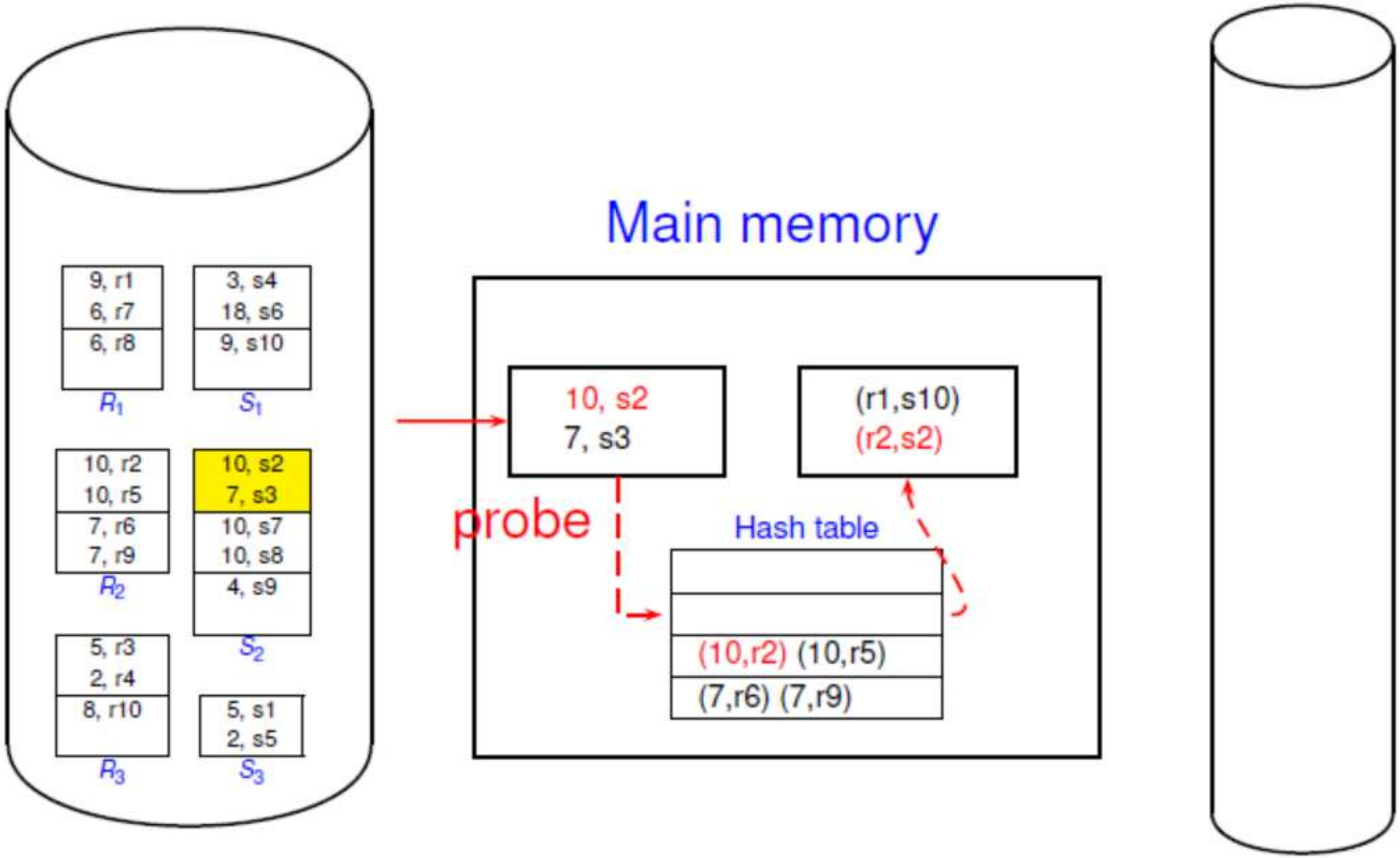
Grace Hash Join: Probing/Joining Phase



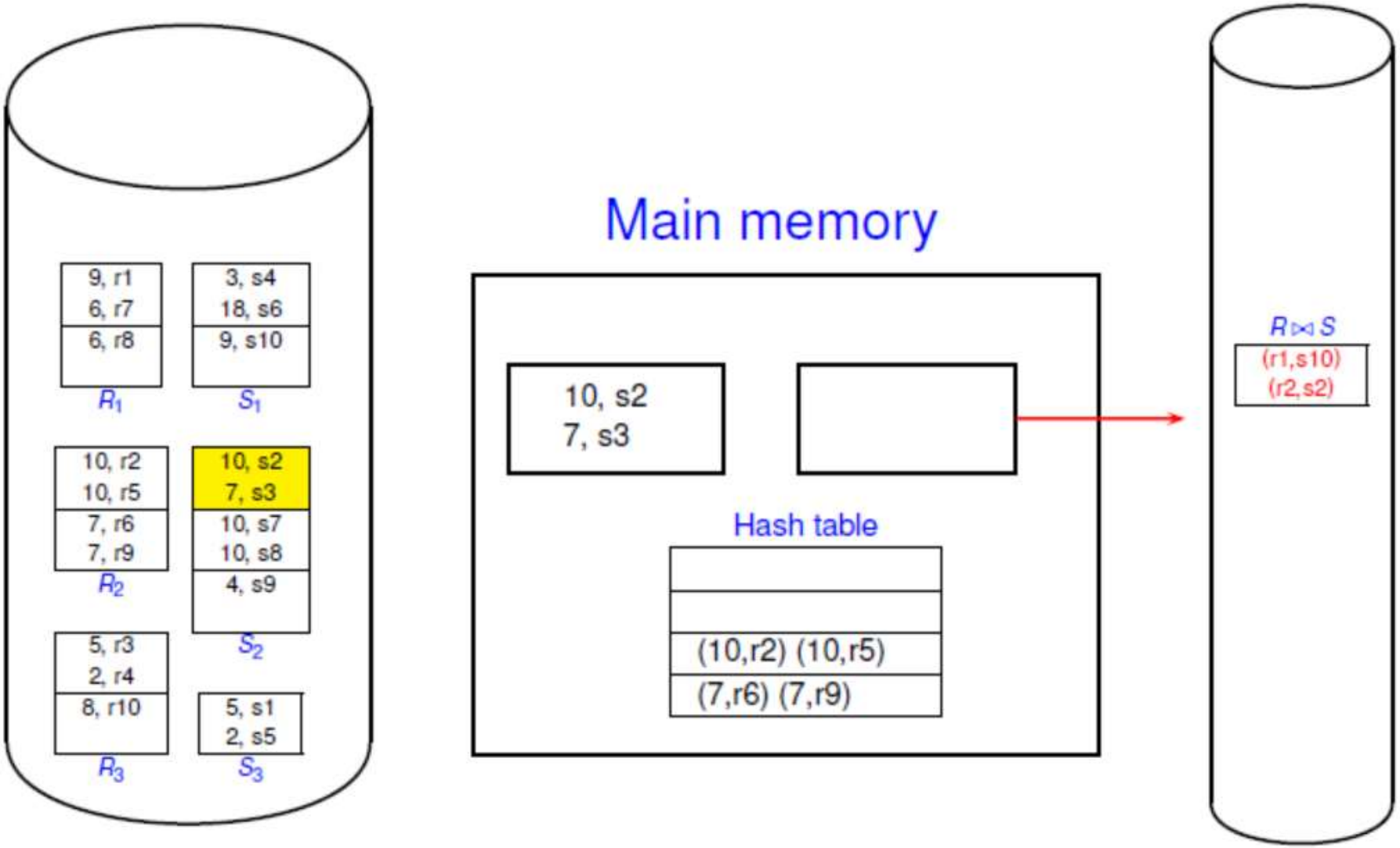
Grace Hash Join: Probing/Joining Phase



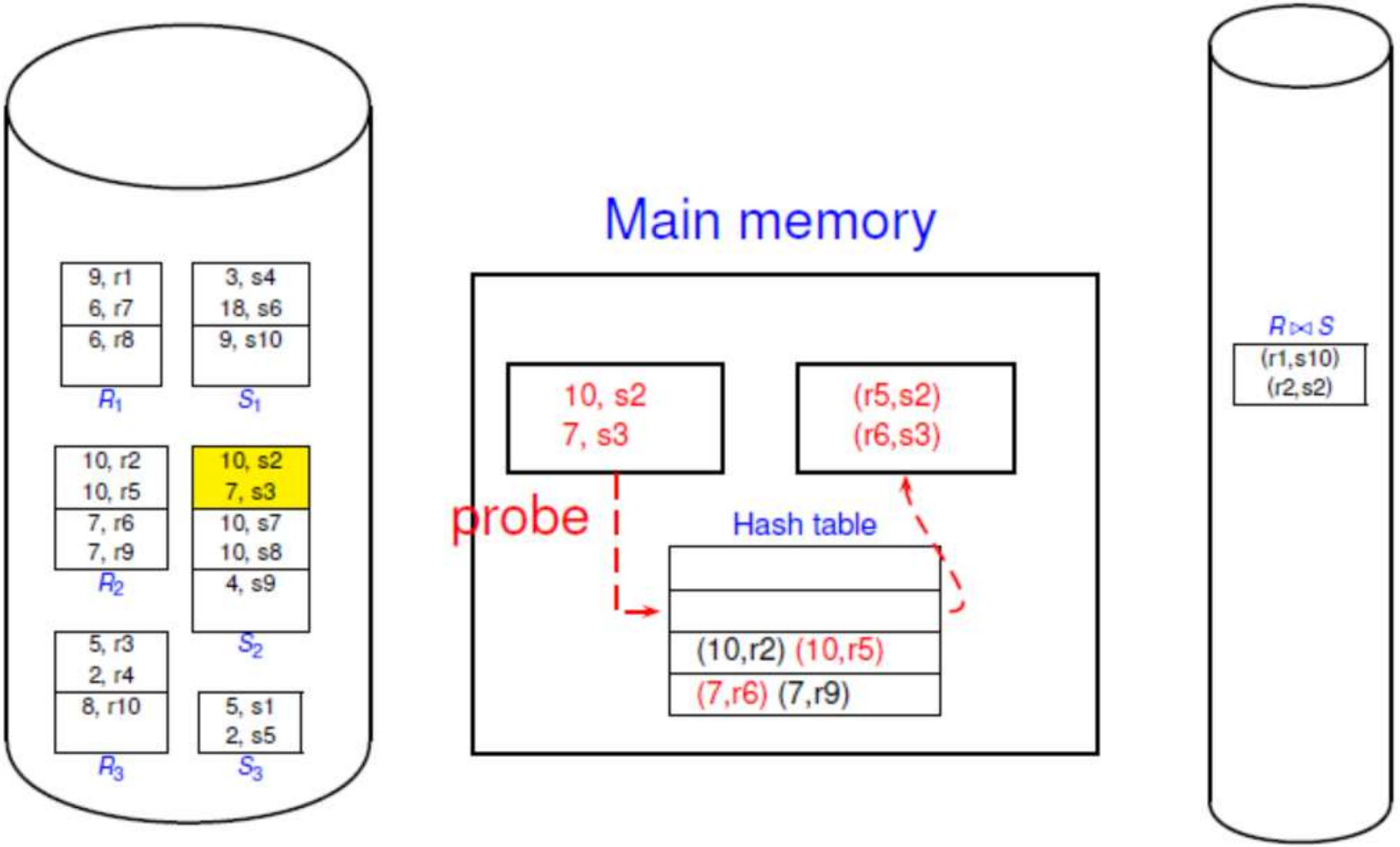
Grace Hash Join: Probing/Joining Phase



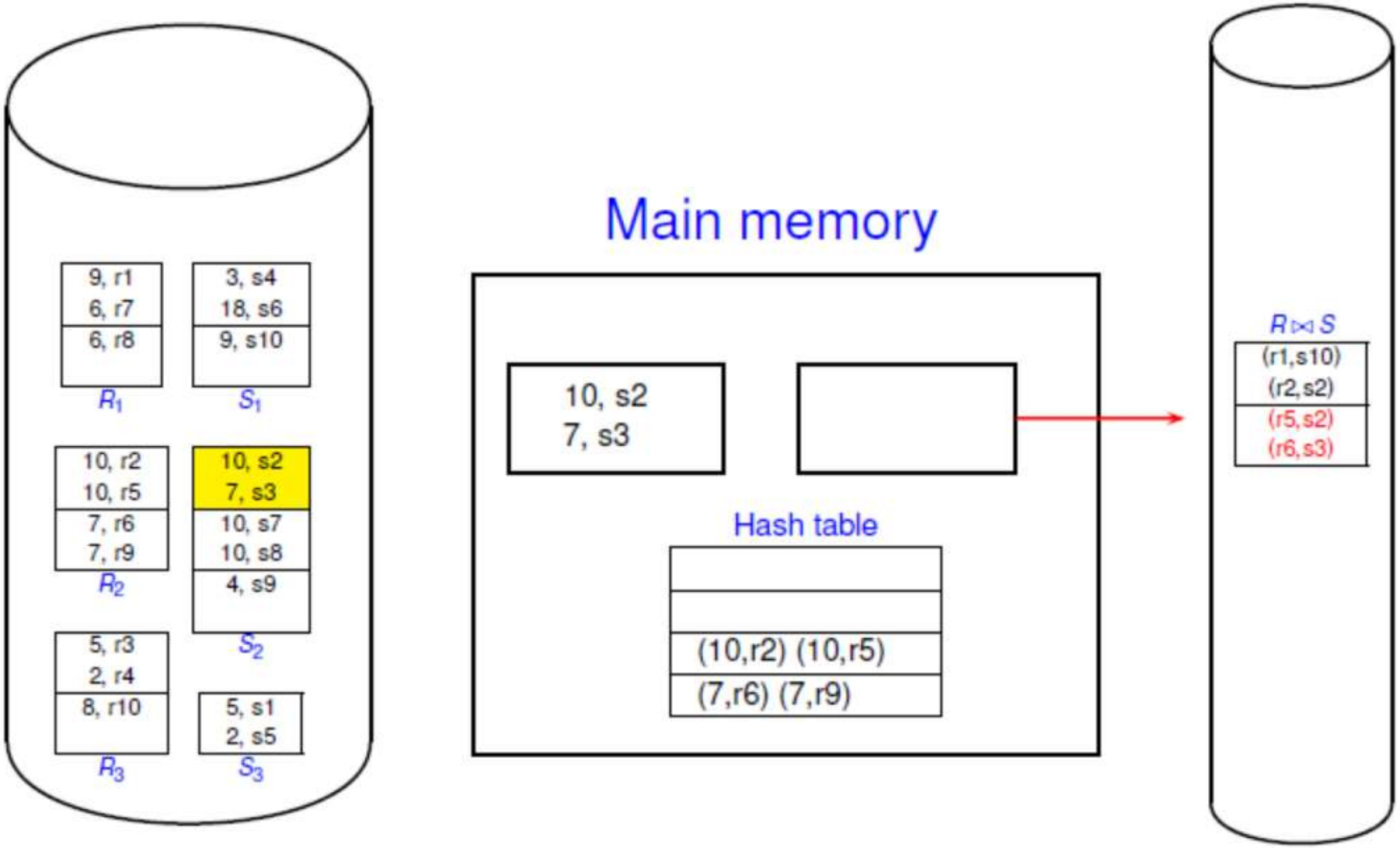
Grace Hash Join: Probing/Joining Phase



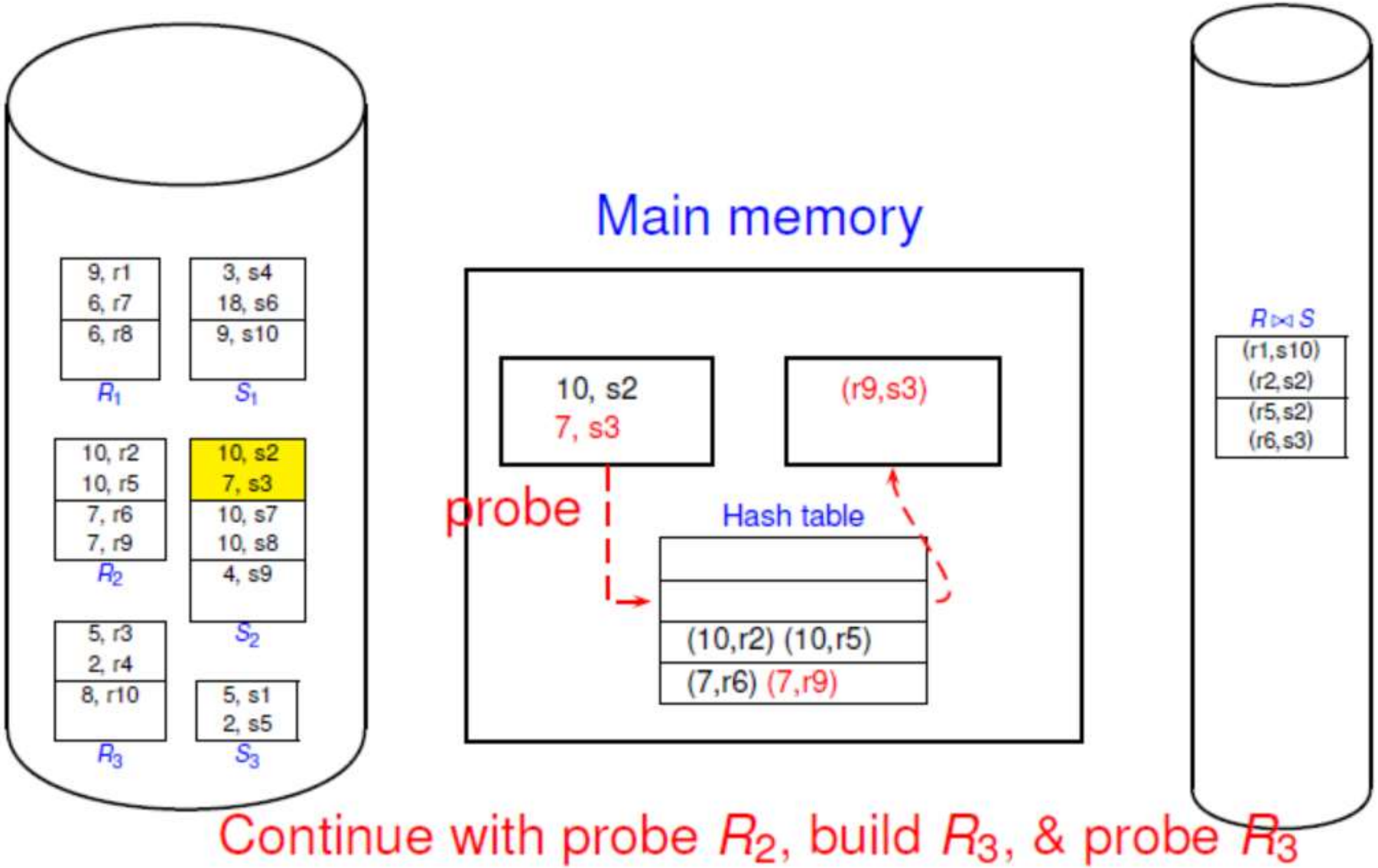
Grace Hash Join: Probing/Joining Phase



Grace Hash Join: Probing/Joining Phase



Grace Hash Join: Probing/Joining Phase



Cost of Hash-Join

- In partitioning phase, read+write both relns
  - 2(M+N)
- In matching phase, read both relns
  - M+N I/Os

Thus the total cost is 3M + 3N

- In our running example, this is a total of 4500 I/Os

BNL cost: 2500 to 15000 I/Os
SortMerge cost: 7500 I/Os

Observations on Hash-Join

- #partitions $k \leq$ B-1 (why?)
- B-2 $\geq$ size of *largest* partition to be held in memory
- Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) $\leq$ B-2, i.e., B must be $\geq \sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. How?
  - Apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition
  - Can also apply any other join algorithms on that pair of partitions
- What if B < $\sqrt{M}$?

$R_1 \bowtie S_1 \cup R_2 \bowtie S_2 \cup R_{21} \bowtie S_{21} \cup R_{22} \bowtie S_{22} \cup R_3 \bowtie S_{23}$

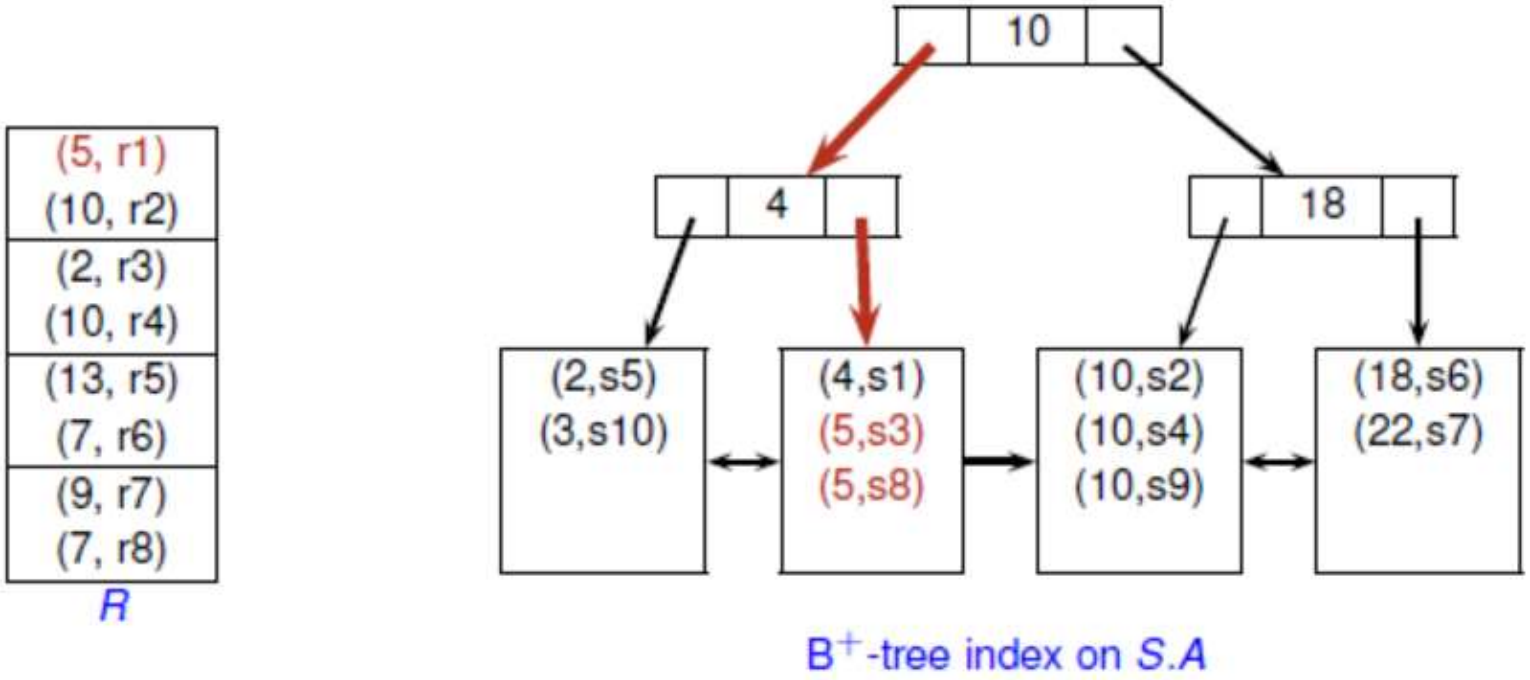
Index Nested Loops Join

foreach tuple r in R do
search index of S on sid using $S_{search\text{-}key}$ = r.sid
for each matching key
retrieve s; add (r, s) to result

- Precondition: there is an index on the join column of one relation, say S; make S the inner relation; and exploit the index

Index Nested Loops Join

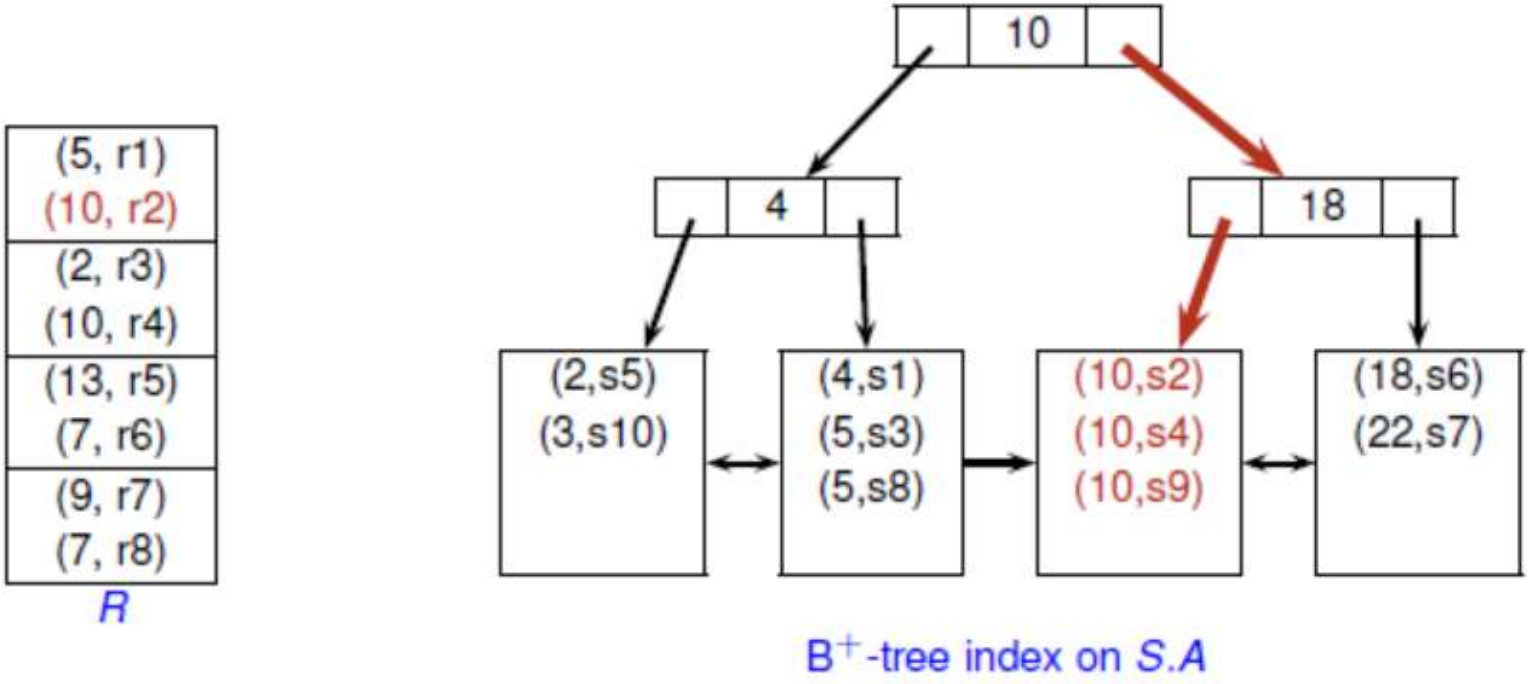
- Consider R(A, B) $\bowtie_A$ S(A, C)
- Assume that there is a B^+-tree index on S.A



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

Index Nested Loops Join

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



Next, join (10, r2) ∈ R with matching tuples in S, and so on …

Index Nested Loops Join

foreach tuple r in R do
    search index of S on sid using $S_{search\text{-}key}$ = r.sid
    for each matching key
        retrieve s; add (r, s) to result

- Precondition: there is an index on the join column of one relation, say S; make S the inner relation; and exploit the index
  - Cost: M + (M*p_R) * cost of finding matching S tuples
- For each R tuple, cost of probing S index (assuming format 2 – (Key, Rid)-pair)
  - **B^+ tree**. H (traverse the index = height of tree) + cost of finding S tuples
    - The second component depends on clustering
      - Clustered index: 1 I/O (typical)
      - Unnclustered: upto 1 I/O per matching S tuple
  - **Hash index**. 1.2 (magic number) + cost of finding S tuples

Examples of Index Nested Loops

- Hash-index on sid of S (as inner):
  - Scan R: 1000 page I/Os, 100*1000 tuples
  - For each R tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching S tuple. Total: 220,000 I/Os
- Hash-index on sid of R (as inner):
  - Scan S: 500 page I/Os, 80*500 tuples
  - For each S tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching R tuples
  - Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered

How about B+-tree index??

General Join Conditions

- Equalities over several attributes (e.g., R.sid=S.sid AND R.rname=S.sname):
  - Join on one predicate, and treat the rest as selections;
    - Which attribute should be used for join??
  - For Index NL, build index on <sid, sname> (if S is inner); use existing indexes on sid or sname
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns
- Inequality join (R.sid < S.sid)
  - Nested loops is fine; index nested loops join requires a B+-tree index
  - Sort-merge join is fine – incur sorting overhead but save some cost in scanning of S
  - Hash-based joins are not directly applicable

Simple Selections

- Of the form: $\sigma_{R.attr\ op\ value}(R)$
- selectivity = Size of result / ||R||

```
SELECT  *
FROM    R
WHERE   weight>64 and
        height>170
```

Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176

$\sigma_{(weight>64)\wedge(height>170)}(R)$

name	age	weight	height
Curly	10	65	171
Larry	12	70	175
Charlie	20	69	173

Access Path

- Access path refers to a way of accessing data records/entries
  - Table scan
    - With no index, unsorted: Must essentially scan the whole relation; cost is M (#pages in R)
  - Index-only scan
    - An index containing the relevant information: Scan the index
  - Index search
    - With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. (You already know this – B+-tree and hash indexes)
  - Index intersection: Combine results from *multiple* index scans/retrieval (e.g., intersection, union)
- Index-based access paths (except index-only scan) can be followed by RID lookups to retrieve data records

Cost/Selectivity of an Access Path

- Cost/Selectivity of an access path = number of index and data pages retrieved to access data records/entries
- The *most selective* access path = one that retrieves the fewest pages
  - Usually, index-based access paths are superior, but table scans can win too

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering)
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples)
    - Clustered index: ~100 I/Os
    - Unclustered: ~ 10000 I/Os (since 1 I/O per tuple)

Two Approaches to General Selections

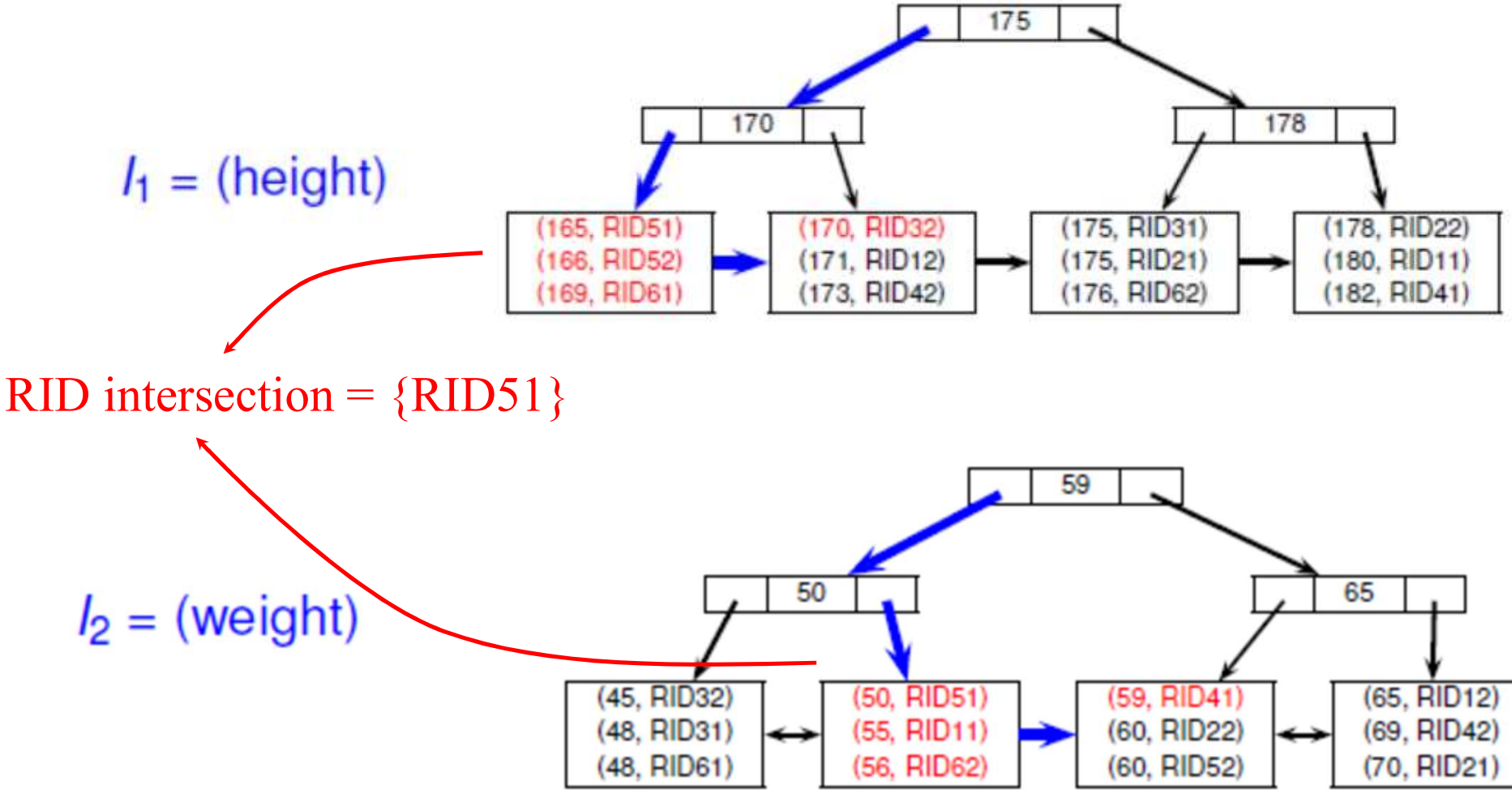
- First approach: Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - Most selective access path: An index or file scan that we estimate will require the fewest page I/Os
  - Terms that match this index reduce the number of tuples retrieved; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched
  - Consider day<8/9/94 AND bid=5 AND sid=3.
    - A B+ tree index on day can be used; then, bid=5 and sid=3 must be checked for each retrieved tuple
      - What is the I/O cost?
    - Similarly, a hash index on <bid, sid> could be used; day<8/9/94 must then be checked

Intersection of Rids

- Second approach (if we have 2 or more matching indexes (assuming leaf data entries are pointers):
  - Get sets of rids of data records using each matching index
  - Then intersect these sets of rids (we'll discuss intersection soon!)
  - Retrieve the records and apply any remaining terms
  - Consider day<8/9/94 AND bid=5 AND sid=3
    - If we have a B+ tree index on day and an index on sid, we can retrieve rids of records satisfying day<8/9/94 using the first, rids of recs satisfying sid=3 using the second, intersect the rids, retrieve records and check for bid=5

B$^+$-tree: Index Intersection

select height, weight **from** Student
where height **between** 164 **and** 170
and weight **between** 50 **and** 59



The Projection Operation (Duplicate Elimination)

- $\pi_L(R)$ 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

$\pi_{age}(R)$

age
17
15
10
12

Relational algebra vs SQL (relational DBMS)

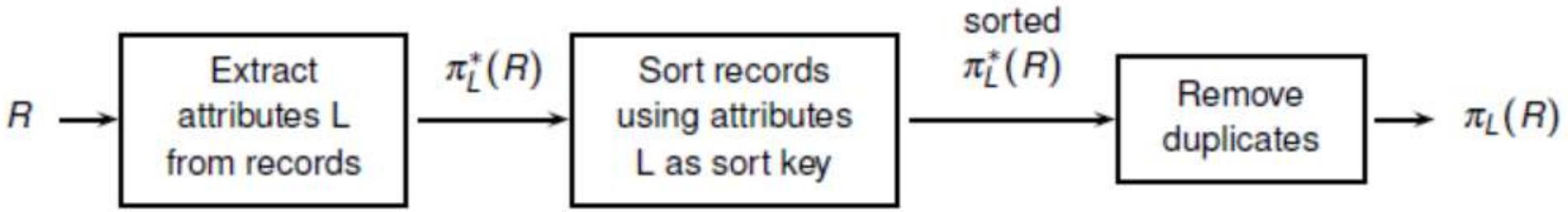
$\pi^*_{age}(R)$

age
17
15
10
12
17
10

- What about duplicates?
  - $\pi^*_L(R)$ same as $\pi_L(R)$ but preserves duplicates

The Projection Operation: Sort-based Approach

naive algorithm based on sorting



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

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

age
17
15
10
12
17
10

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

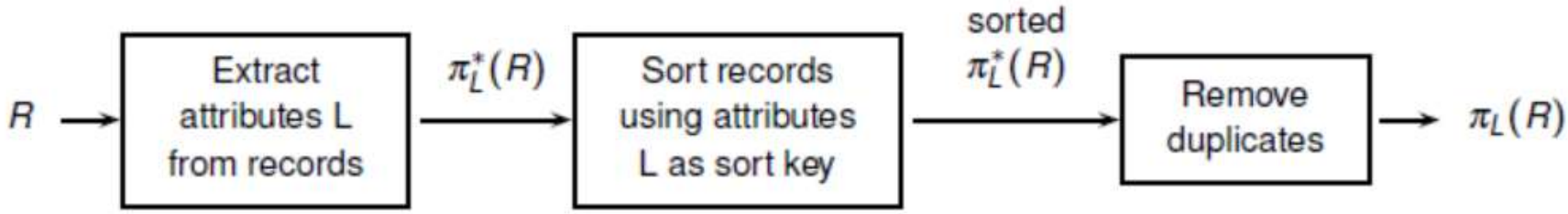
age
10
10
12
15
17
17

$\pi_{age}(R)$

age
10
12
15
17

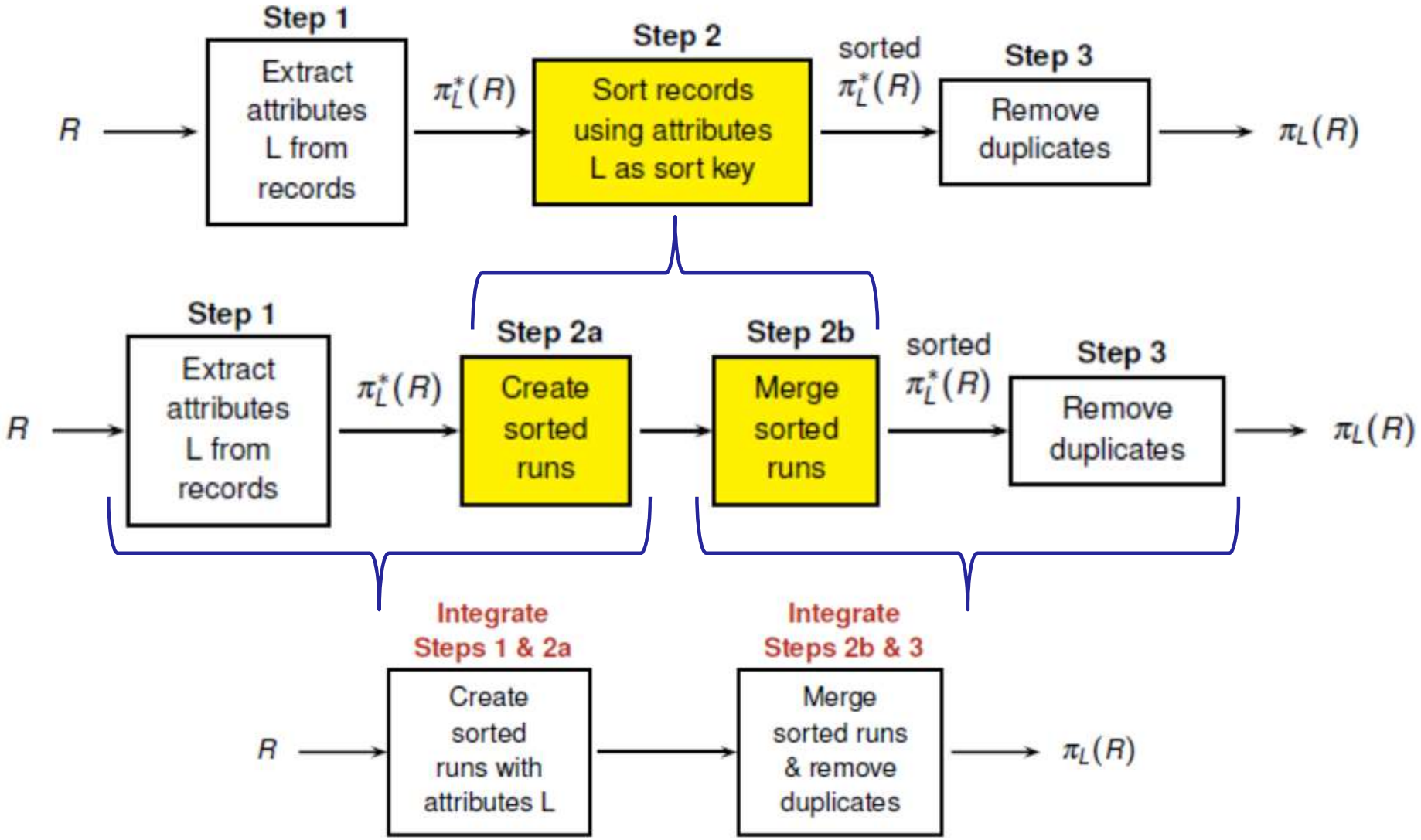
removing duplicates will scan the output again

Sort-based Approach: 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; m = merge factor
- Step 3:
  - Cost to scan records = $|\pi^*_L(R)|$
  - Optional: Cost to store answer = $|\pi_L(R)|$

Optimized Sort-based Approach



Optimized Sort-based Approach

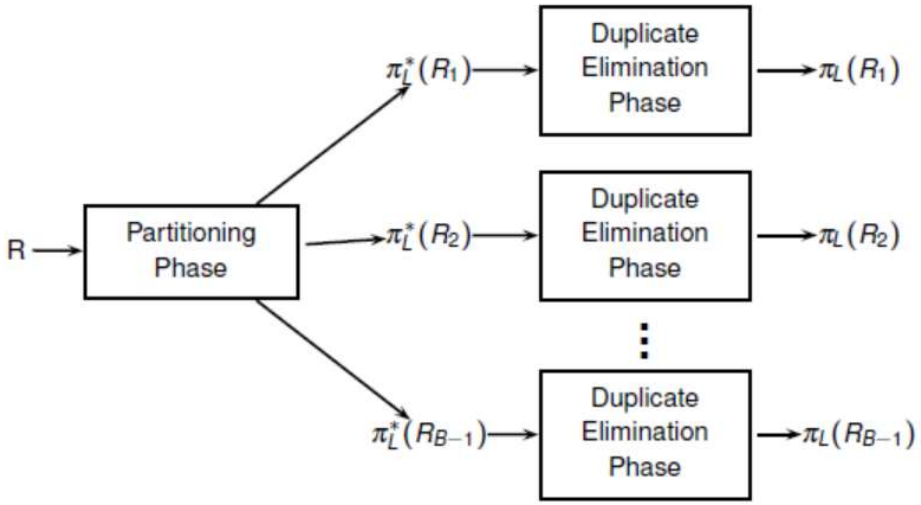
- An approach based on sorting:
  - Modify phase 1 of external sort to eliminate unwanted fields
    - Runs are produced, but tuples in runs are smaller than input tuples (Size ratio depends on # and size of fields that are dropped)
  - Modify merging passes to eliminate duplicates
    - Number of result tuples smaller than input (Difference depends on # of duplicates)
  - Cost:
    - In phase 1, read original relation (size M), write out same number of smaller tuples
    - In merging passes, fewer tuples written out in each pass

Hash-based Approach

Consists of two phases:

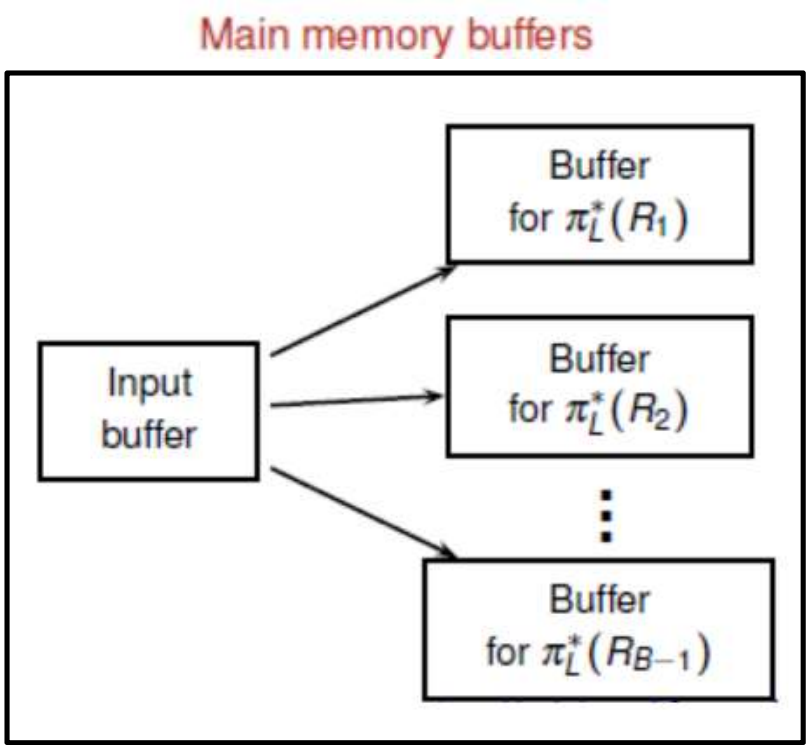
1. Partitioning phase: partitions R into $R_1, R_2, \cdots, R_{B-1}$
   - Hash on $t.L$ for each tuple $t \in R$
   - $R = R_1 \cup R_2 \cup \cdots \cup R_{B-1}$
   - $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$ for each pair R_i & R_j, $i \neq j$
2. Duplicate elimination phase: eliminates duplicates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

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



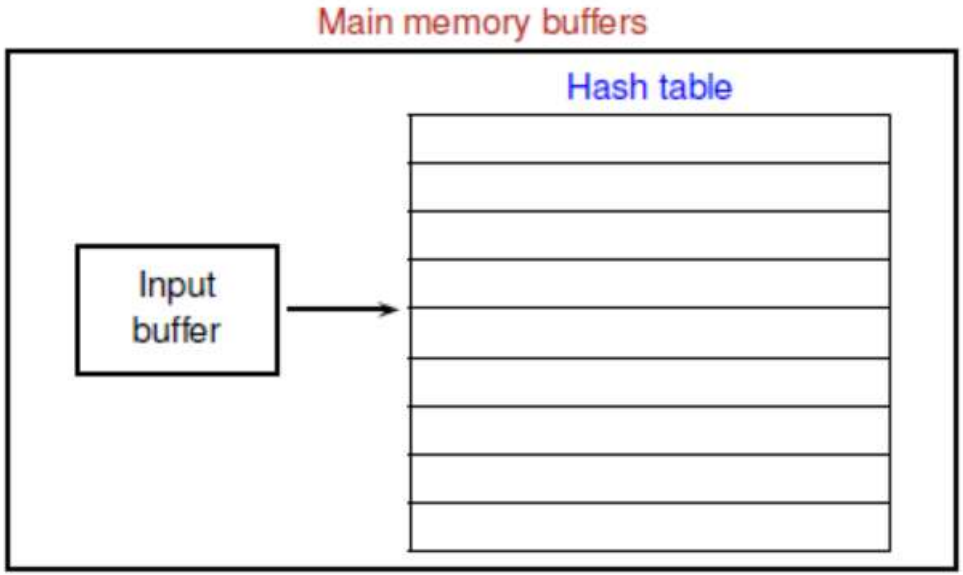
Partitioning Phase

- Use one buffer for input and (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
  - Optimization: If it so happen that a duplicate is found in the output buffer, can remove it immediately. Overhead??

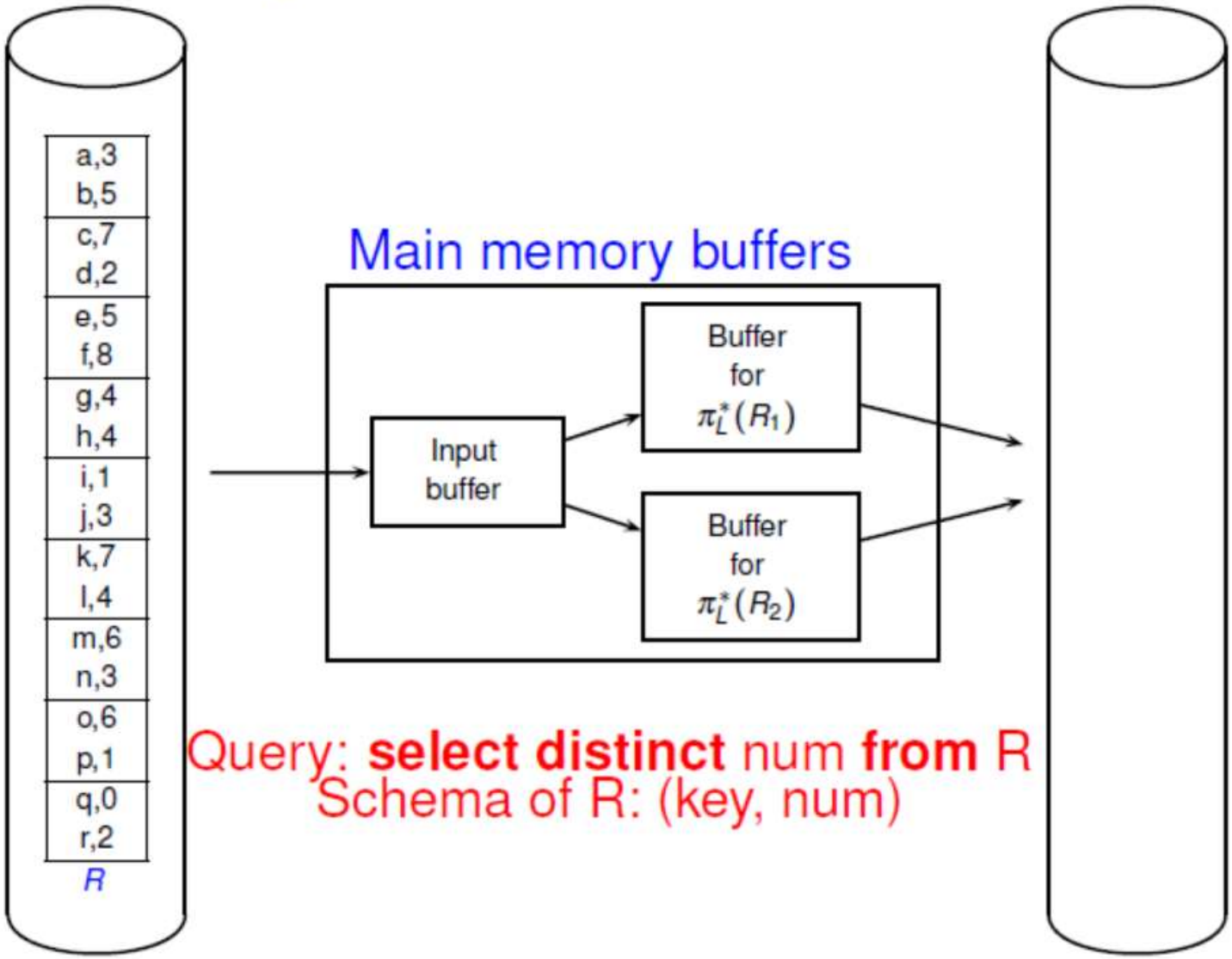


Duplicate Elimination Phase

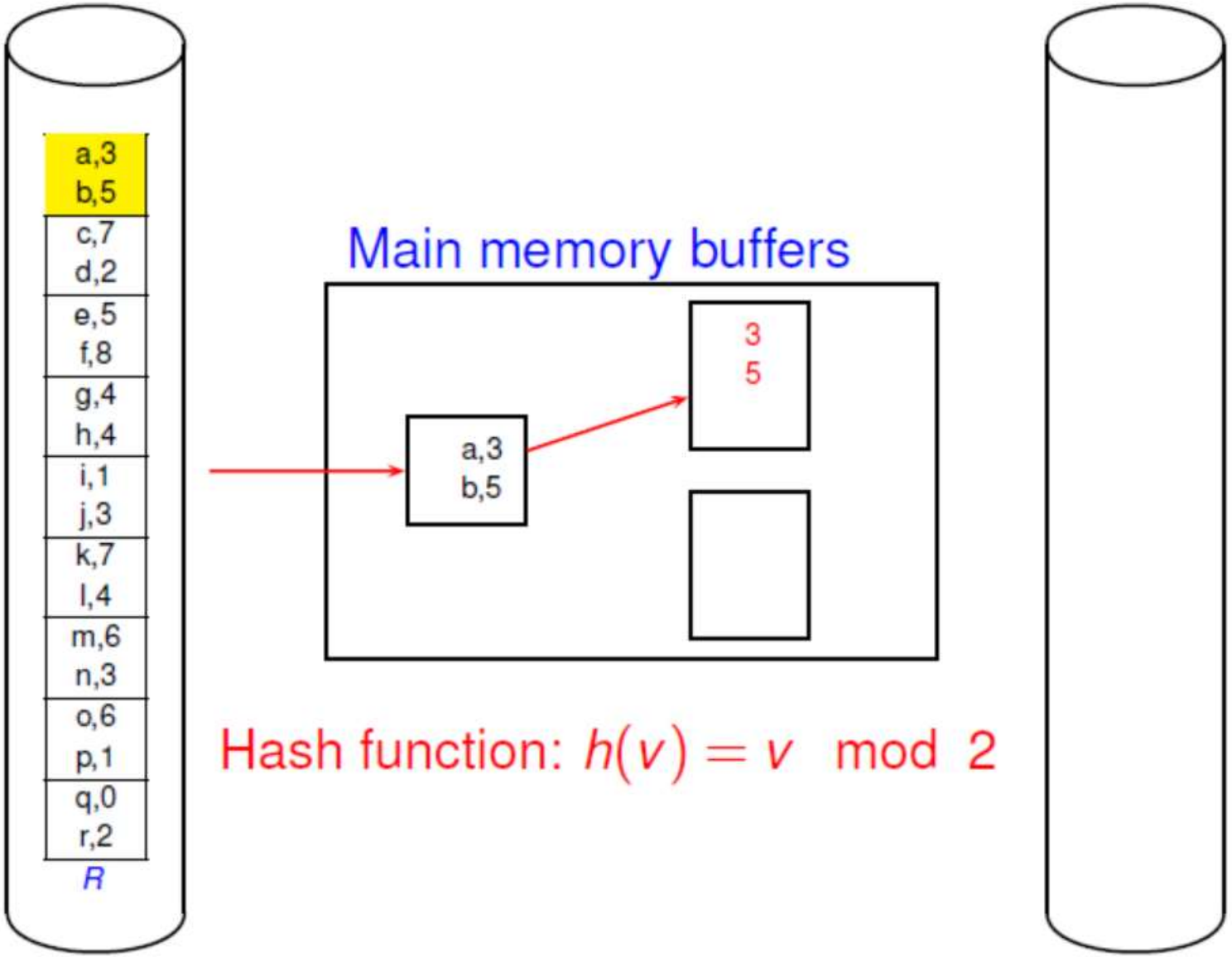
- For each partition R_j
  - Initialize an in-memory hash table
  - Read $\pi^*_L(R_j)$ one page at a time; for each tuple t read
    - Hash t into bucket B_j with hash function h' (h' ≠ h)
    - Insert t into B_j if t ∉ B_j
  - Write out tuples in hash table to results



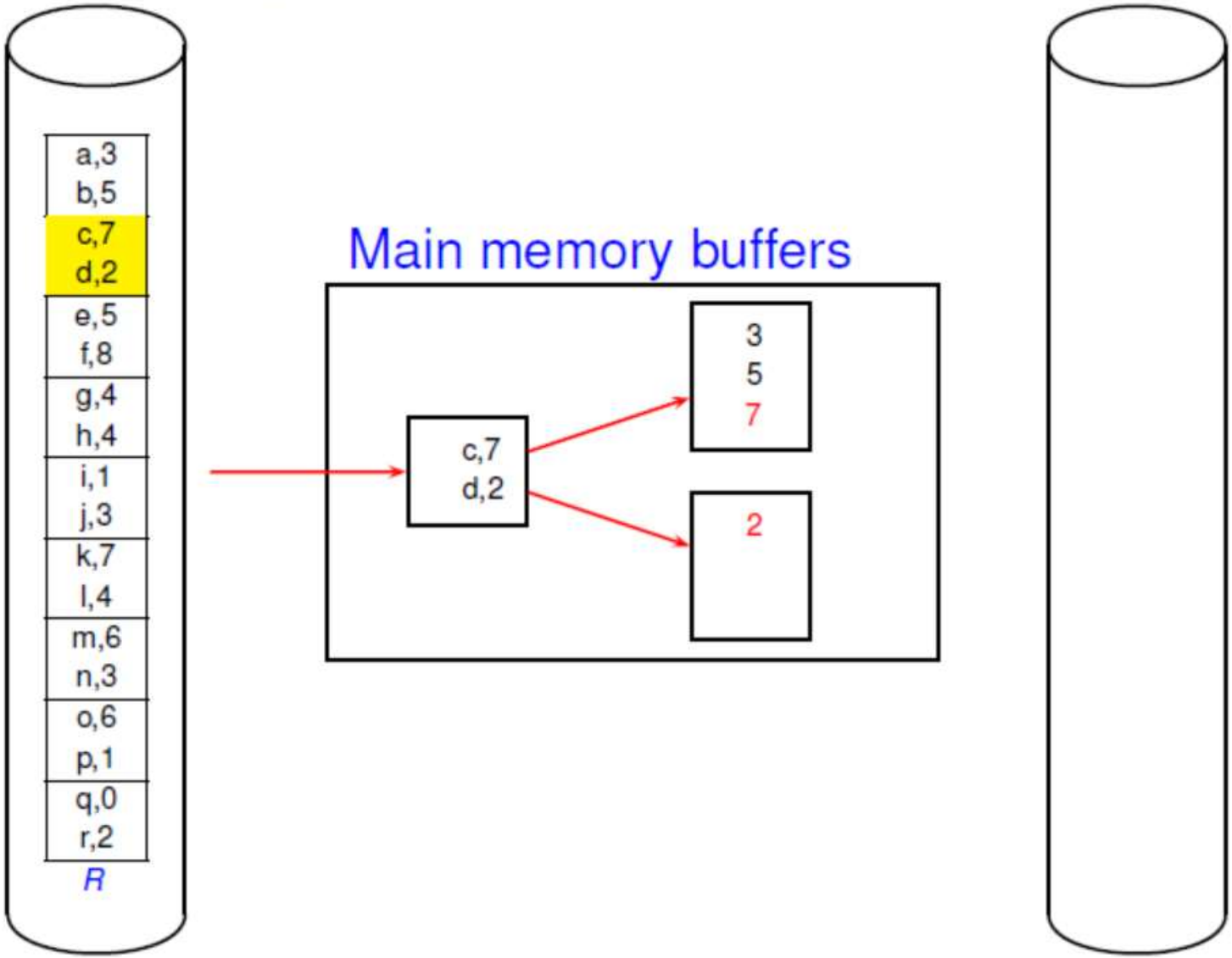
Example: Partitioning Phase



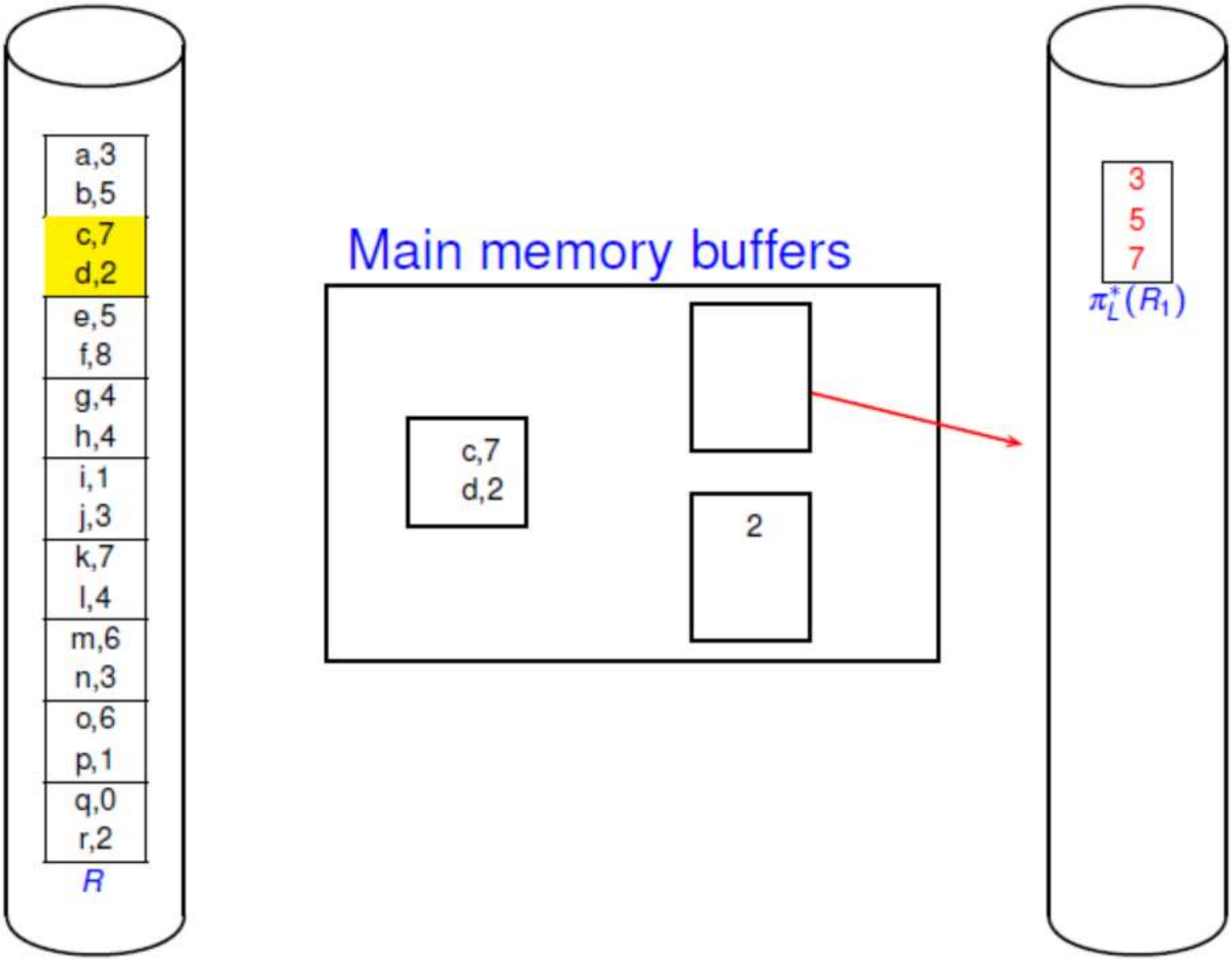
Example: Partitioning Phase



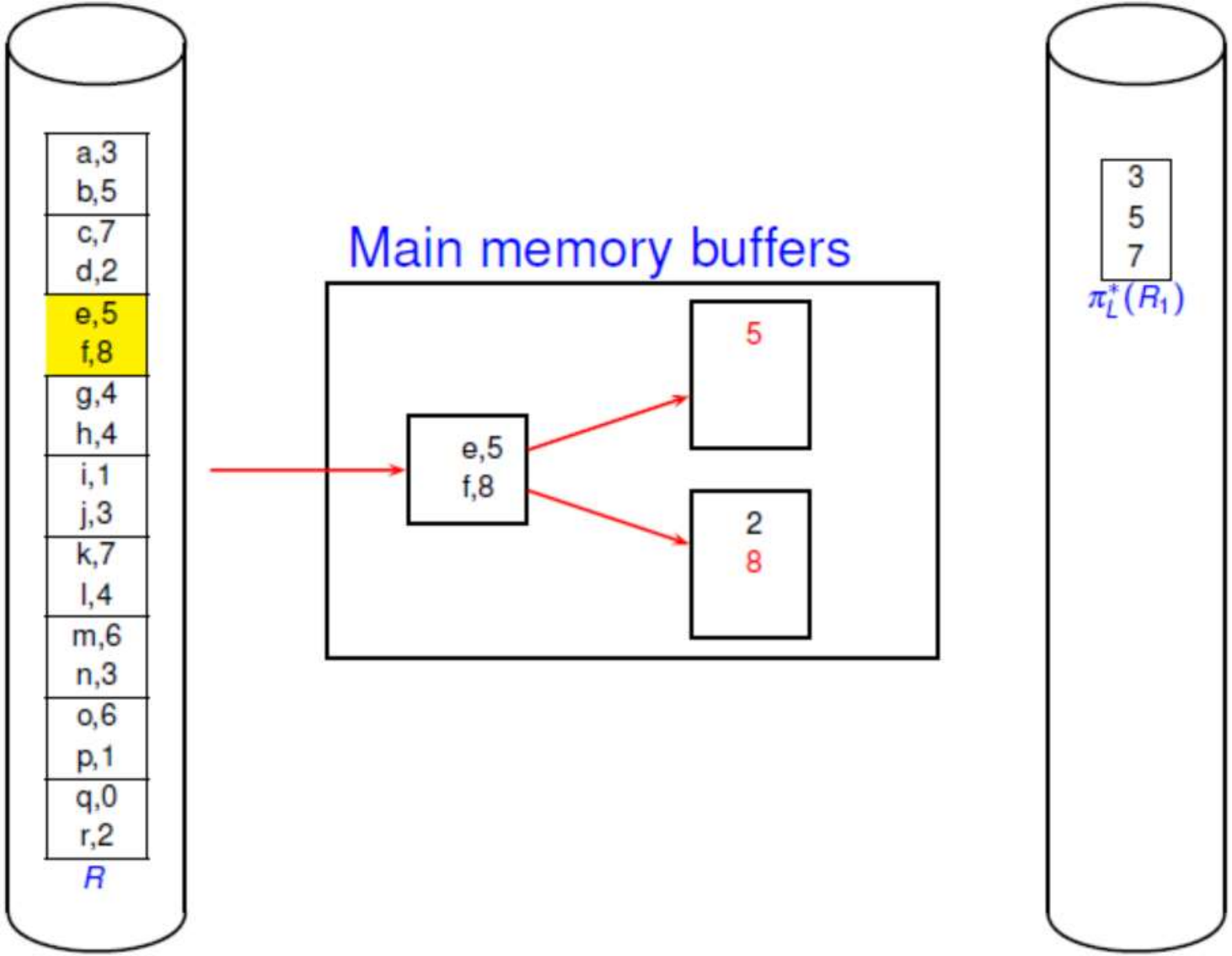
Example: Partitioning Phase



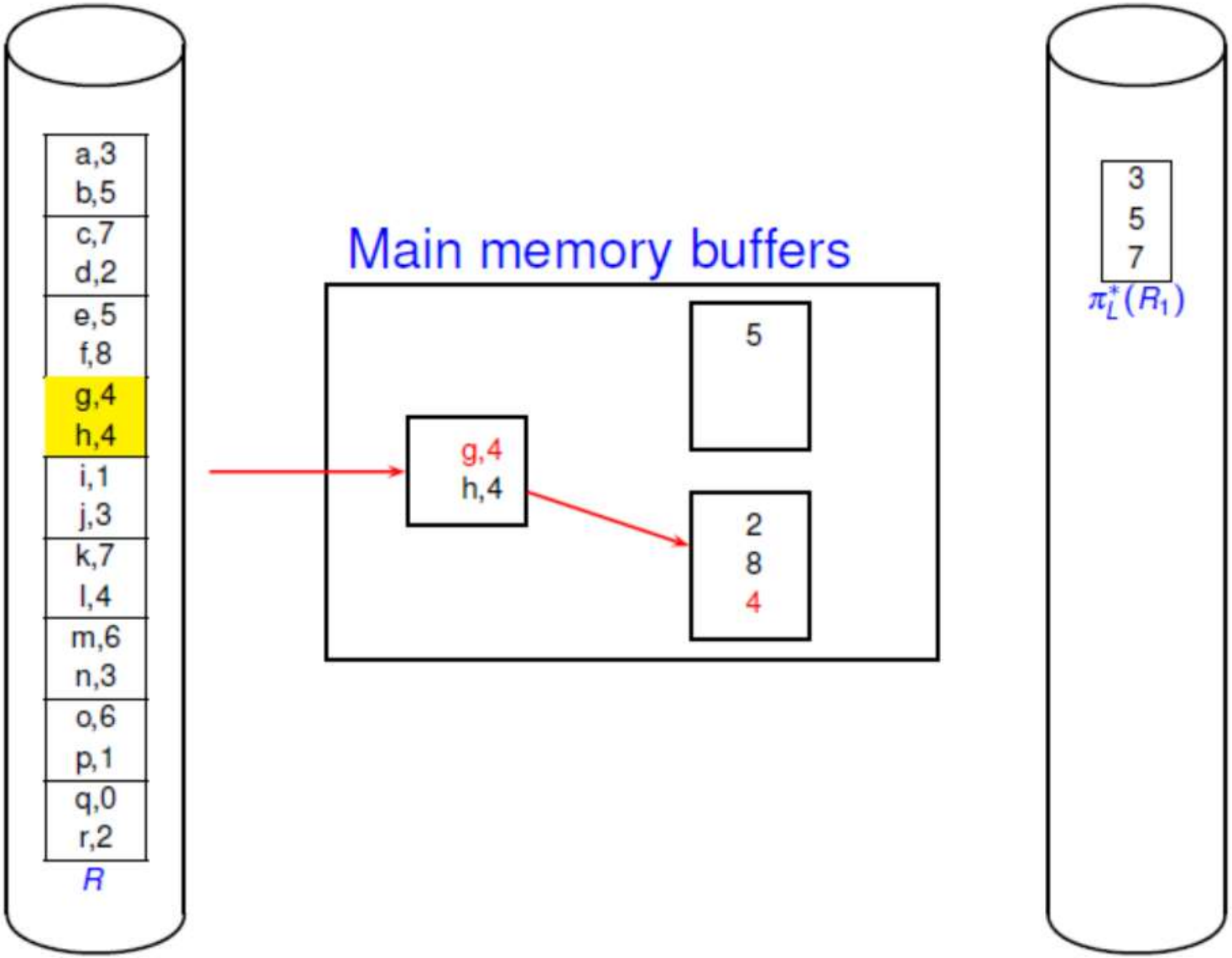
Example: Partitioning Phase



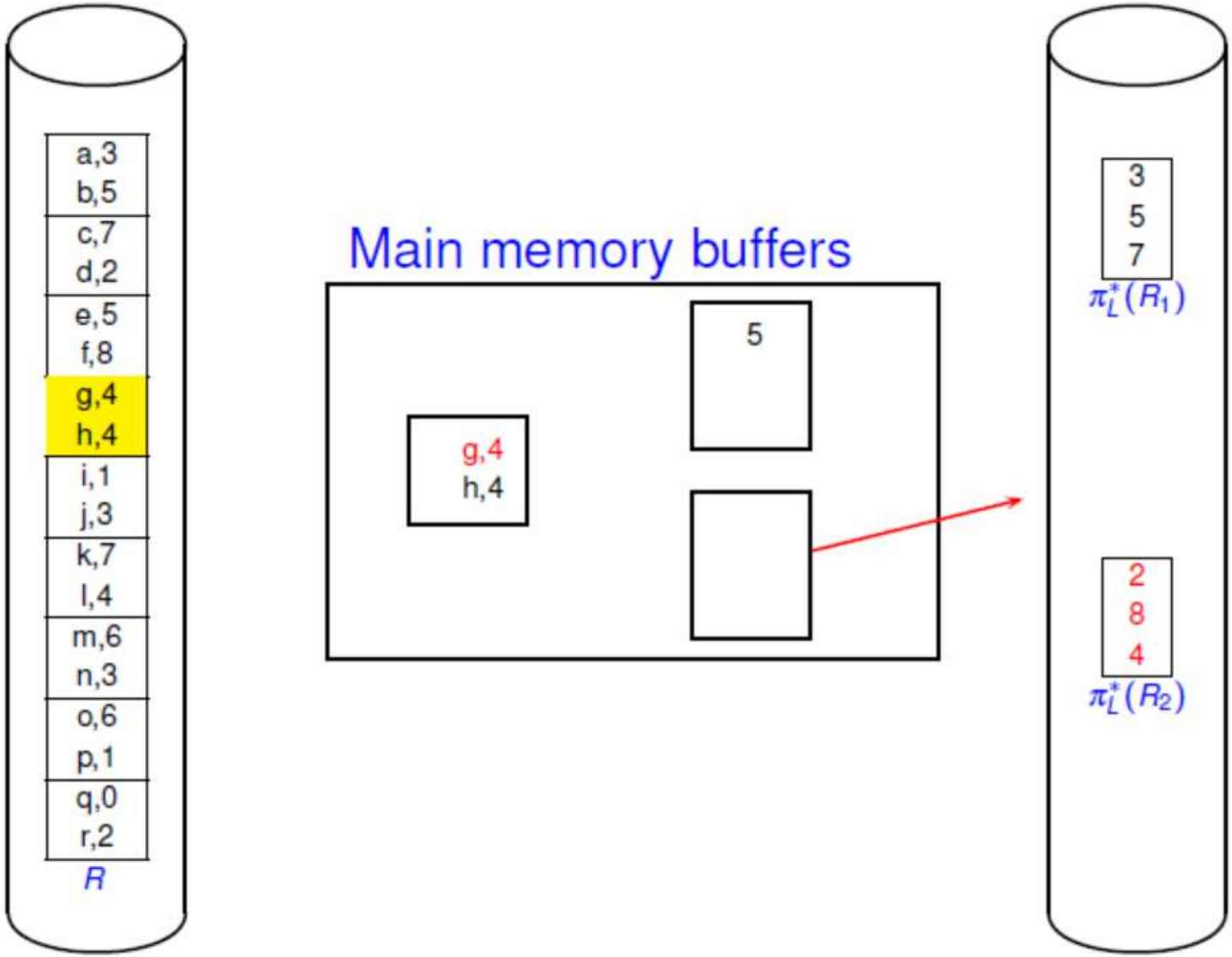
Example: Partitioning Phase



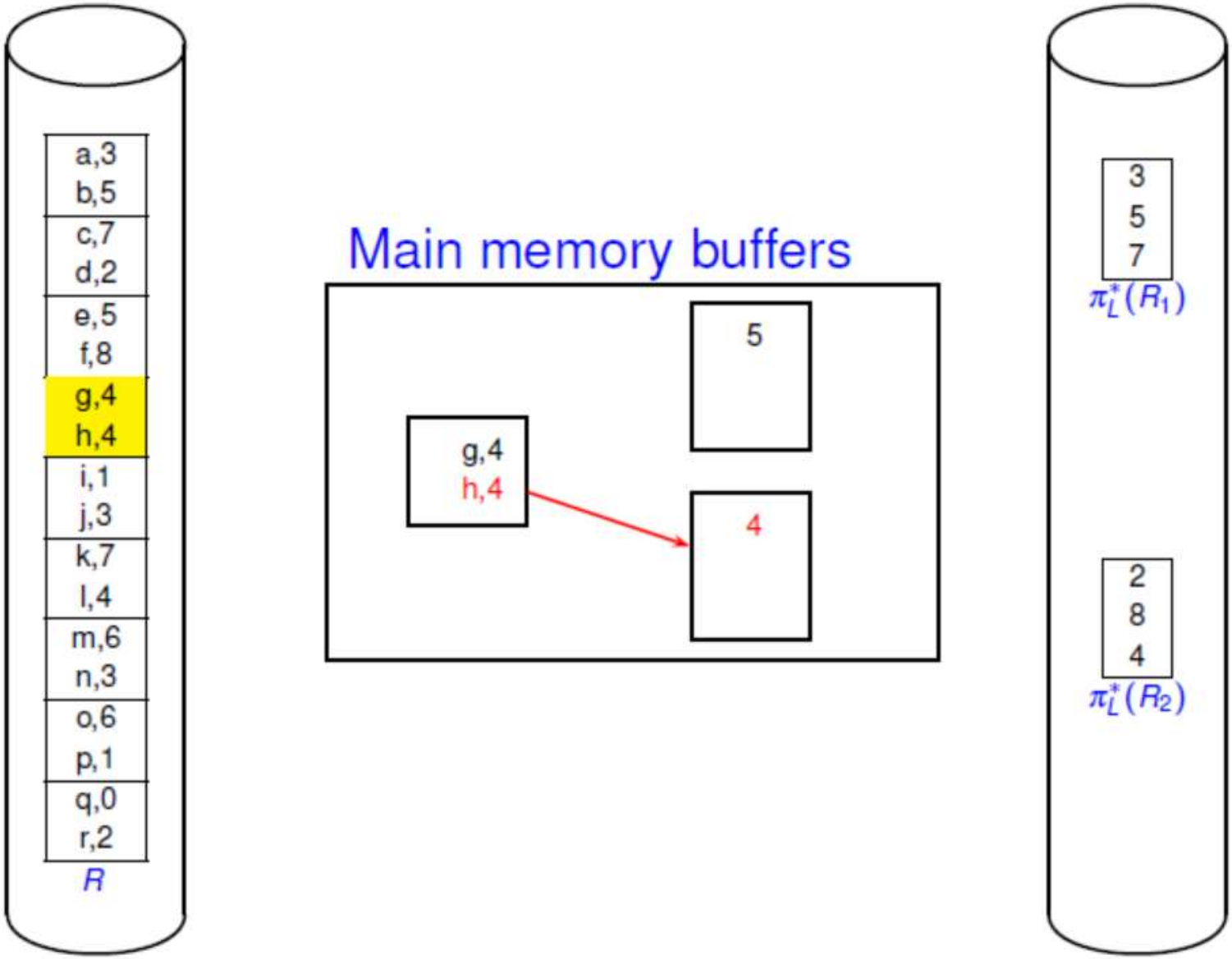
Example: Partitioning Phase



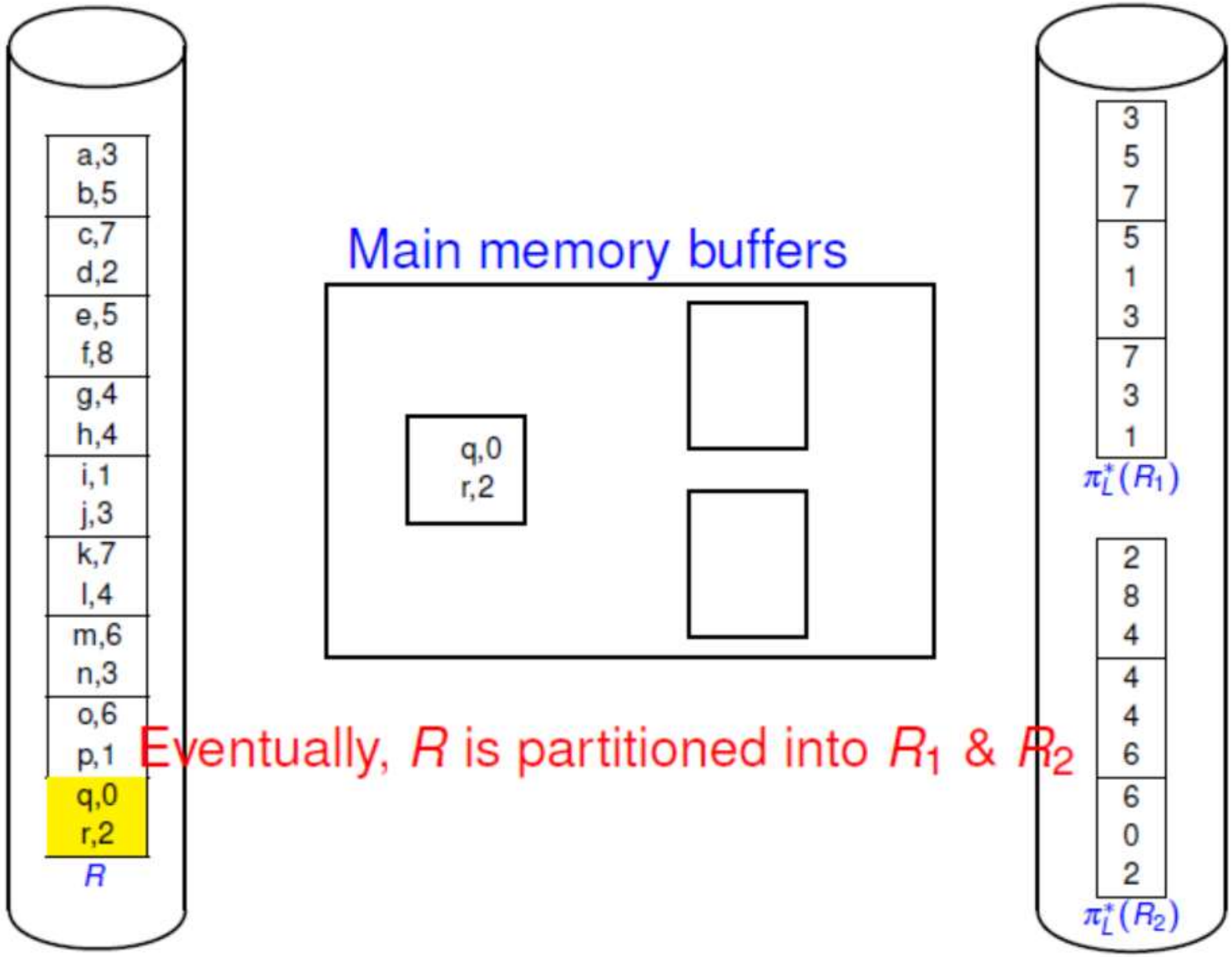
Example: Partitioning Phase



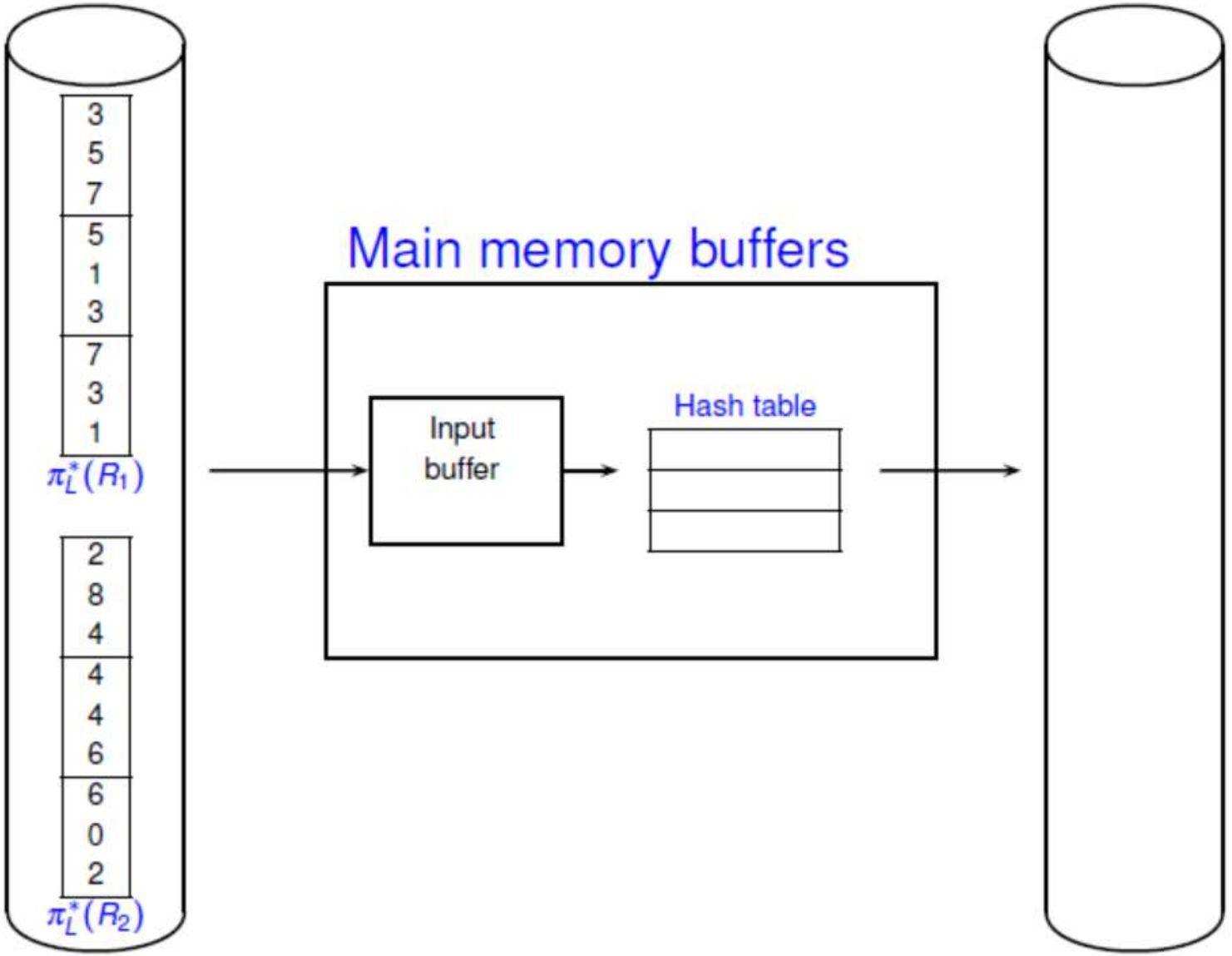
Example: Partitioning Phase



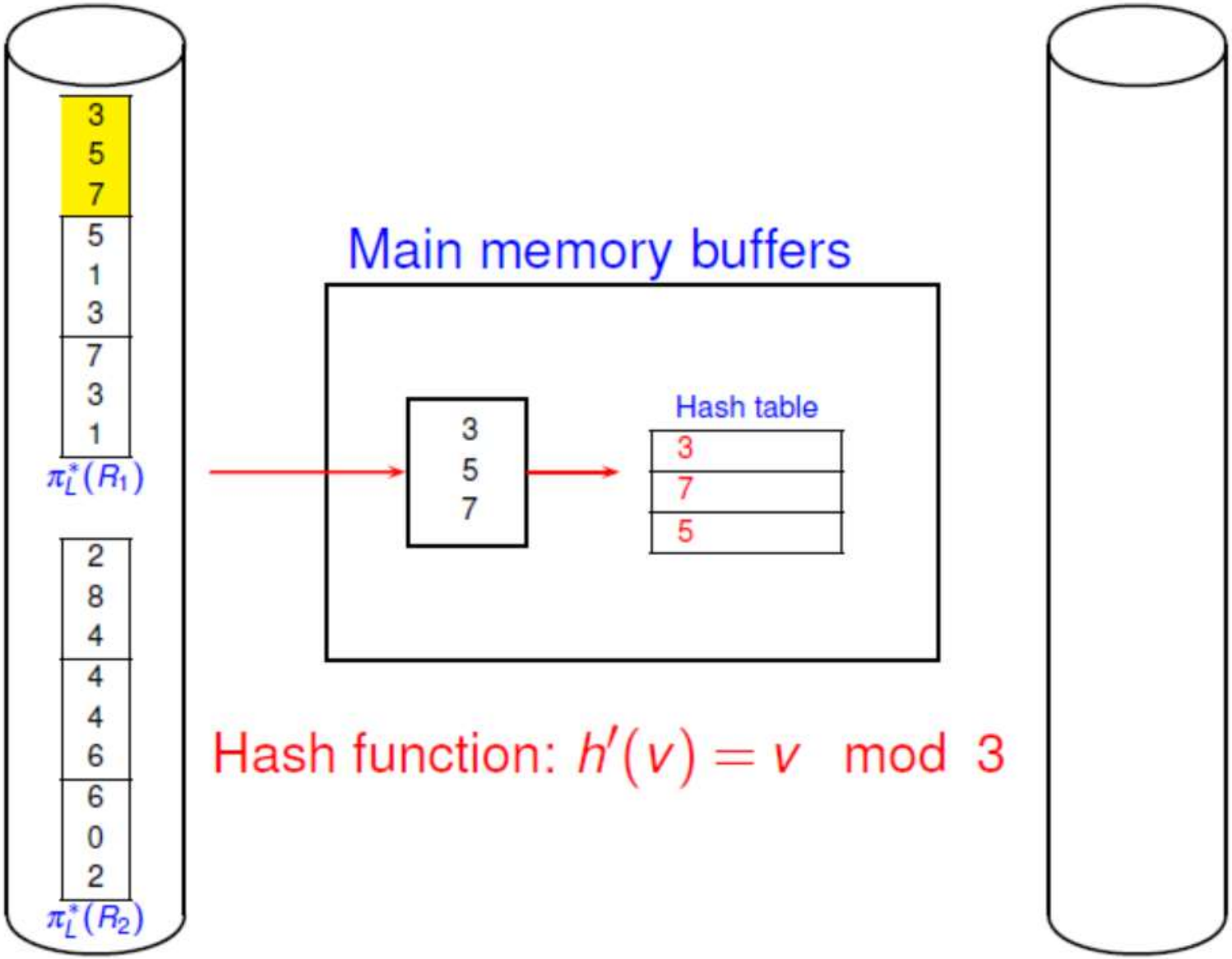
Example: Partitioning Phase



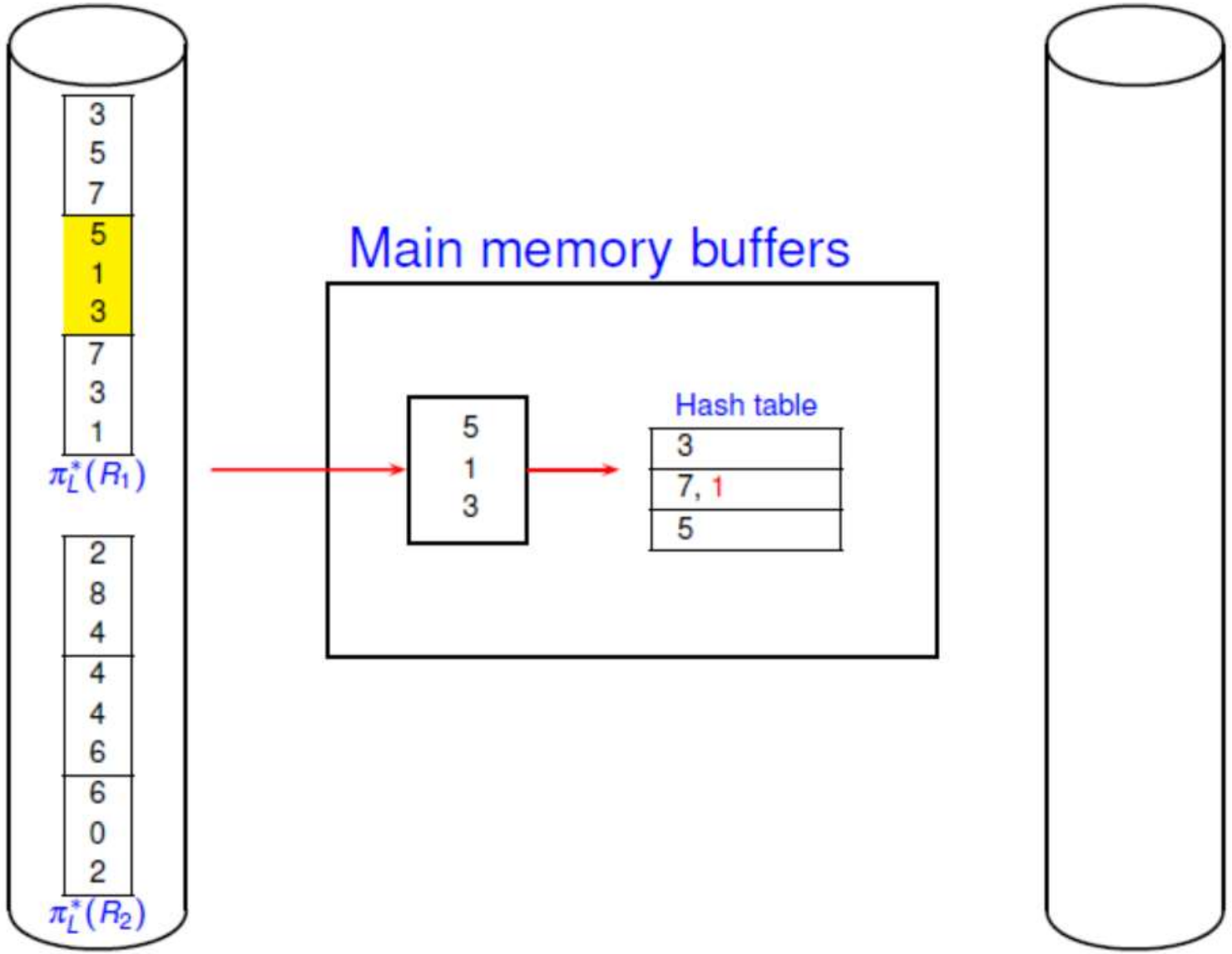
Example: Duplicate Elimination Phase



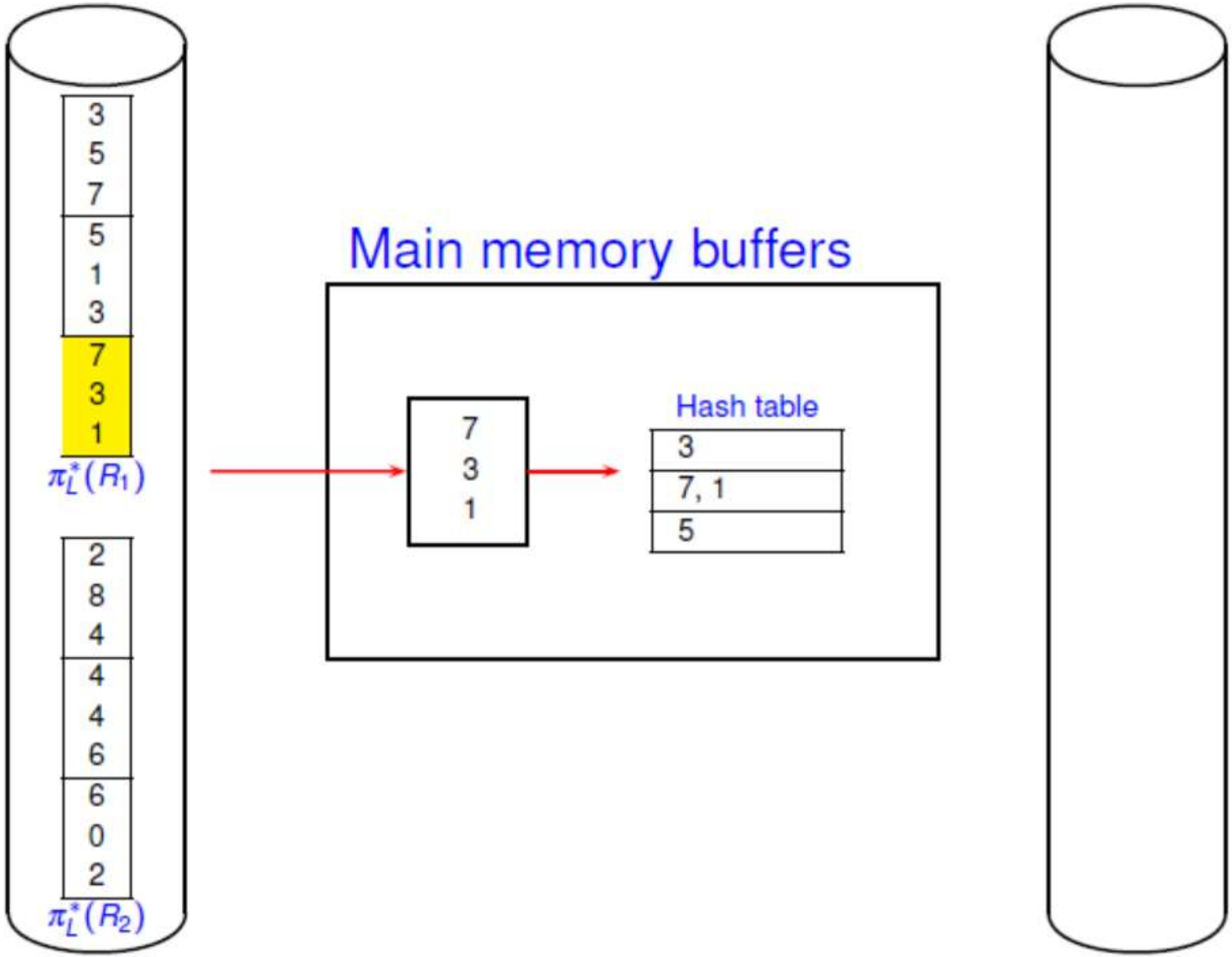
Example: Duplicate Elimination Phase



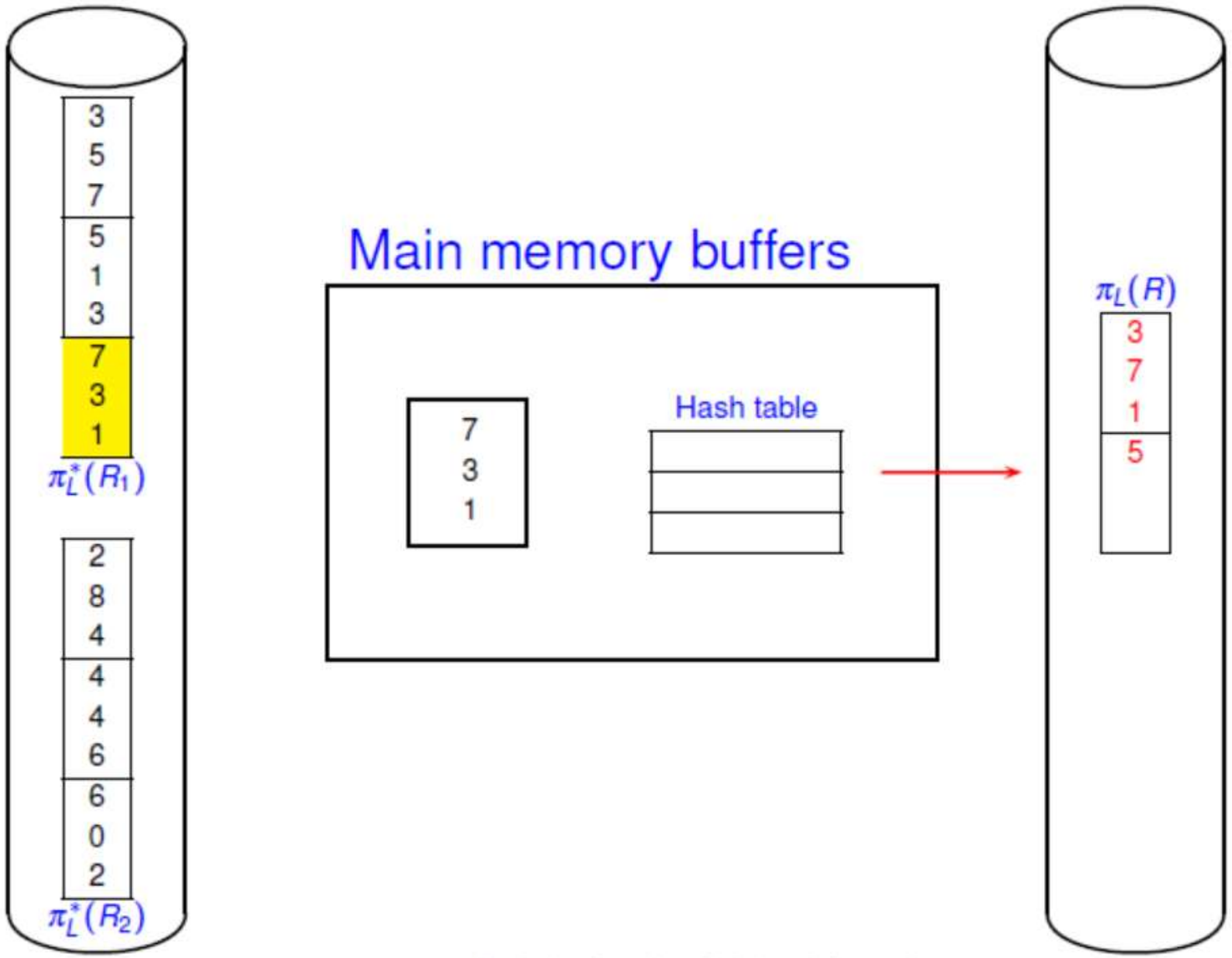
Example: Duplicate Elimination Phase



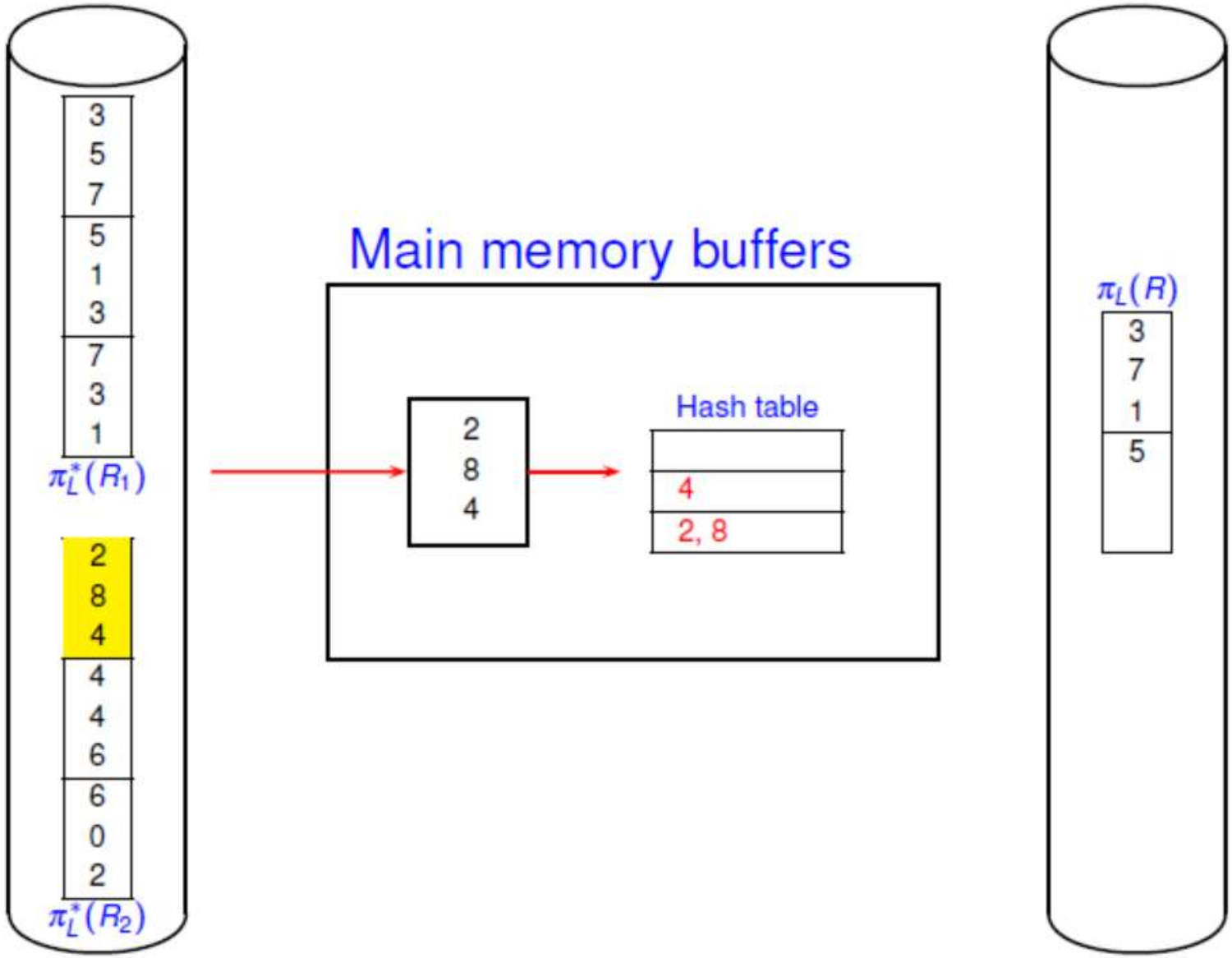
Example: Duplicate Elimination Phase



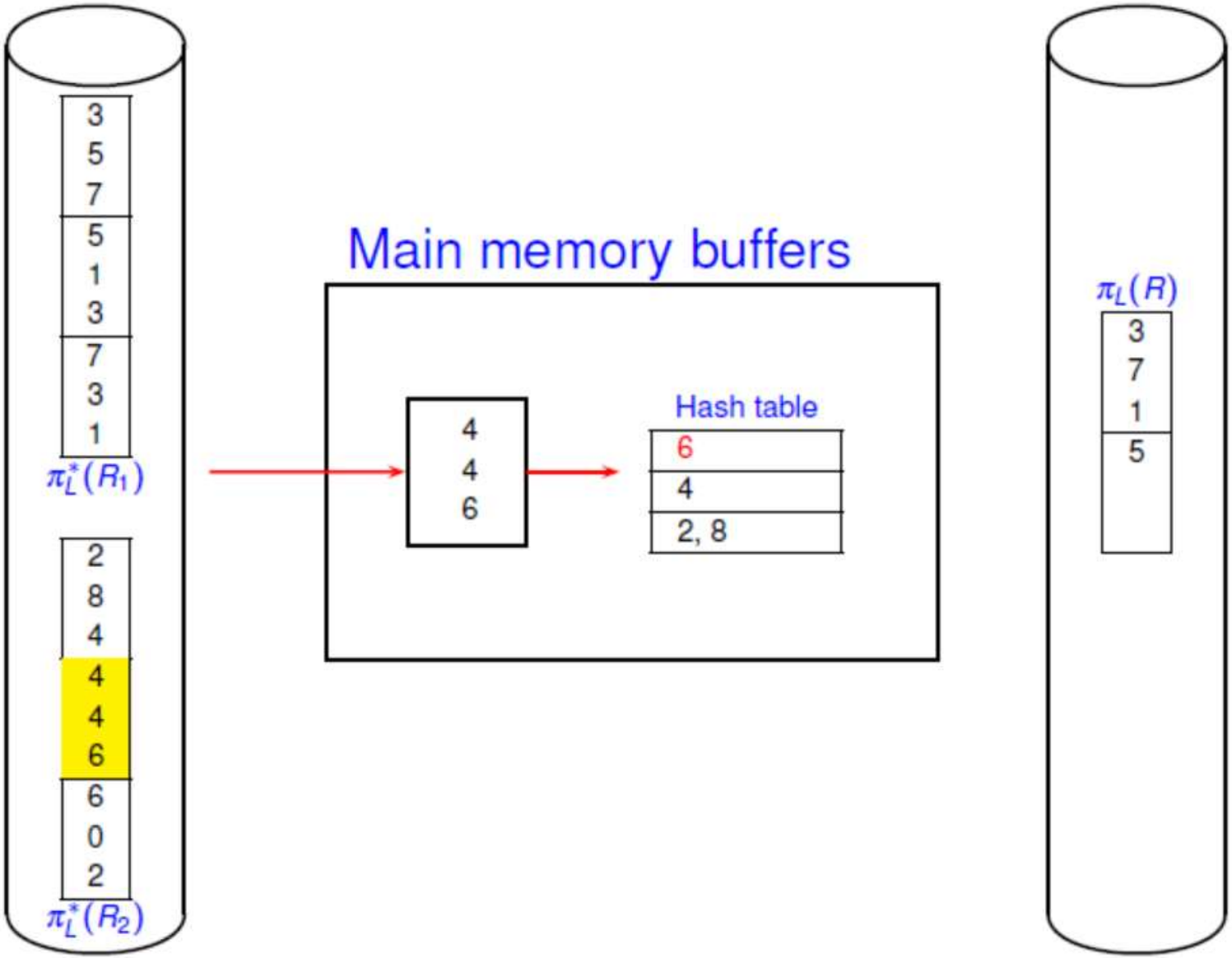
Example: Duplicate Elimination Phase



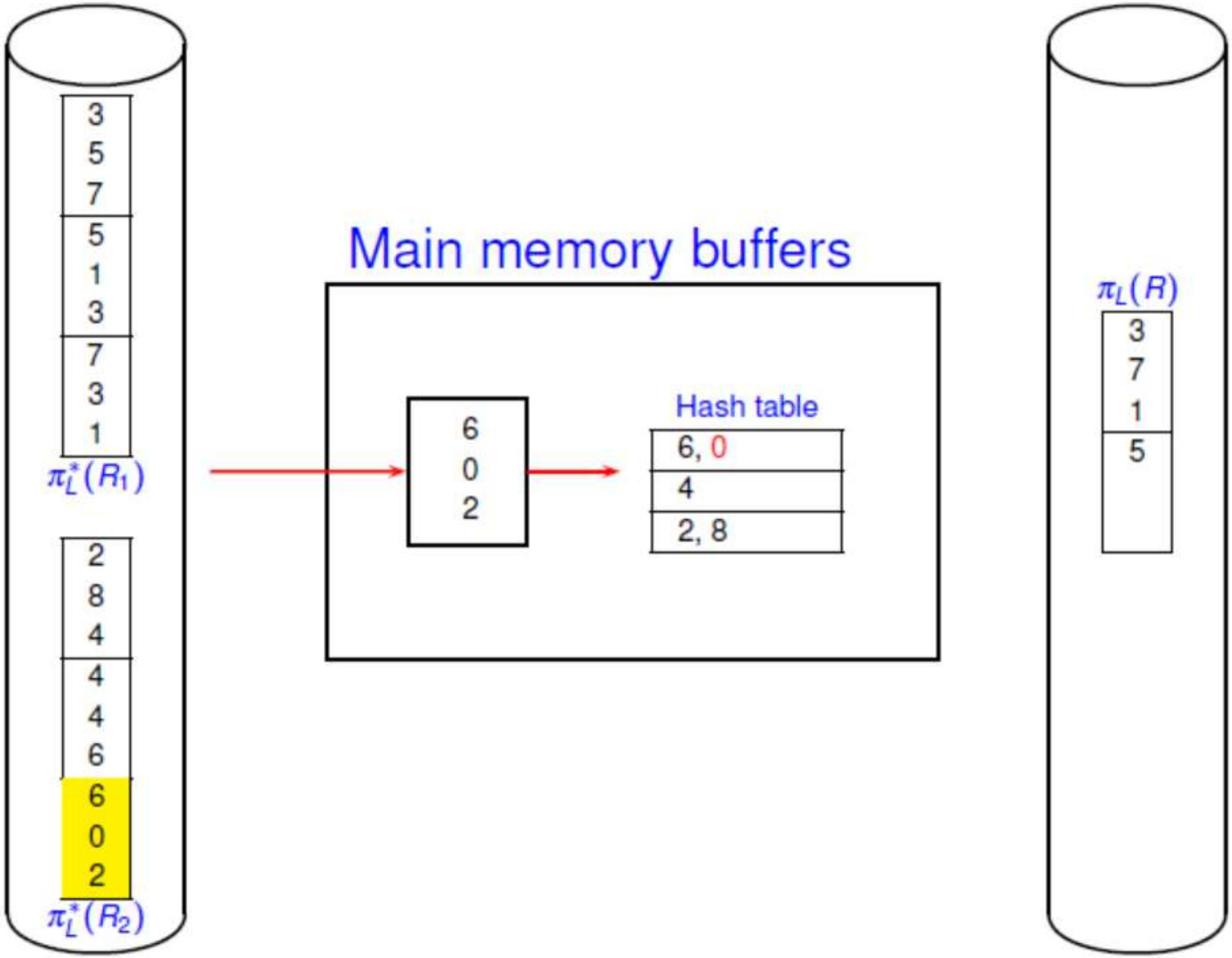
Example: Duplicate Elimination Phase



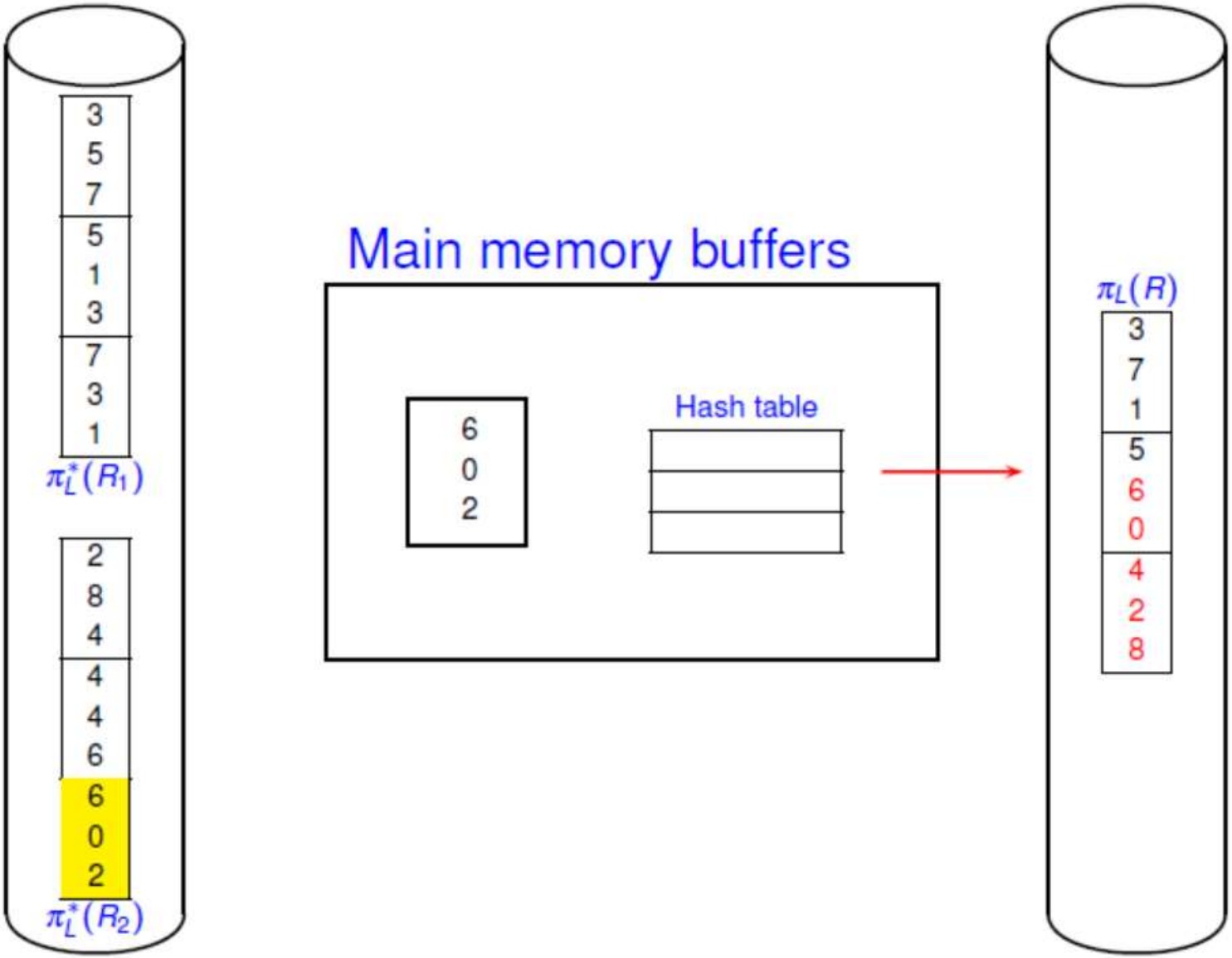
Example: Duplicate Elimination Phase



Example: Duplicate Elimination Phase

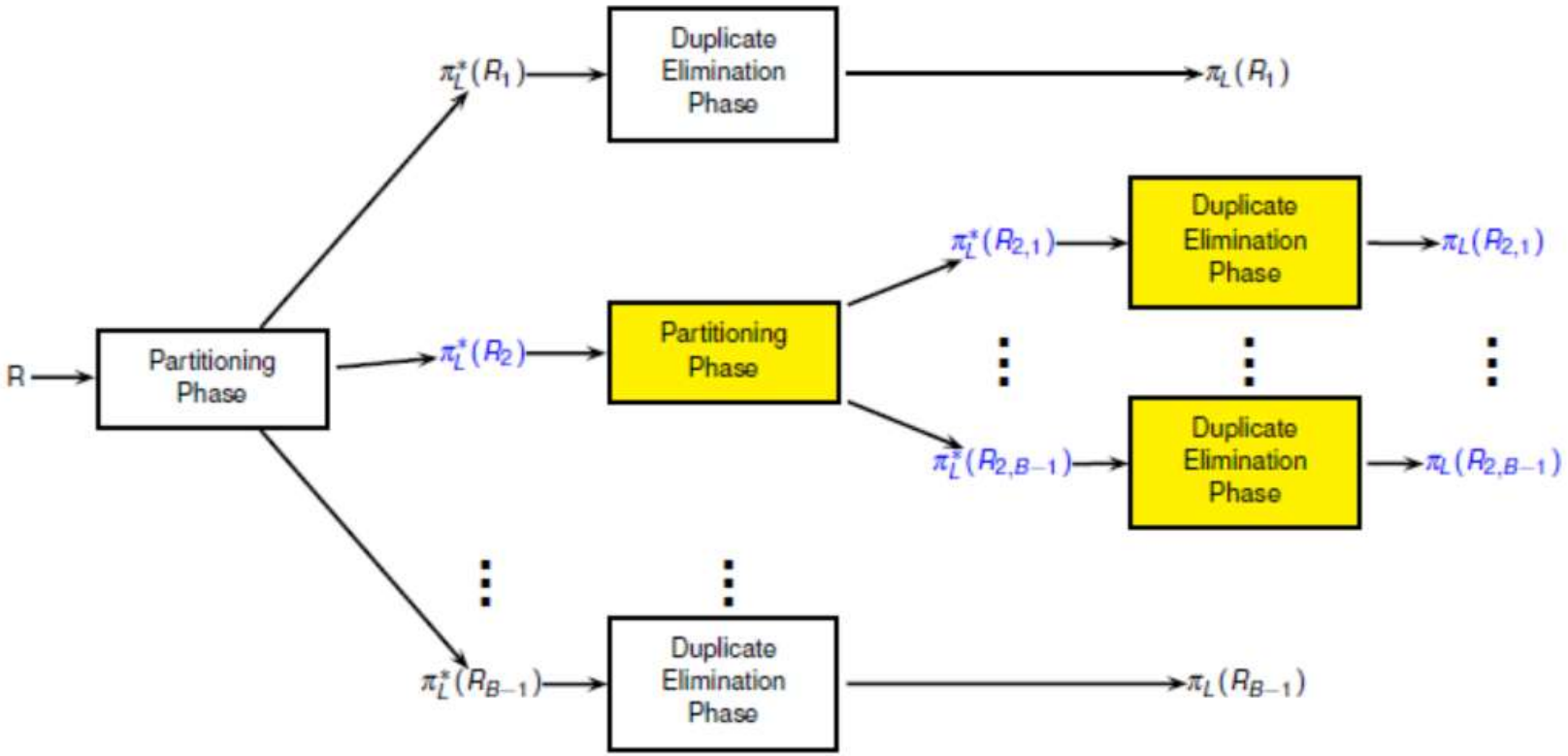


Example: Duplicate Elimination Phase



Hash-based Approach: Partition Overflow

- What happen if $\pi^*_L(R_j)$ is larger than the available memory?
  - Recursively apply hash-based partitioning to the overflowed partition



Cost Analysis of Hash-based Approach

- 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 = *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:
  - Cost of partitioning phase: $|R| + |\pi_L^*(R)|$
  - Cost of duplicate elimination phase: $|\pi_L^*(R)|$
  - Total cost = $|R| + 2|\pi_L^*(R)|$

(ignore cost to write output)

What if an index is available?

- If the search key of an index contains all the wanted attributes
  - Replace table scan with index scan!
- If the 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 $\pi_A(R)$

Set Operations

- Set operations
  - Cross-product: R x S
  - Intersection: $R \cap S$
  - Union: $R \cup S$
  - Difference: R – S
- Intersection and cross-product: special cases of join
  - $R(A, B) \cap S(A, B) = \pi_{R.*}(R \bowtie_p S)$
    - $p = (R.A = S.A) \wedge (R.B = S.B)$
  - R x S = $R \bowtie S$ with an empty join predicate
- Union (Distinct) and Difference are similar
  - Sorting based approach
  - Hash based approach

Set Operations

- Sorting based approach to union:
  - Sort both relations (on combination of all attributes)
  - Scan sorted relations and merge them
    - Implementation typically removes duplicates while merging (why?)
- Hash based approach to union:
  - Partition R and S using hash function h
  - For each S-partition, build in-memory hash table (using h2), scan corr. R-partition and add tuples to table while discarding duplicates
- Algorithms for R – S are similar

Aggregate Operations (COUNT, SUM, AVG, MIN, MAX)

```
SELECT AVG(SALARY)
FROM EMPLOYEE
```

- Without grouping:
  - In general, requires scanning the relation
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan

Aggregate Operations (COUNT, SUM, AVG, MIN, MAX)

```
SELECT DEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DEPT
```

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group
  - Similar approach based on hashing on group-by attributes
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order)(no sorting needed)

Summary

- A virtue of relational DBMSs: queries are composed of a few basic operators; the implementation of these operators can be carefully tuned (and it is important to do this!)
- Many alternative implementation techniques for each operator; no universally superior technique for most operators
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops