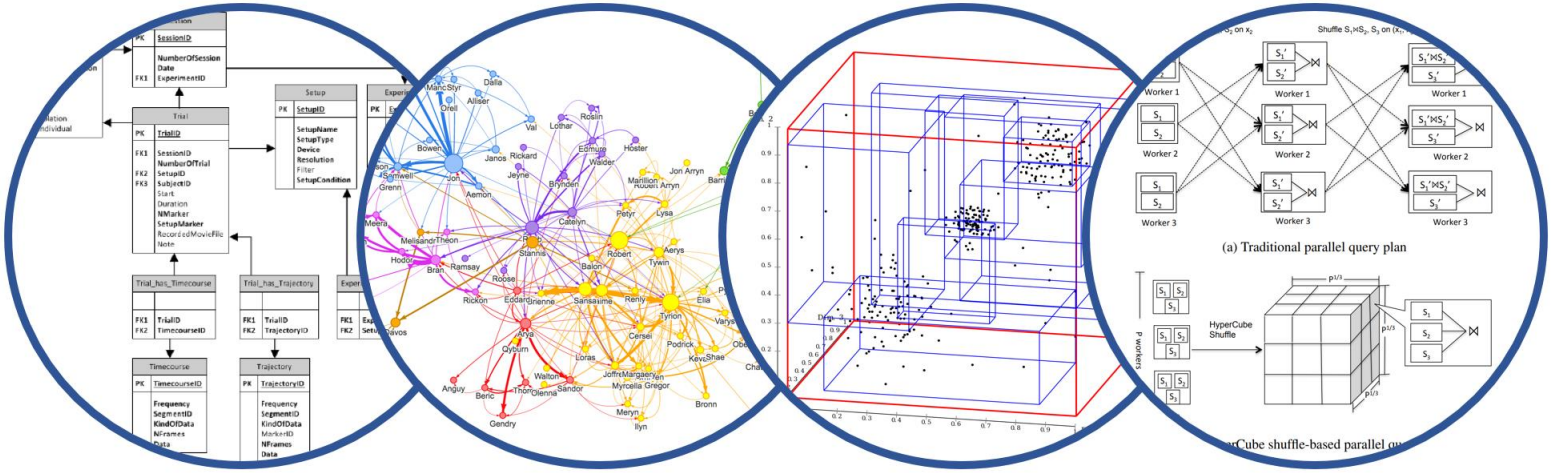


Introduction to Data Management

External Memory Algorithms

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

HW6: Part 2 due 11/27. More work than part 1

Announcements

- Lecture on Wednesday, Nov. 27:
  - No in-person lecture
  - Recording only
- Monday, Dec. 9, Final Review session
  - CSE2 G10
  - 10:30 – 12:20 (we may finish earlier)

External Memory Operators

Overview

Parameters:

- T(R) = number of tuples in the relation R
- B(R) = number of blocks containing R
- V(R,A) = number of distinct values of attribute R.A

- M = size of main memory in #pages (blocks)

Goal: minimize number disk I/O’s

Observations

- B(R) ≤ T(R) why?
- We assume data is uniformly distributed. Then:

$$|\sigma_{A=value}(R)| = \frac{T(R)}{V(R,A)}$$

External Memory Join Operators

Selection: index-based

Joins:

- Index based join
- Block Nested Loop Join
- Partitioned Hash Join
- Merge Join

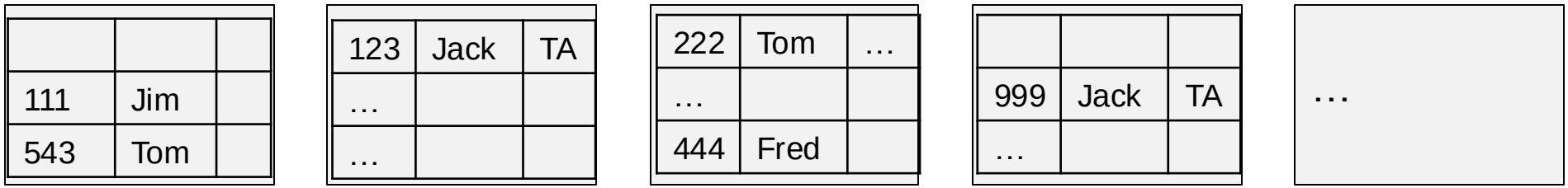
Index-based Selection

Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$$\sigma_{name=Jack}(Payroll)$$



Payroll data file

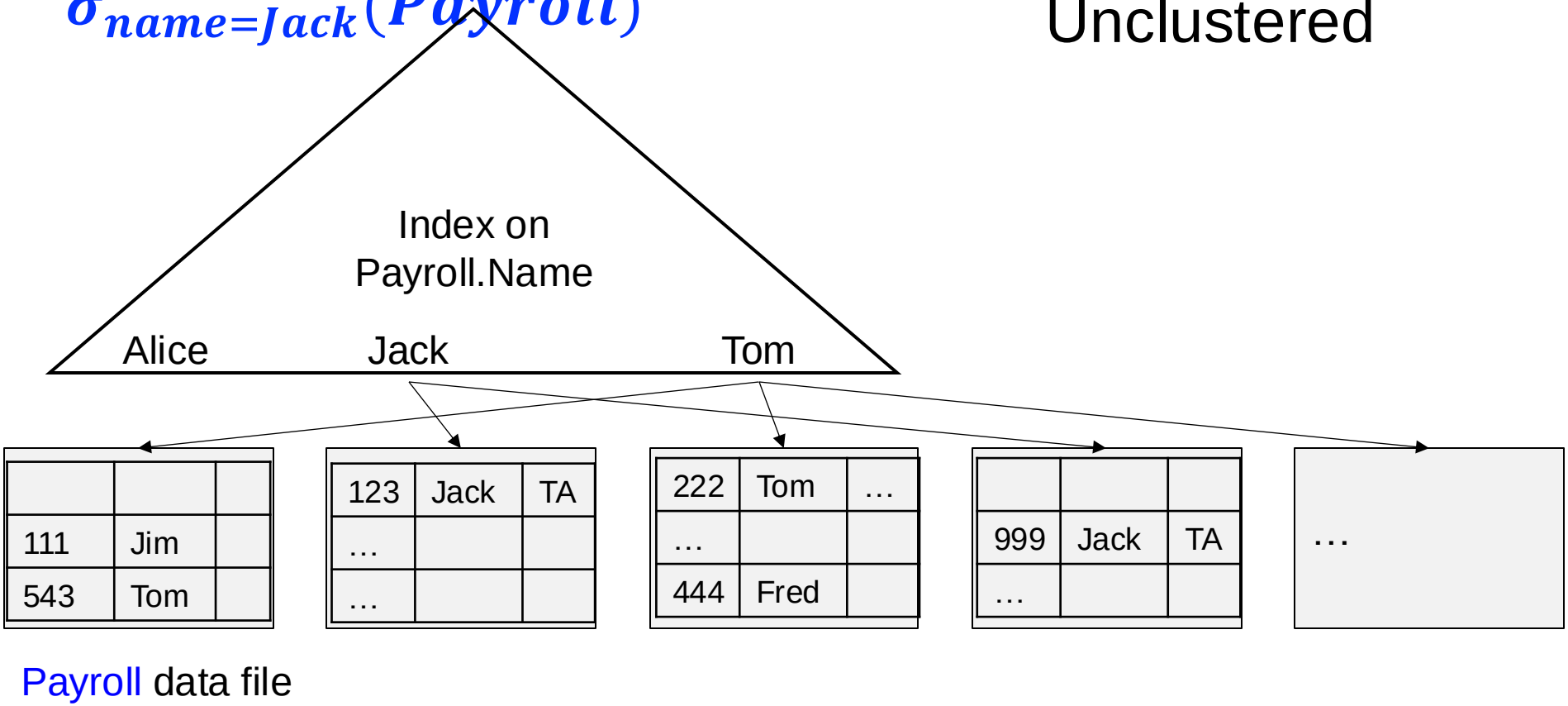
Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$\sigma_{name=Jack}(Payroll)$

Unclustered

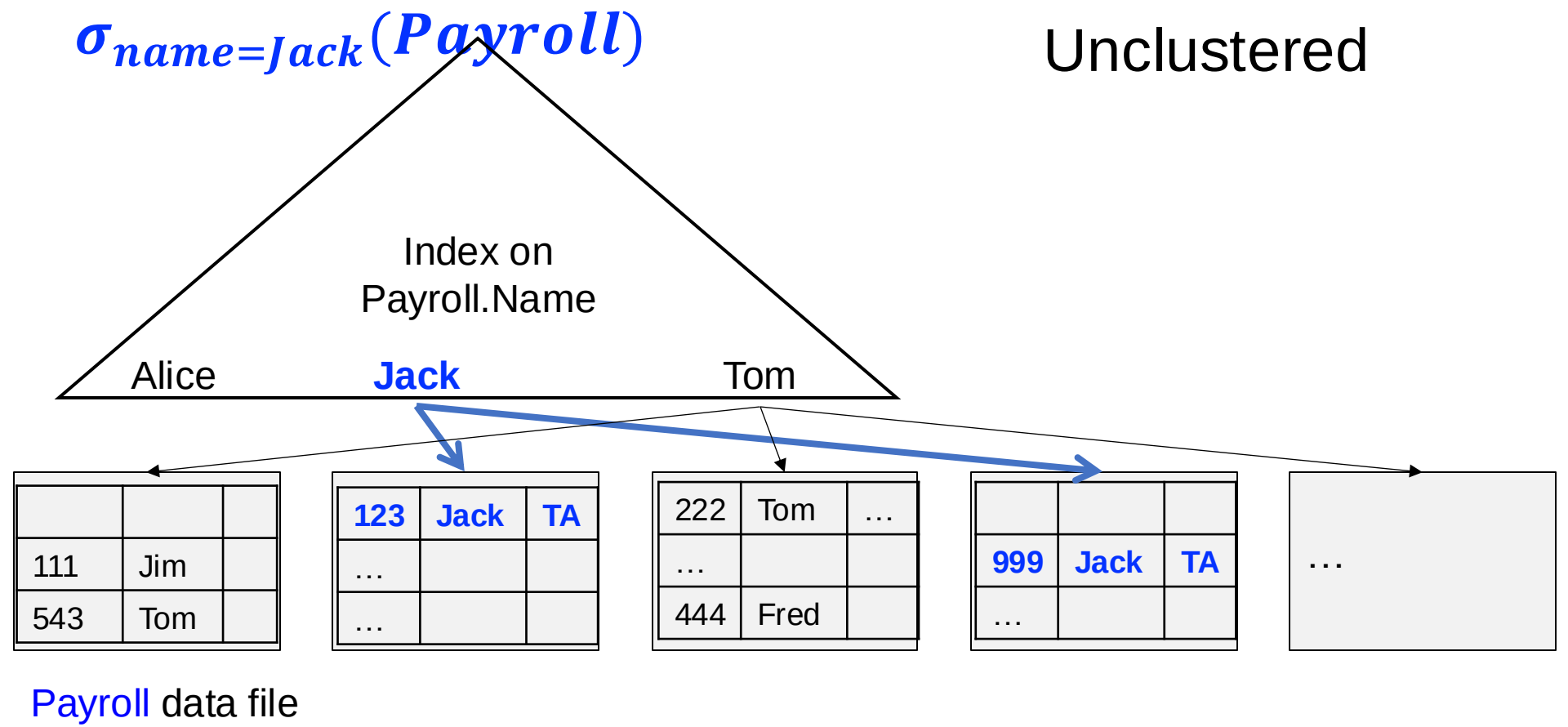


Payroll data file

Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

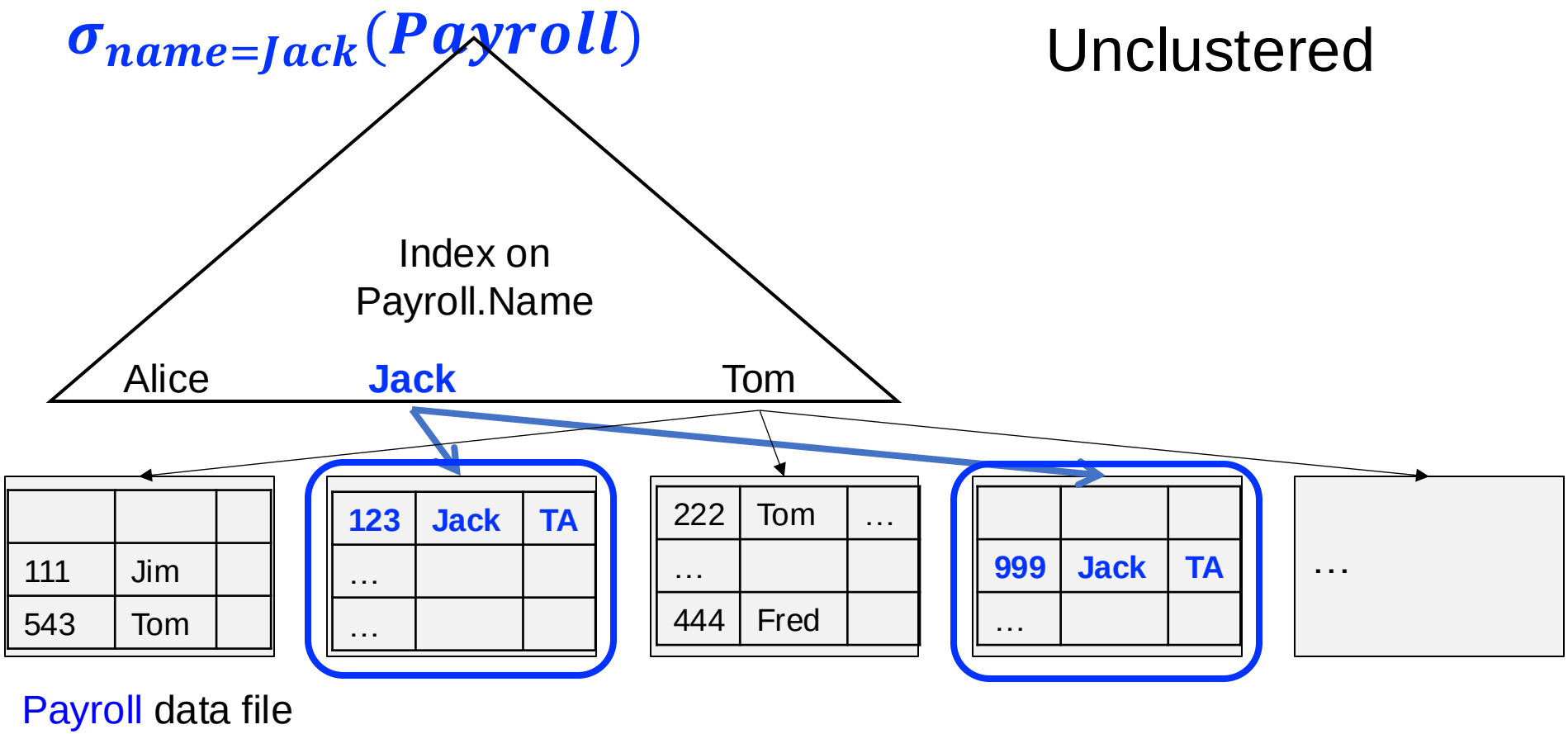
```
CREATE INDEX Pname on Payroll(name)
```



Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```



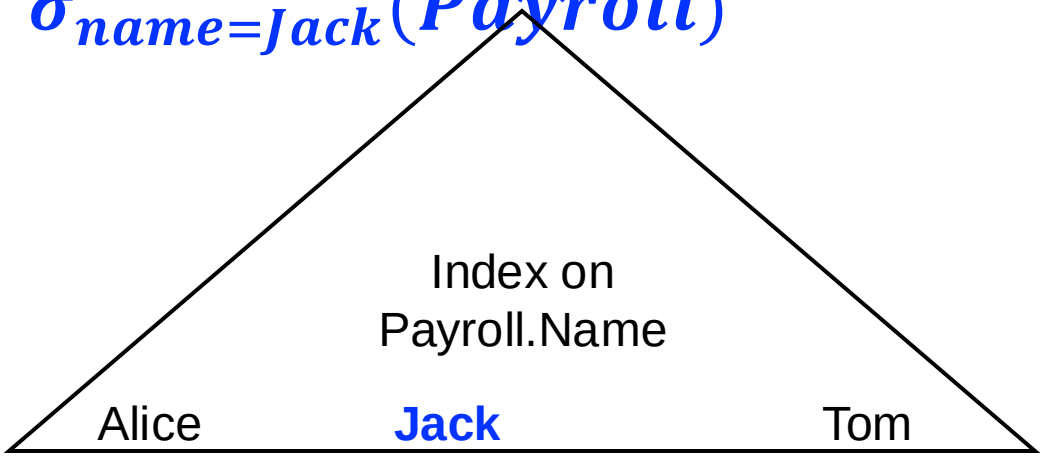
Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

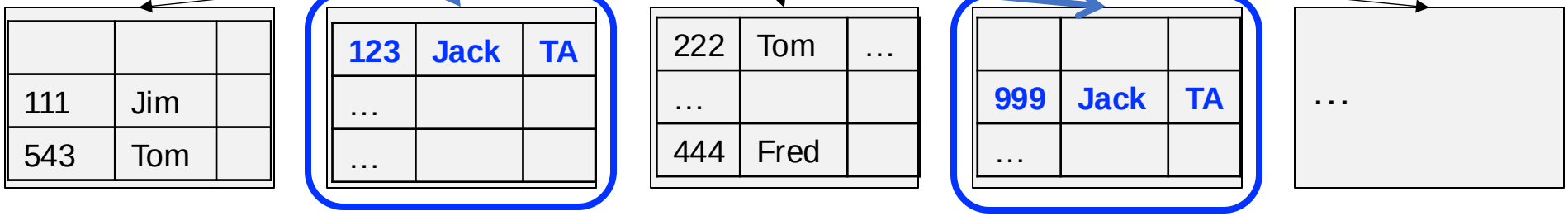
$\sigma_{name=Jack}(Payroll)$

Unclustered



$$\textbf{Cost} = \frac{T(Payroll)}{V(Payroll,Name)}$$

Random access!



Payroll data file

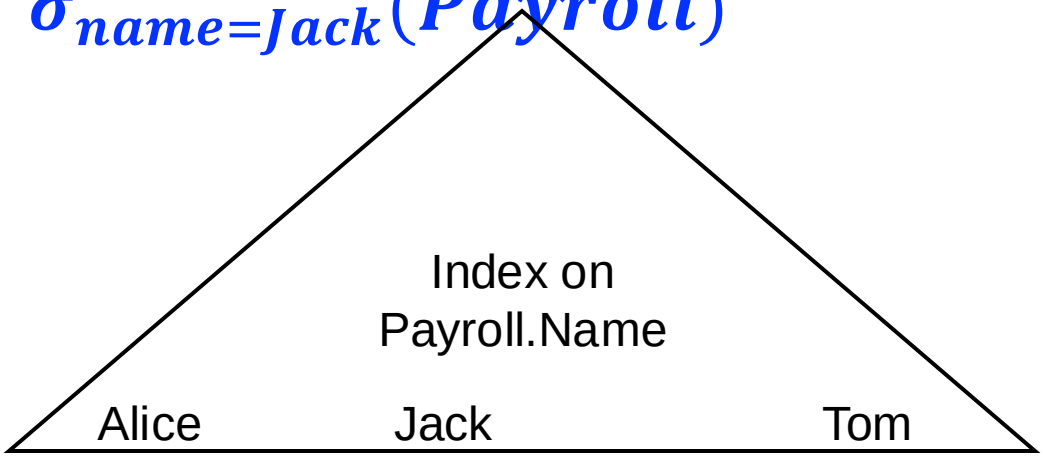
Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$\sigma_{name=Jack}(Payroll)$

Clustered



111	Alice	
…	Alice	

	Fred	
123	Jack	TA
…	Jack	…

222	Jack	…
…		
444	Jack	

…	Jack	
999	Jack	TA
…	Jim	

…

Payroll data file

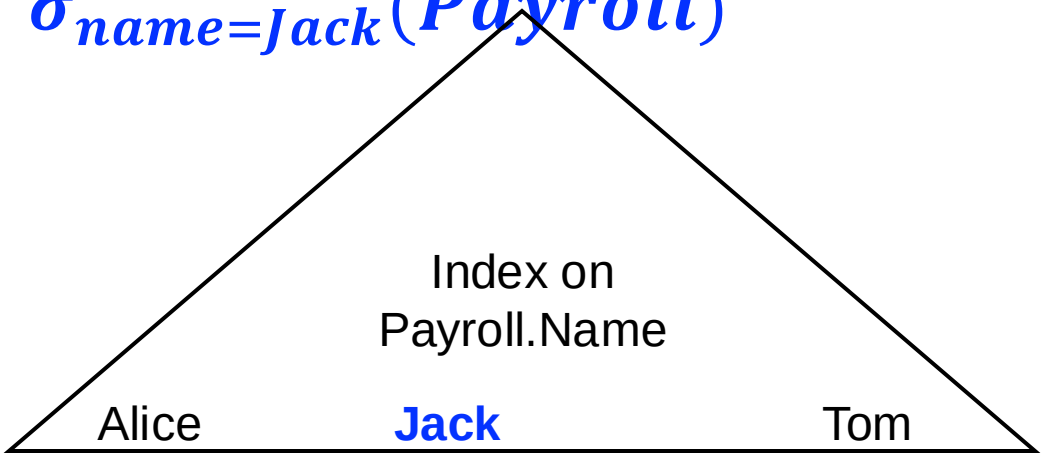
Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$\sigma_{name=Jack}(Payroll)$

Clustered



111	Alice	
…	Alice	

	Fred	
123	Jack	TA
…	Jack	…

222	Jack	…
…		
444	Jack	

…	Jack	
999	Jack	TA
…	Jim	

…

Payroll data file

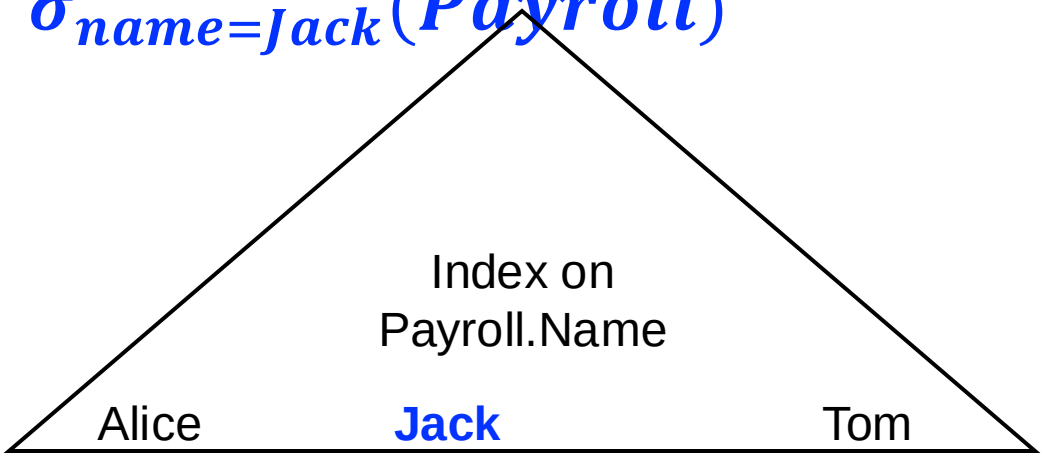
Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$\sigma_{name=Jack}(Payroll)$

Clustered



111	Alice	
…	Alice	

	Fred	
123	Jack	TA
…	Jack	…

222	Jack	…
…		
444	Jack	

…	Jack	
999	Jack	TA
…	Jim	

…

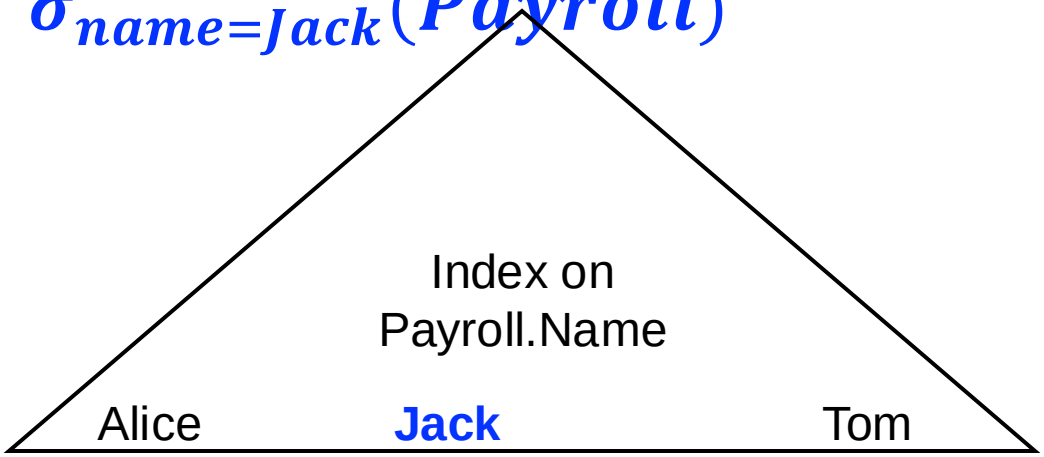
Payroll data file

Index-Based Selection

```
CREATE TABLE Payroll(UserID int, Name text, Job text, Salary int)
```

```
CREATE INDEX Pname on Payroll(name)
```

$\sigma_{name=Jack}(Payroll)$



Clustered:

Cost = $\frac{B(Payroll)}{V(Payroll,Name)}$

Sequential access!

111	Alice	
…	Alice	

	Fred	
123	Jack	TA
…	Jack	…

222	Jack	…
…		
444	Jack	

…	Jack	
999	Jack	TA
…	Jim	

…

Payroll data file

Index-Based Selection

$\sigma_{A=value}(R)$

- Sequential scan: Cost = $B(R)$
- $\text{Cost}_{\text{unclustered}} = \frac{T(R)}{V(R,A)}$ random access
- $\text{Cost}_{\text{clustered}} = \frac{B(R)}{V(R,A)}$ sequential access

Question

How will these be answered (assume unclustered)?

- $\sigma_{Name=Jim}(Payroll)$
- $\sigma_{Job=TA}(Payroll)$
- $\sigma_{Gender=F}(Payroll)$

Question

How will these be answered (assume unclustered)?

- $\sigma_{Name=Jim}(Payroll)$

V(Payroll,Name) is large
Index Lookup

- $\sigma_{Job=TA}(Payroll)$
- $\sigma_{Gender=F}(Payroll)$

Question

How will these be answered (assume unclustered)?

- $\sigma_{Name=Jim}(Payroll)$

V(Payroll,Name) is large
Index Lookup

- $\sigma_{Job=TA}(Payroll)$

- $\sigma_{Gender=F}(Payroll)$

V(Payroll,Gender) is small
Sequential Scan

Question

How will these be answered (assume unclustered)?

- $\sigma_{Name=Jim}(Payroll)$

V(Payroll,Name) is large
Index Lookup

- $\sigma_{Job=TA}(Payroll)$

Depends on V(Payroll,Job)

- $\sigma_{Gender=F}(Payroll)$

V(Payroll,Gender) is small
Sequential Scan

Index Join

Index Join

$\text{Payroll} \bowtie_{\text{UserID=UserID}} \text{Regist}$

```
Payroll(UserID,Name,Job,Salary)
Regist(UserID,Car)
```

```
for x in Payroll
  for y in IndexLookup(Regist,UserID=x.UserID)
      output(x,y)
```

$$\text{Cost}_{\text{unclustered}} = \frac{T(Payroll) \times T(Regist)}{V(Regist, UserID)}$$

$$\text{Cost}_{\text{clustered}} = \frac{T(Payroll) \times B(Regist)}{V(Regist, UserID)}$$

Index Join

$R \bowtie_{A=B} S$

- Assume index on S.B
- $\text{Cost}_{\text{unclustered}} = \frac{T(R)T(S)}{V(R,A)}$
- $\text{Cost}_{\text{clustered}} = \frac{T(R)B(S)}{V(R,A)}$

- Works very well when R is small, e.g.

```
SELECT *
FROM Payroll x, Regist y
WHERE x.Salary > 10000000 and x.UserId=y.UserID
```

Block Nested Loop Join

Nested Loop Joins

$R \bowtie S$

```
for x in R do
   for y in S do
      if join(x,y): output(x,y)
```

- Naïve nested loop join: B(R) + T(R) * B(S)

- If T(R)=1,000,000 then this is terrible…

Block Nested Loop Join

Idea: better use of the available memory

- M = # of blocks that fit in main memory

Block Nested Loop Join

Group of (M-2) pages of R, called a “block”

```
for each (M-2) pages PR of R do
    for each page PS of S do
      Main memory join: PR ⋈ PS
```

Block Nested Loop Join

Group of (M-2) pages of R, called a “block”

```
for each (M-2) pages PR of R do
    for each page PS of S do
      Main memory join: PR ⋈ PS
```

Why not use M-1 pages?

Block Nested Loop Join

Group of (M-2) pages of R, called a “block”

```
for each (M-2) pages PR of R do
    for each page PS of S do
      Main memory join: PR ⋈ PS
      use the remaining page for output
```

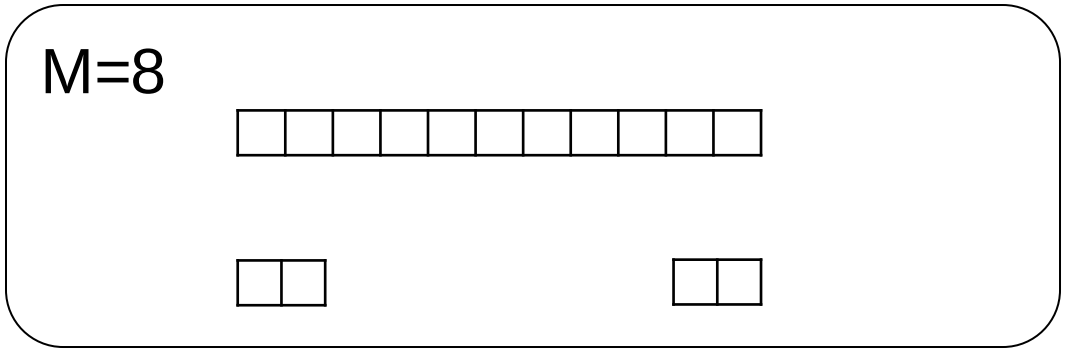
Block Nested Loop Join

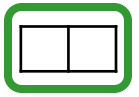
Group of (M-2) pages of R, called a “block”

```
for each (M-2) pages PR of R do
    for each page PS of S do
      Main memory join: PR ⋈ PS
      use the remaining page for output
```

B(R) + B(R)B(S)/(M-2) disk I/Os. WHY?

Block Nested Loop Join

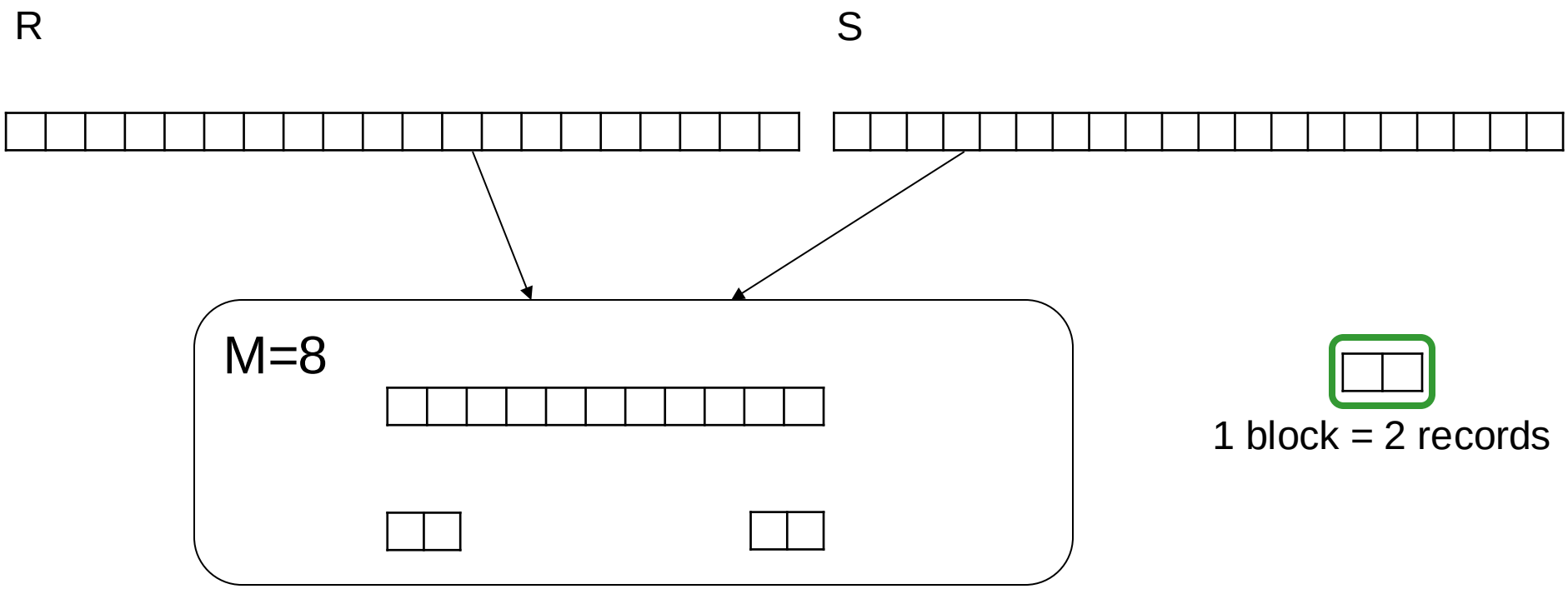




1 block = 2 records

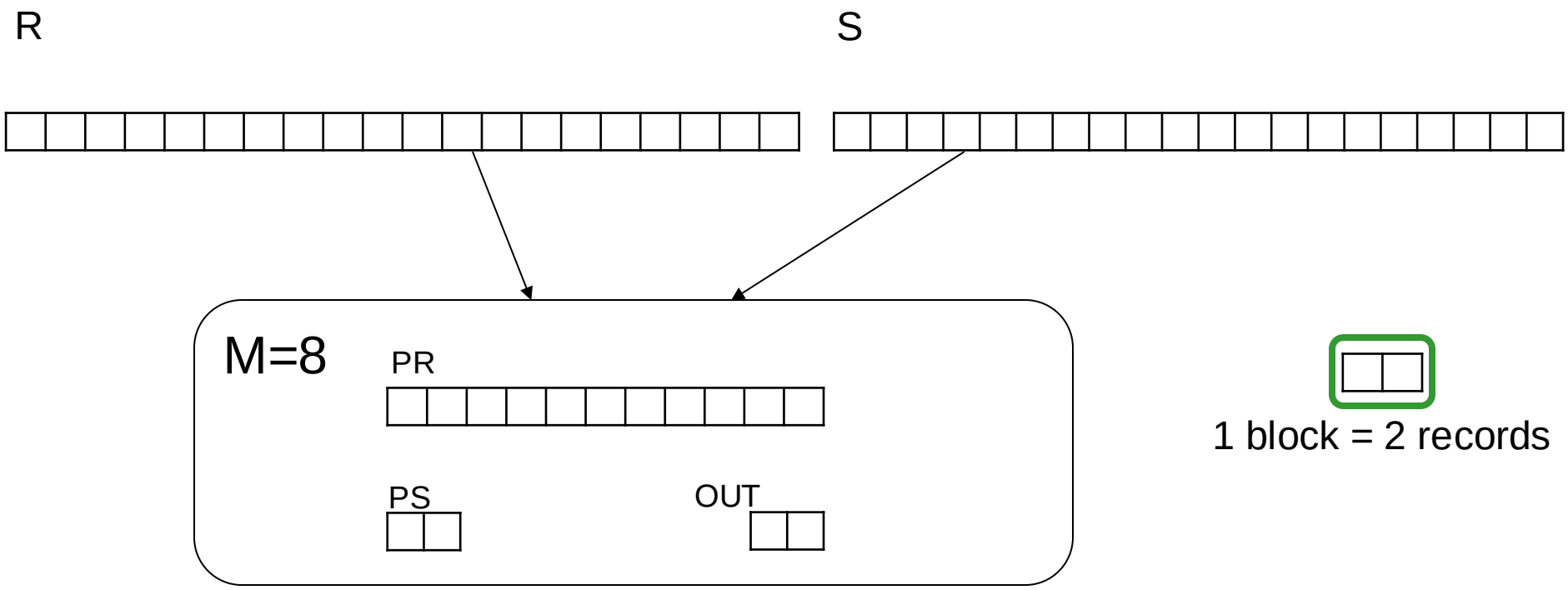
$R \bowtie S$

Block Nested Loop Join



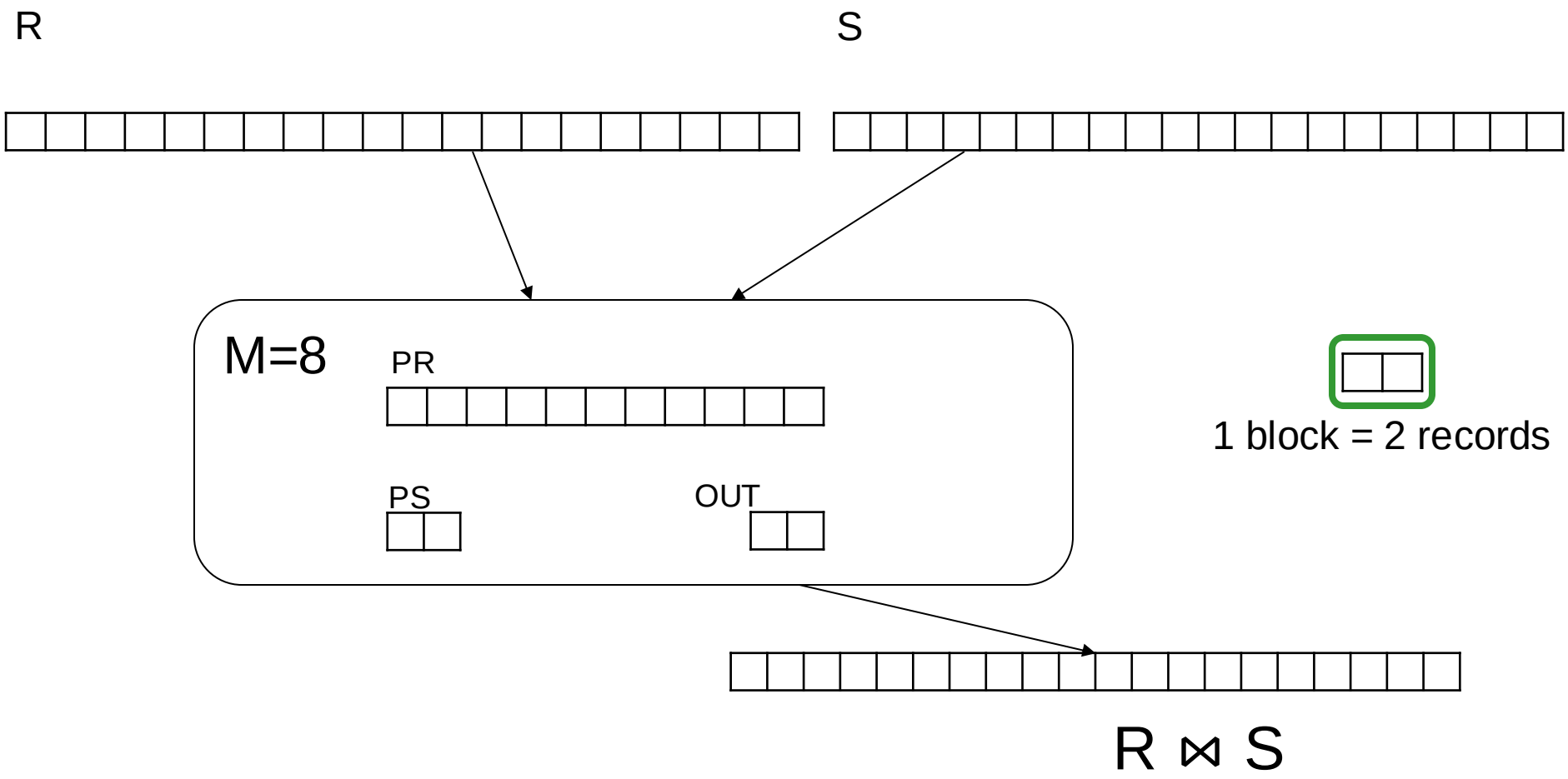
$R \bowtie S$

Block Nested Loop Join

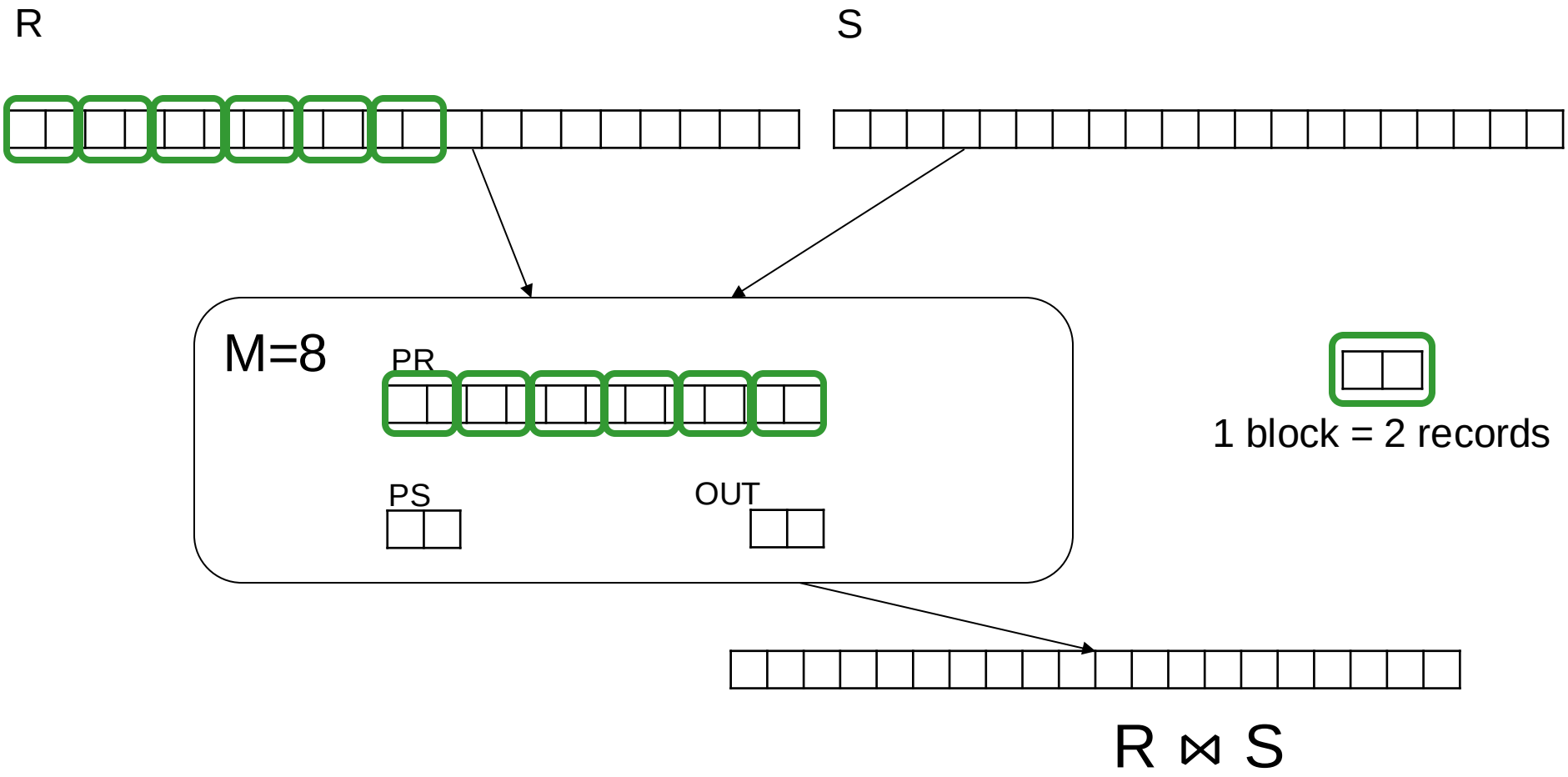


$R \bowtie S$

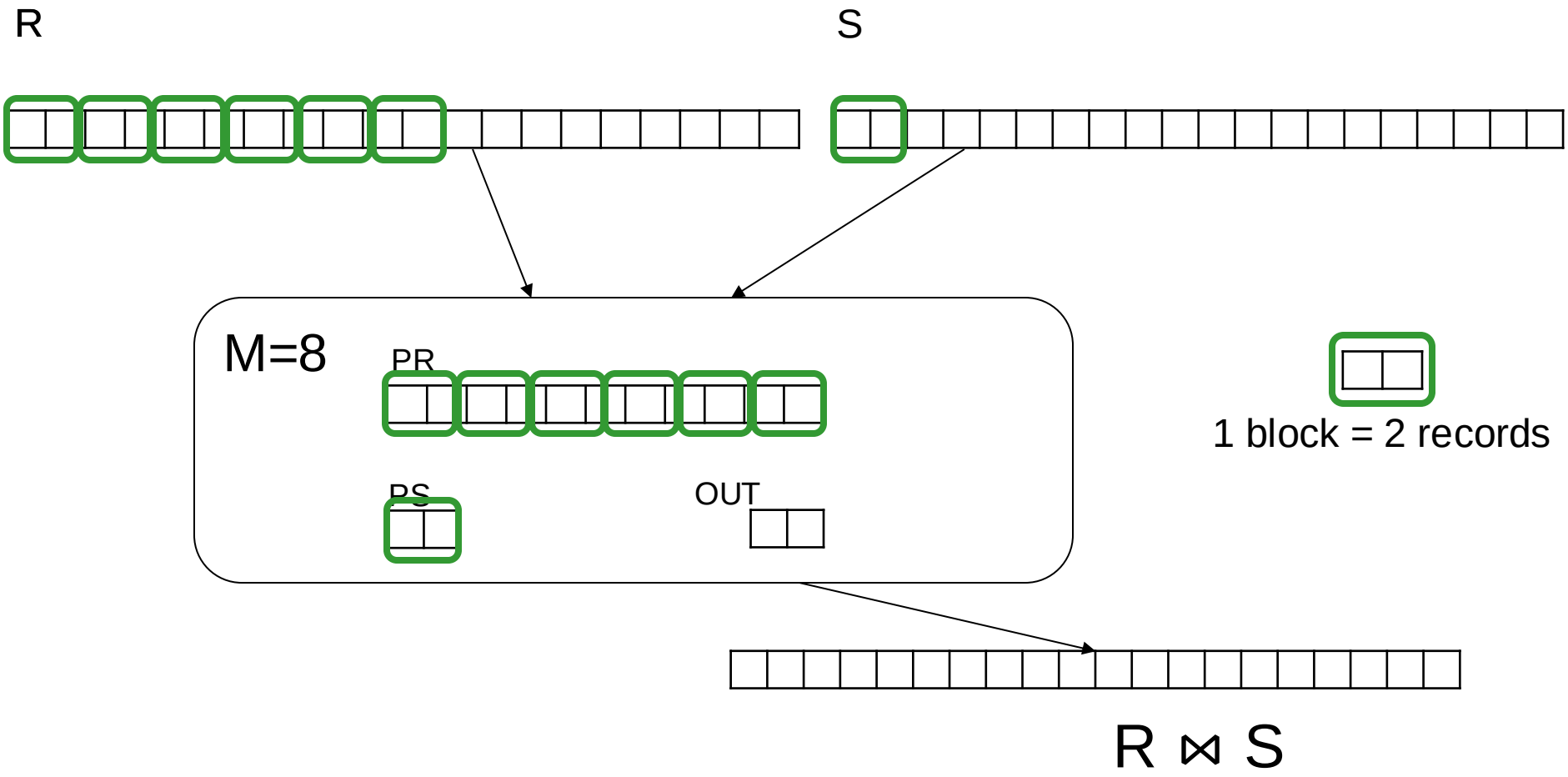
Block Nested Loop Join



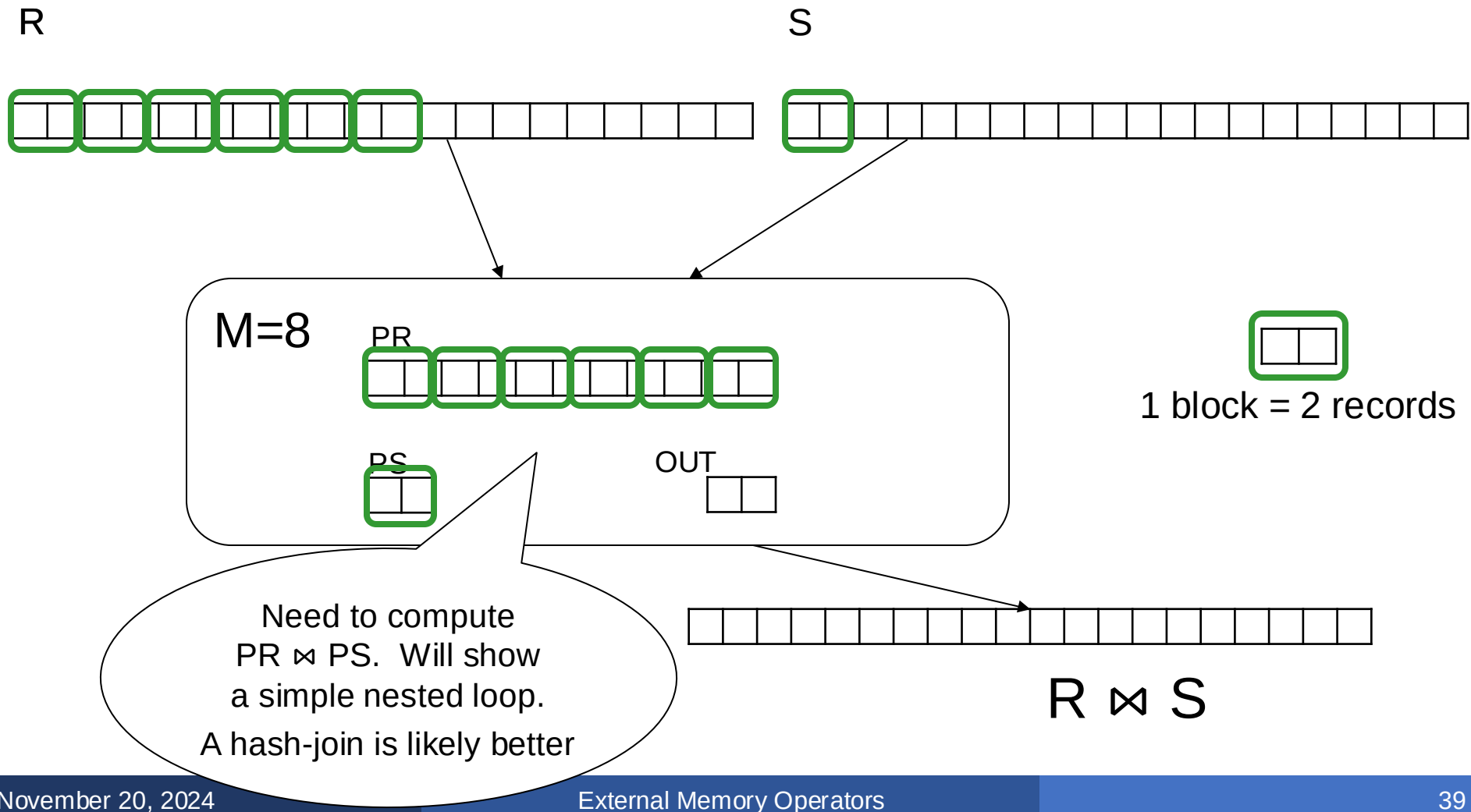
Block Nested Loop Join



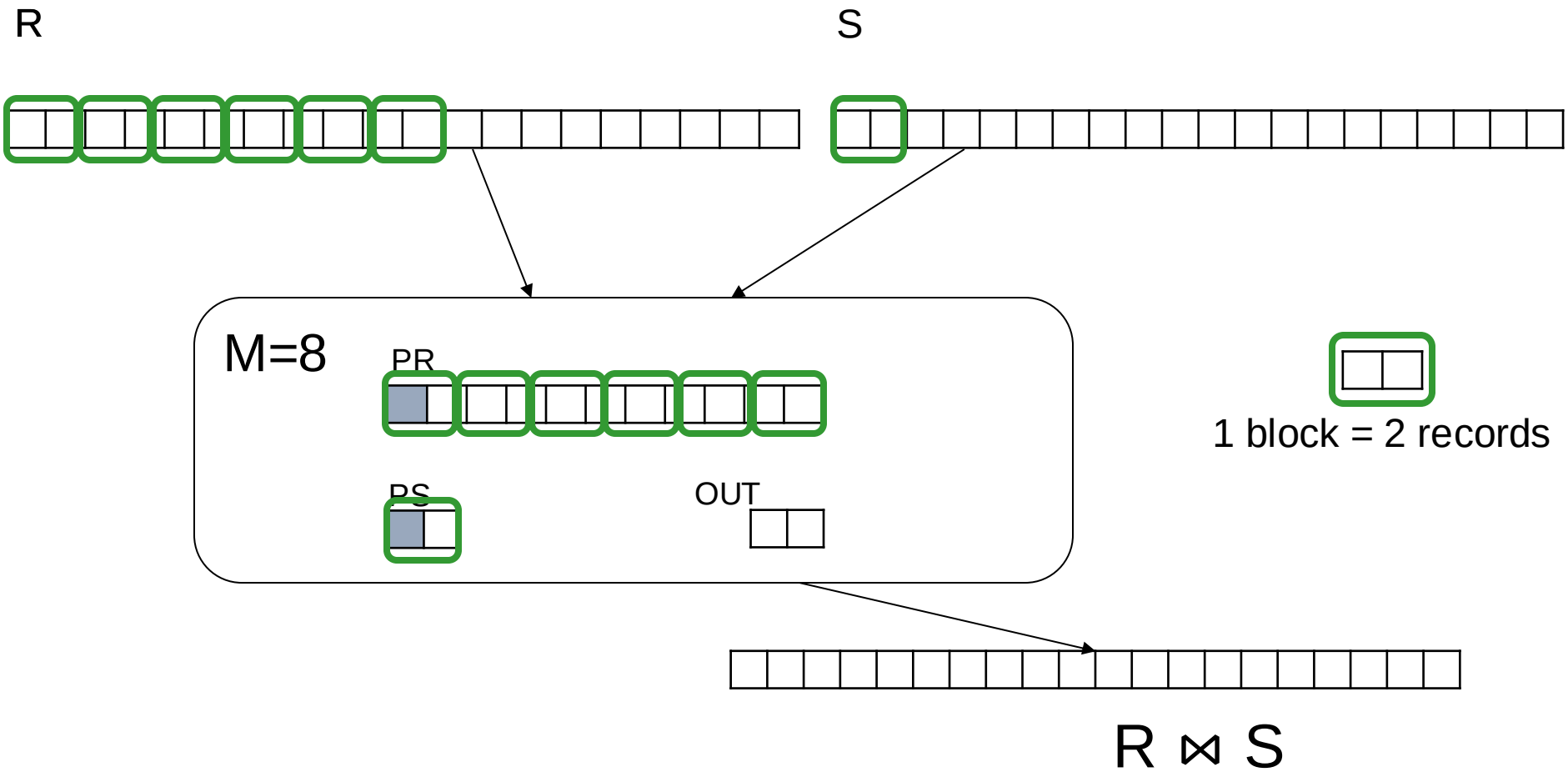
Block Nested Loop Join



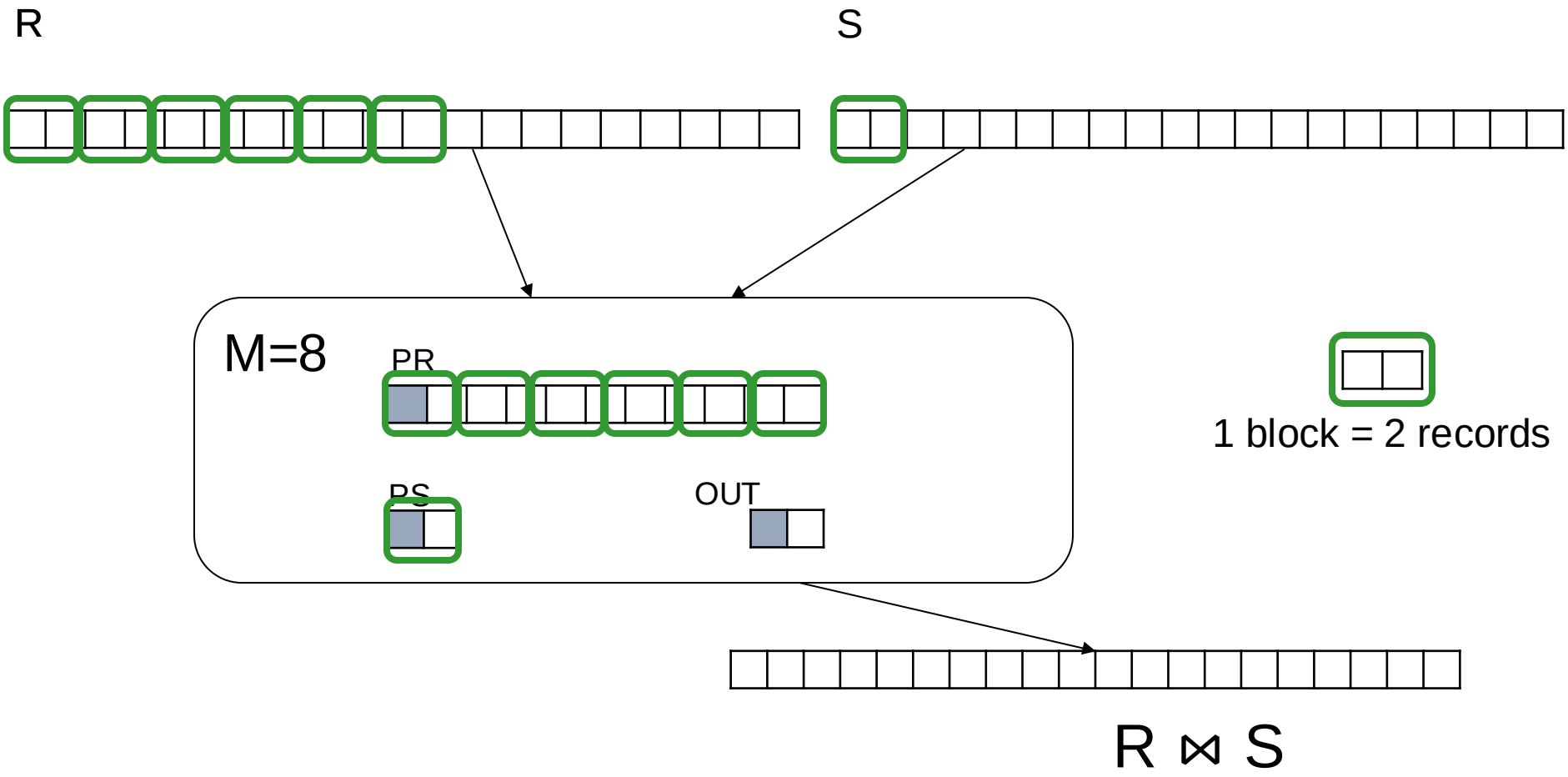
Block Nested Loop Join



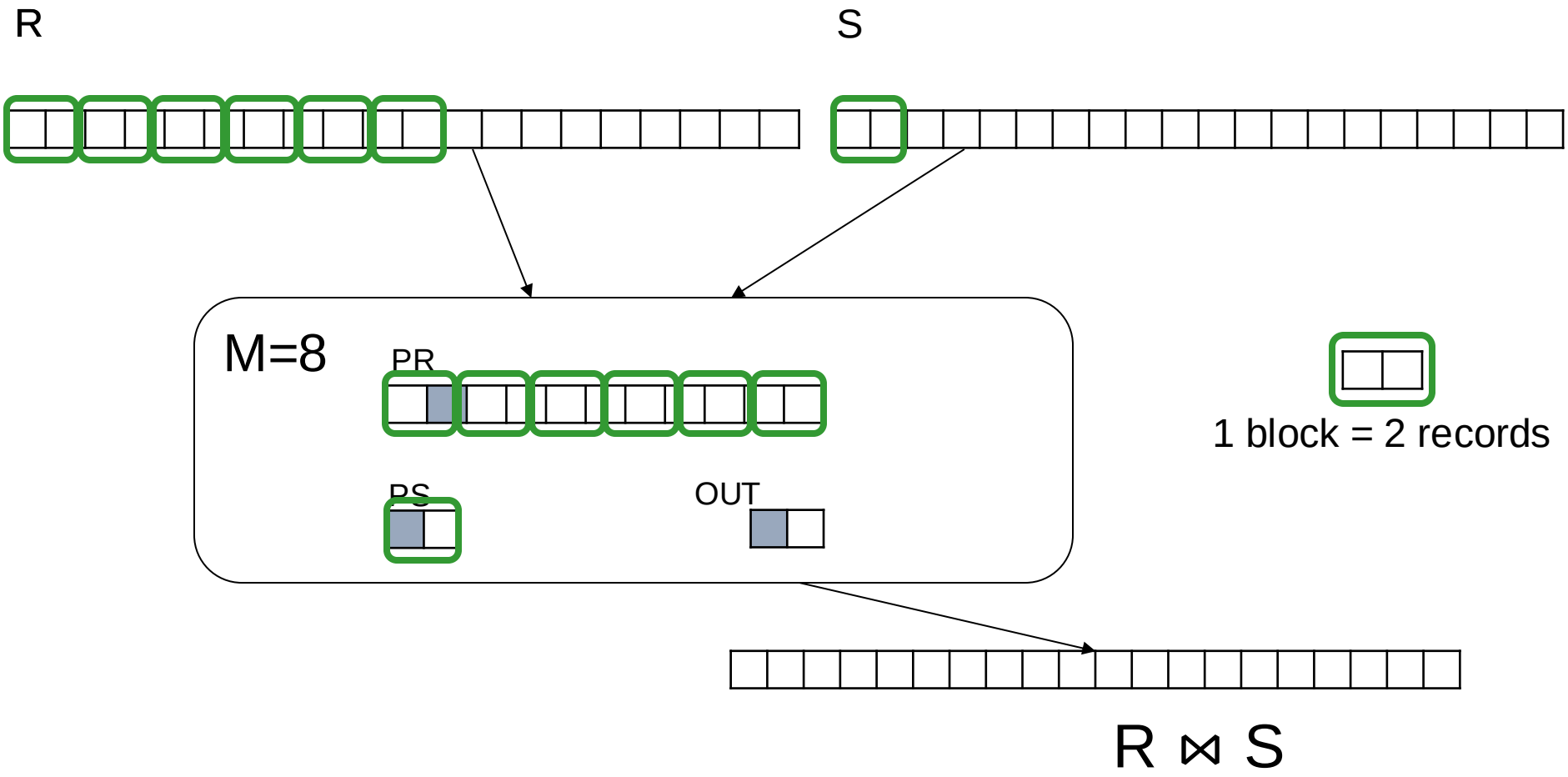
Block Nested Loop Join



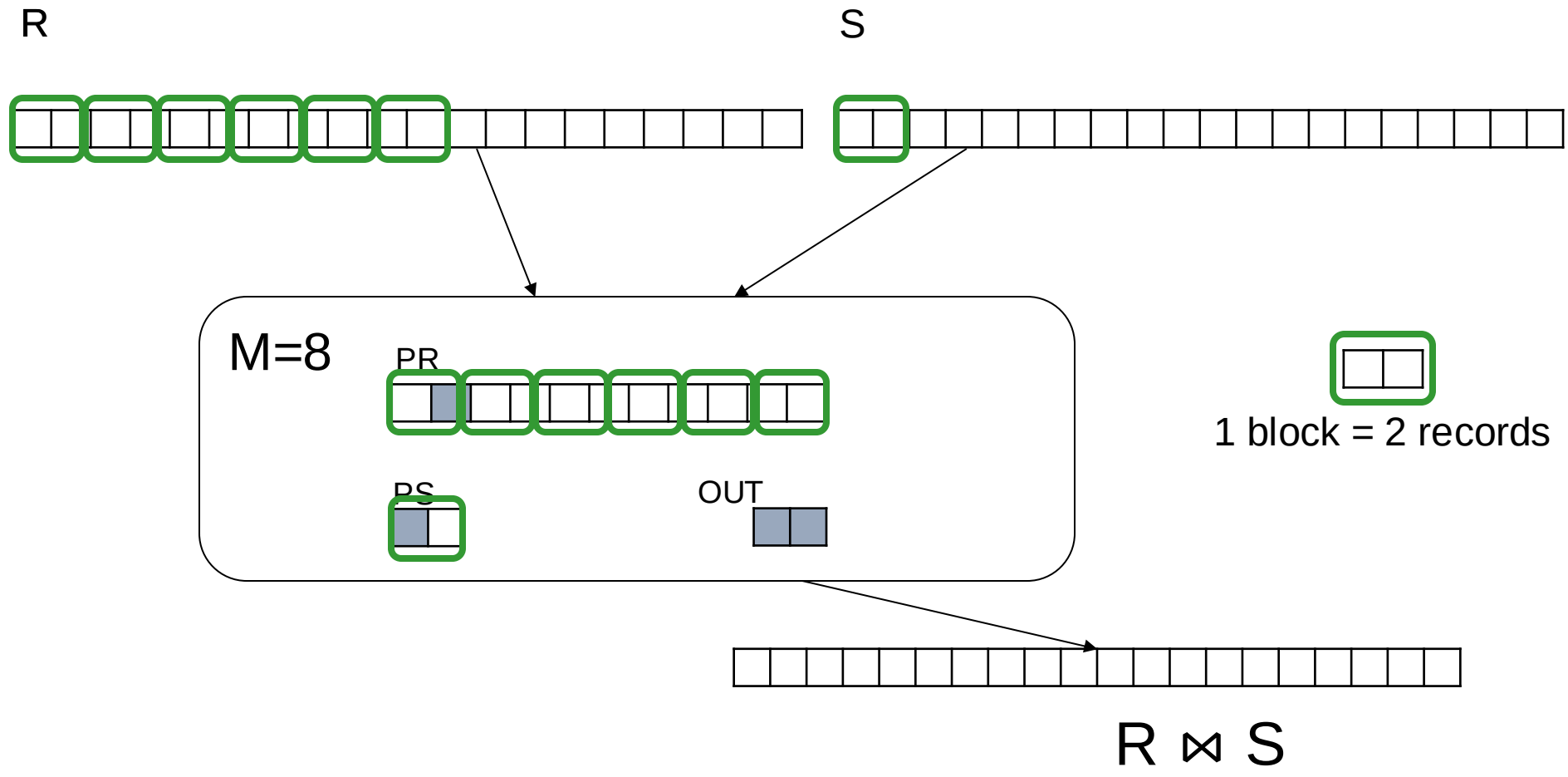
Block Nested Loop Join



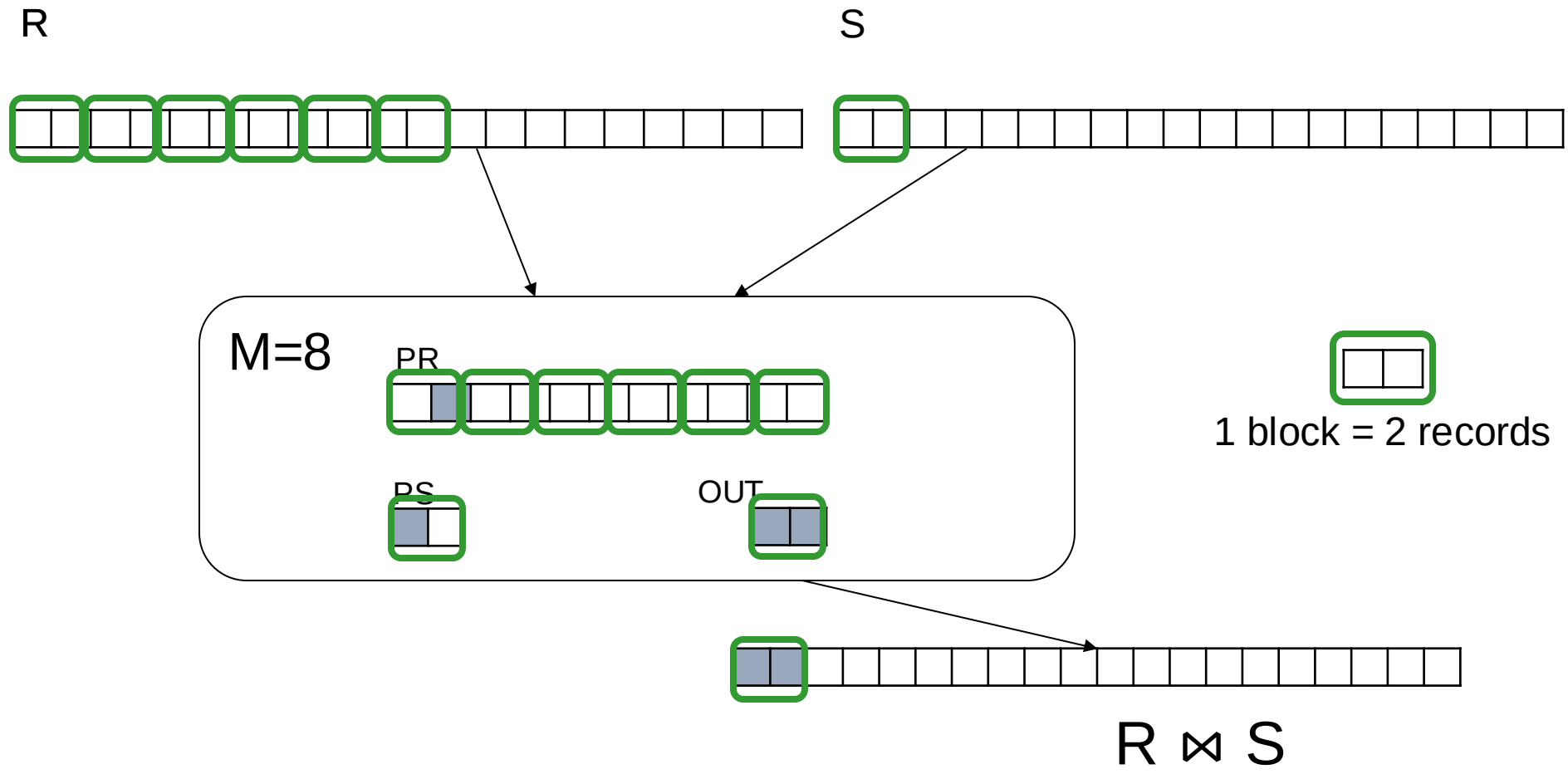
Block Nested Loop Join



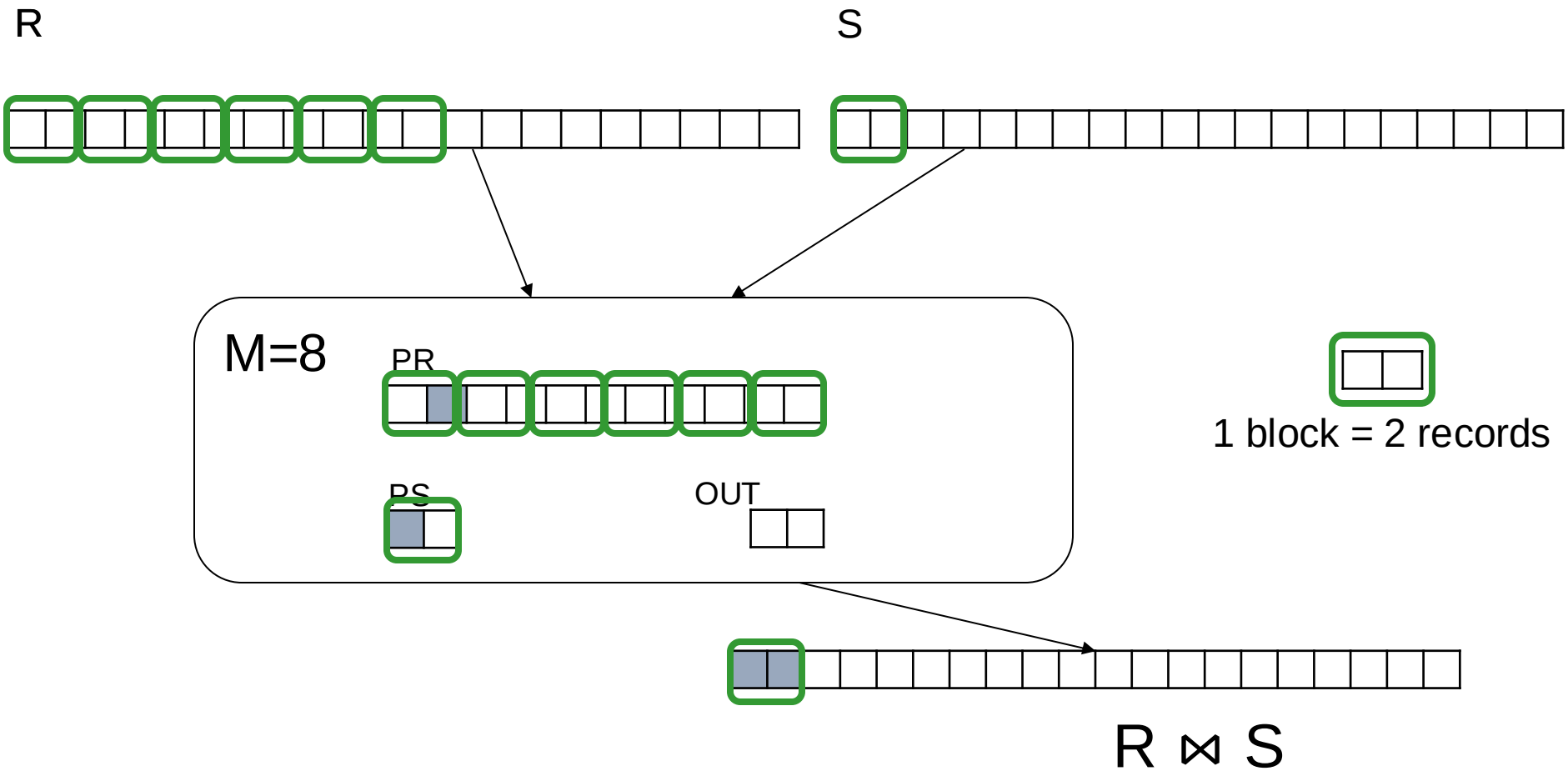
Block Nested Loop Join



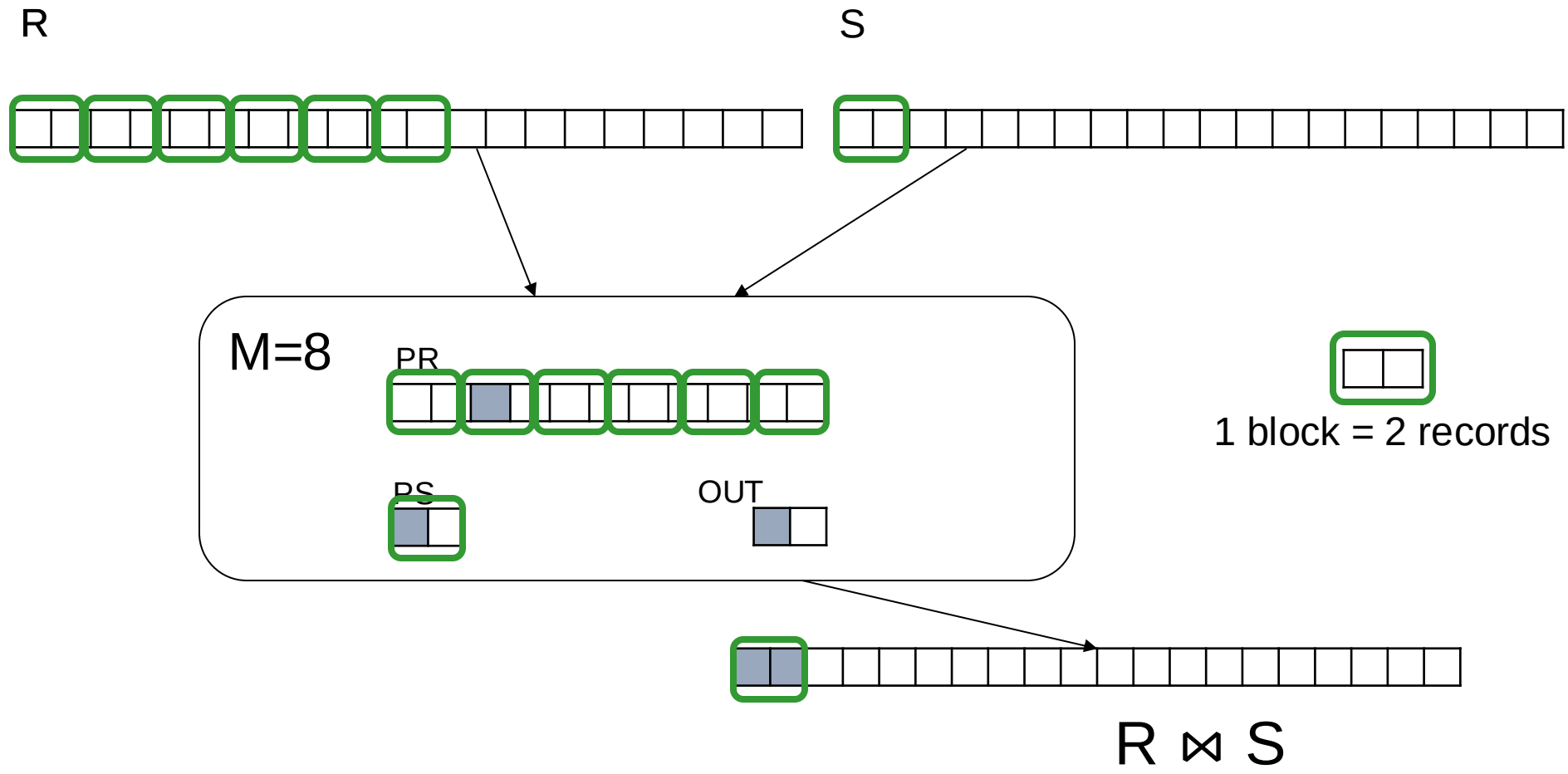
Block Nested Loop Join



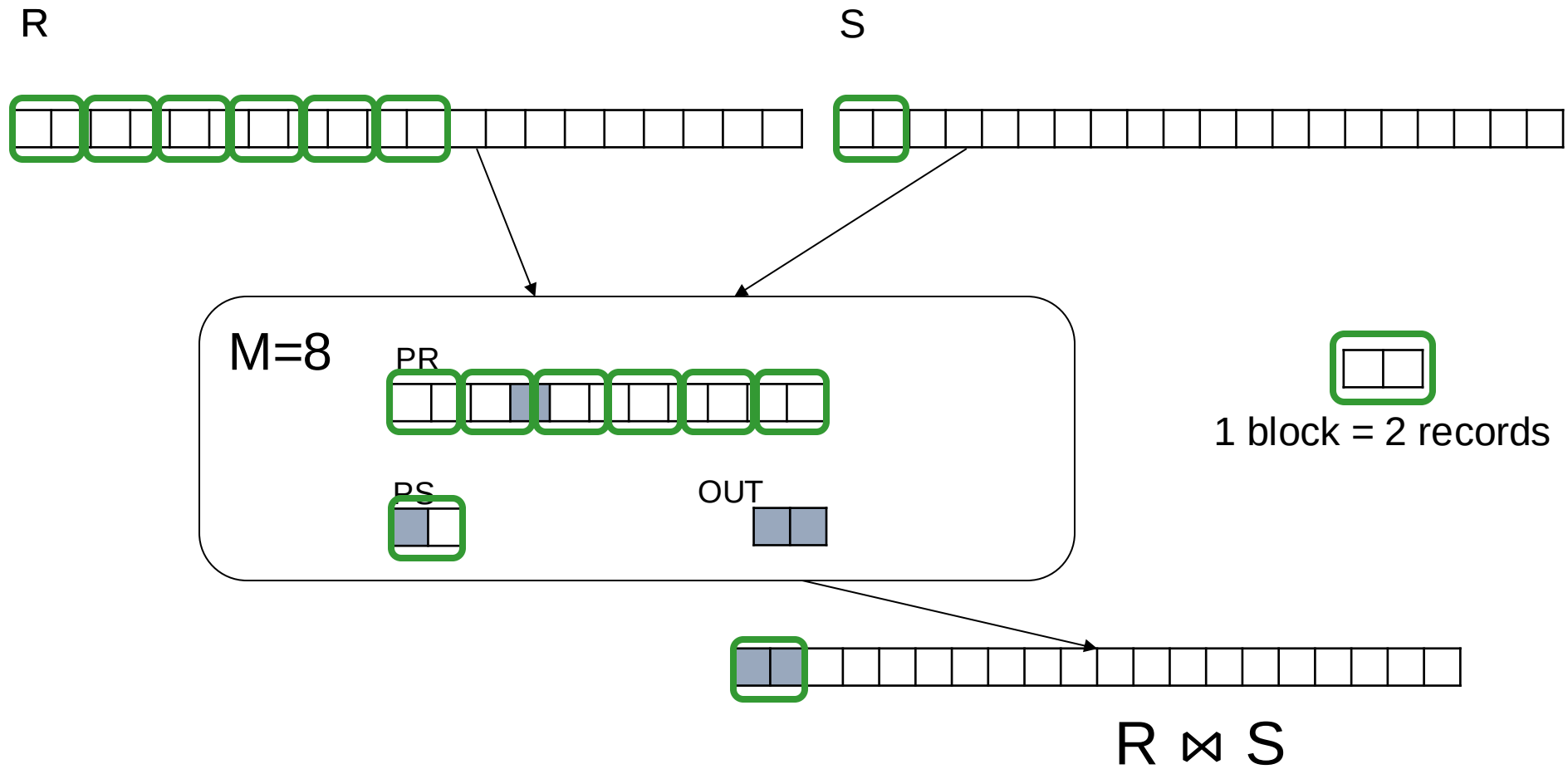
Block Nested Loop Join



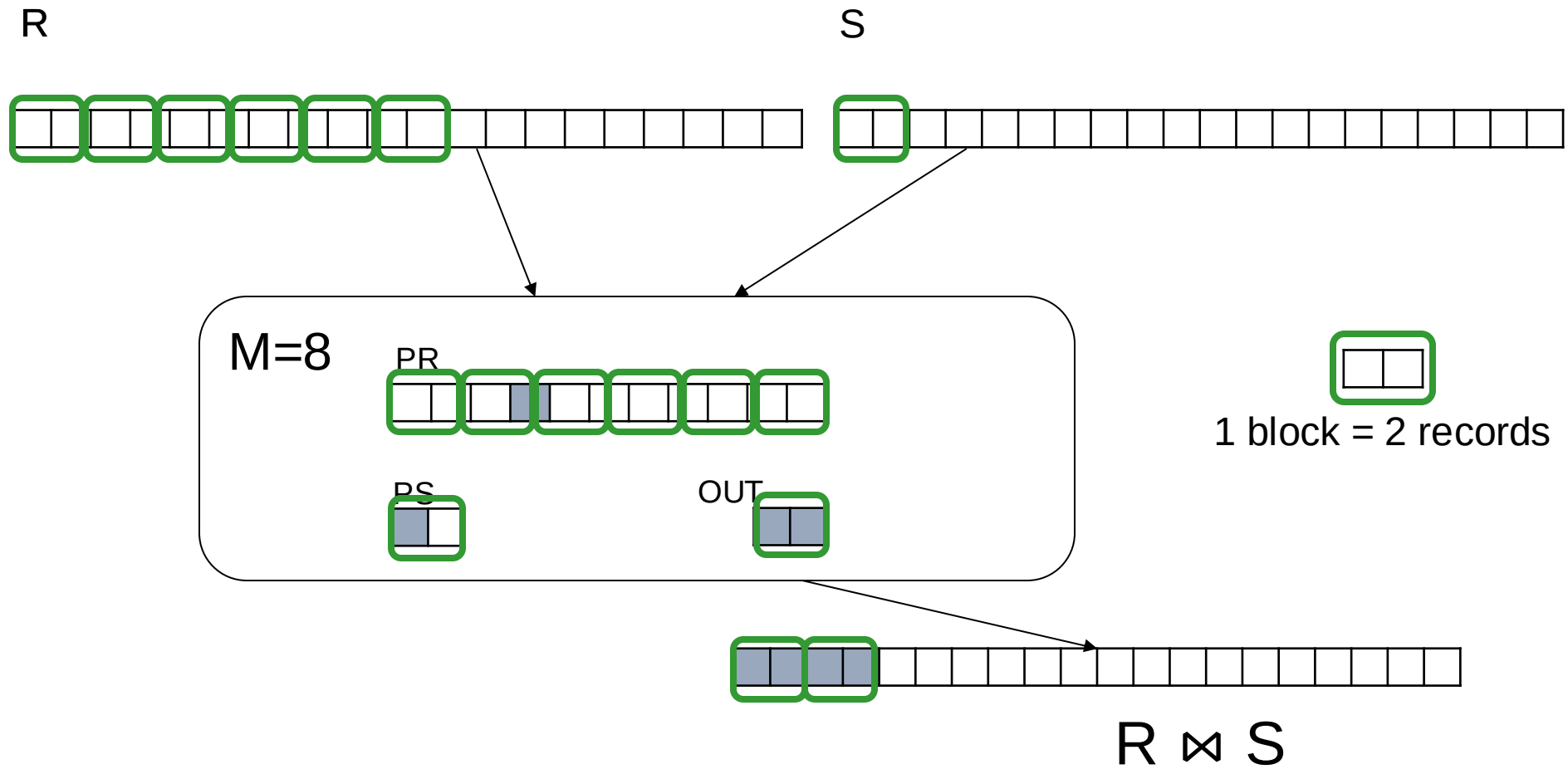
Block Nested Loop Join



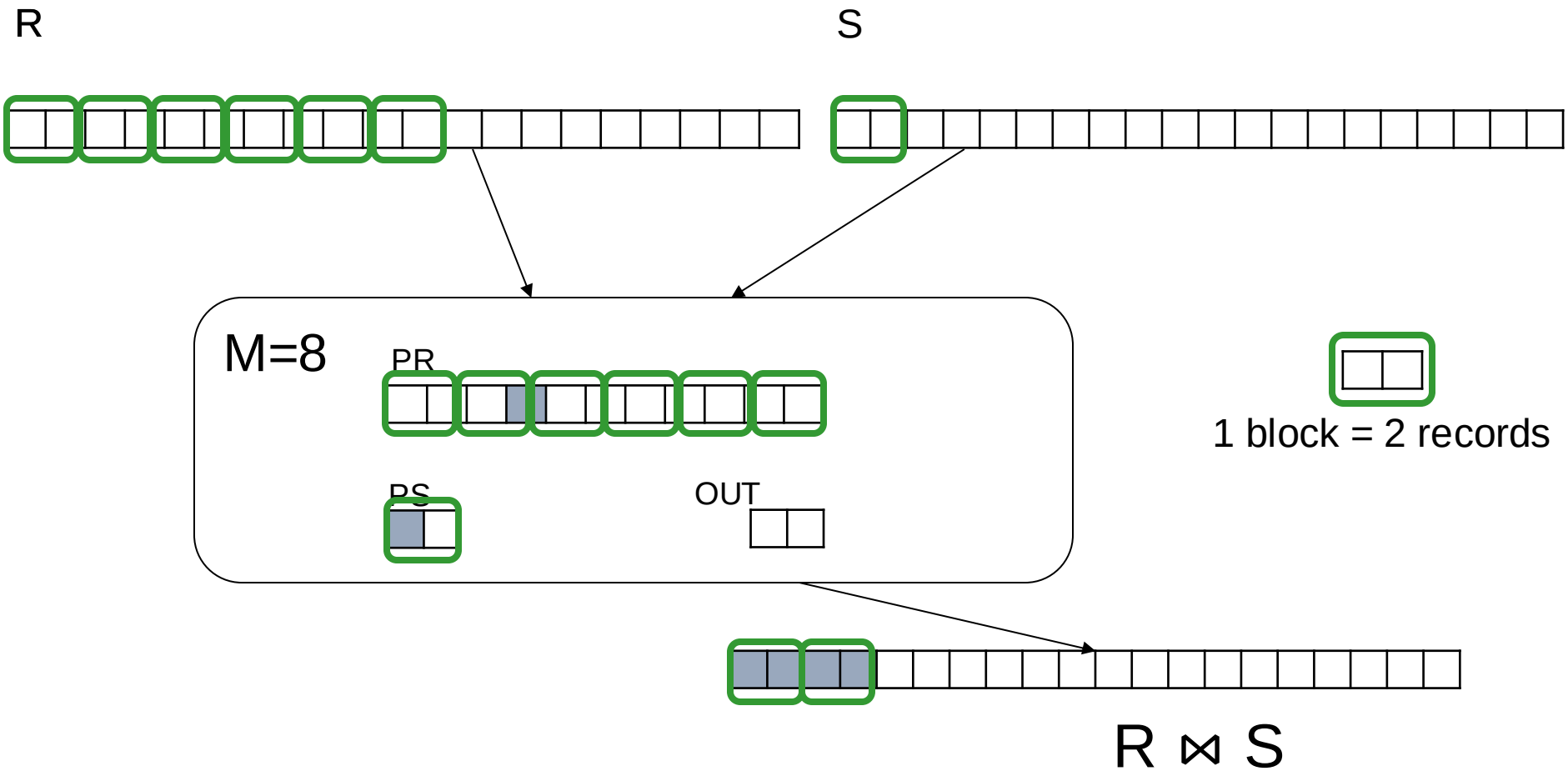
Block Nested Loop Join



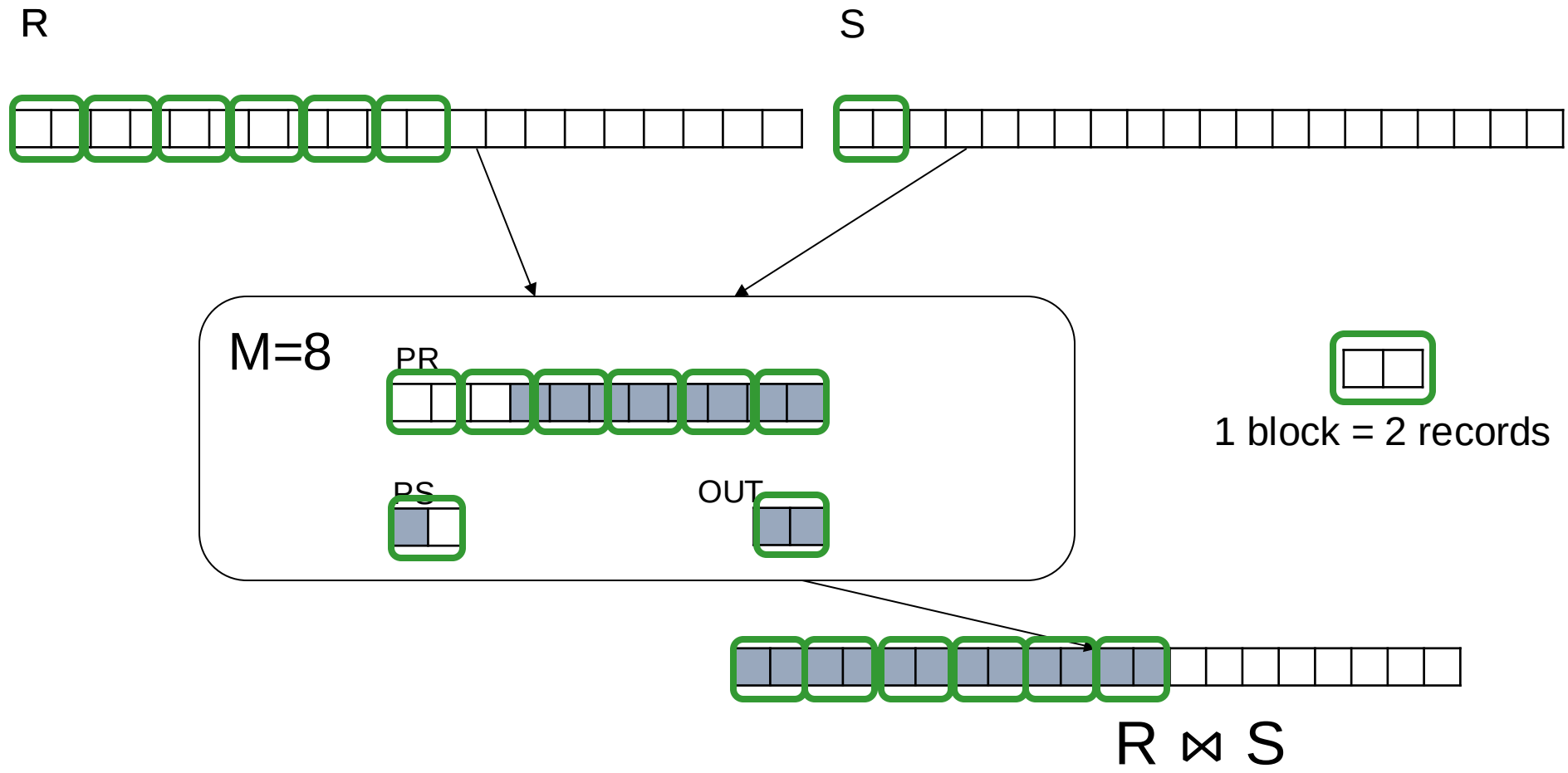
Block Nested Loop Join



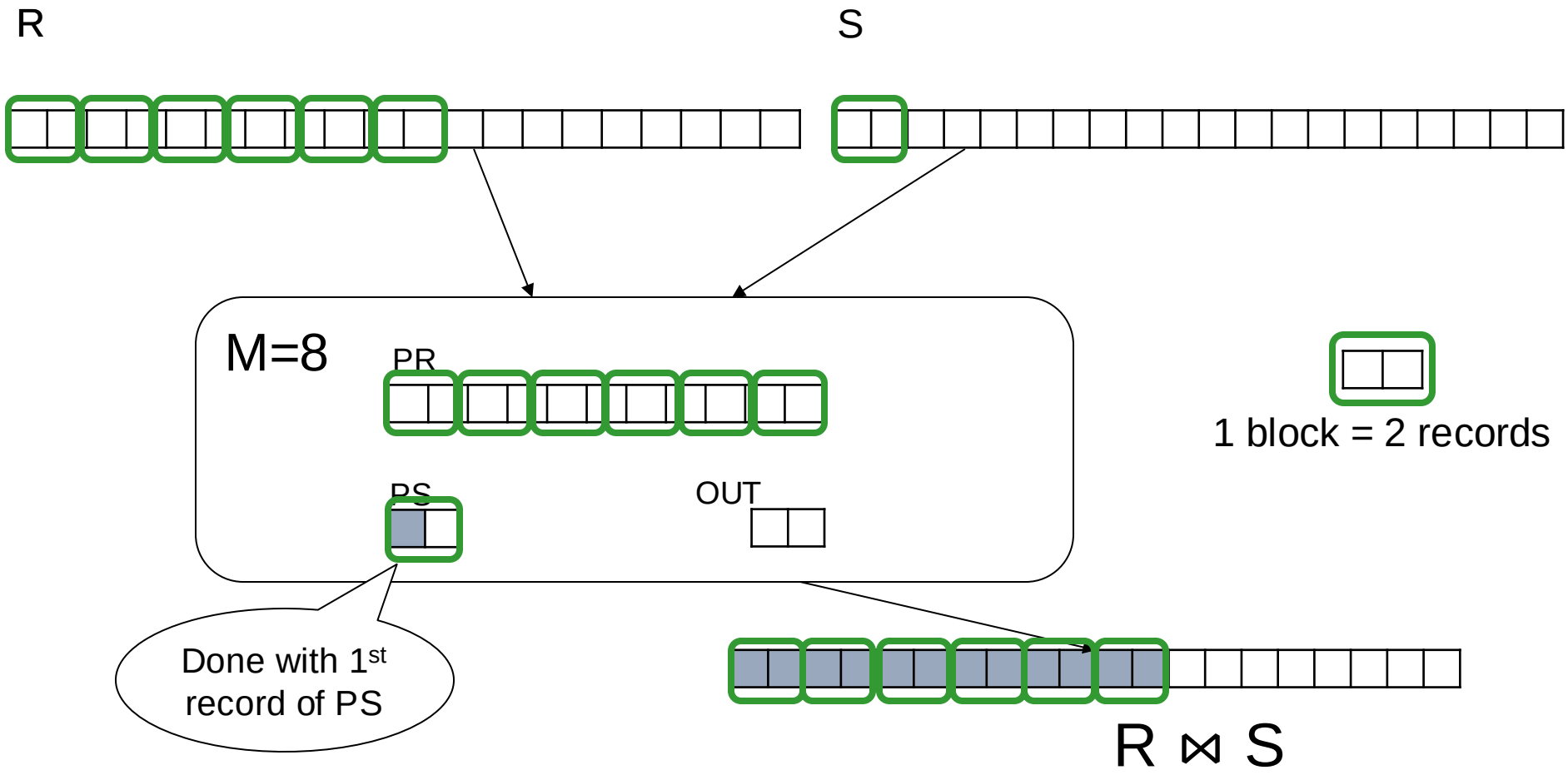
Block Nested Loop Join



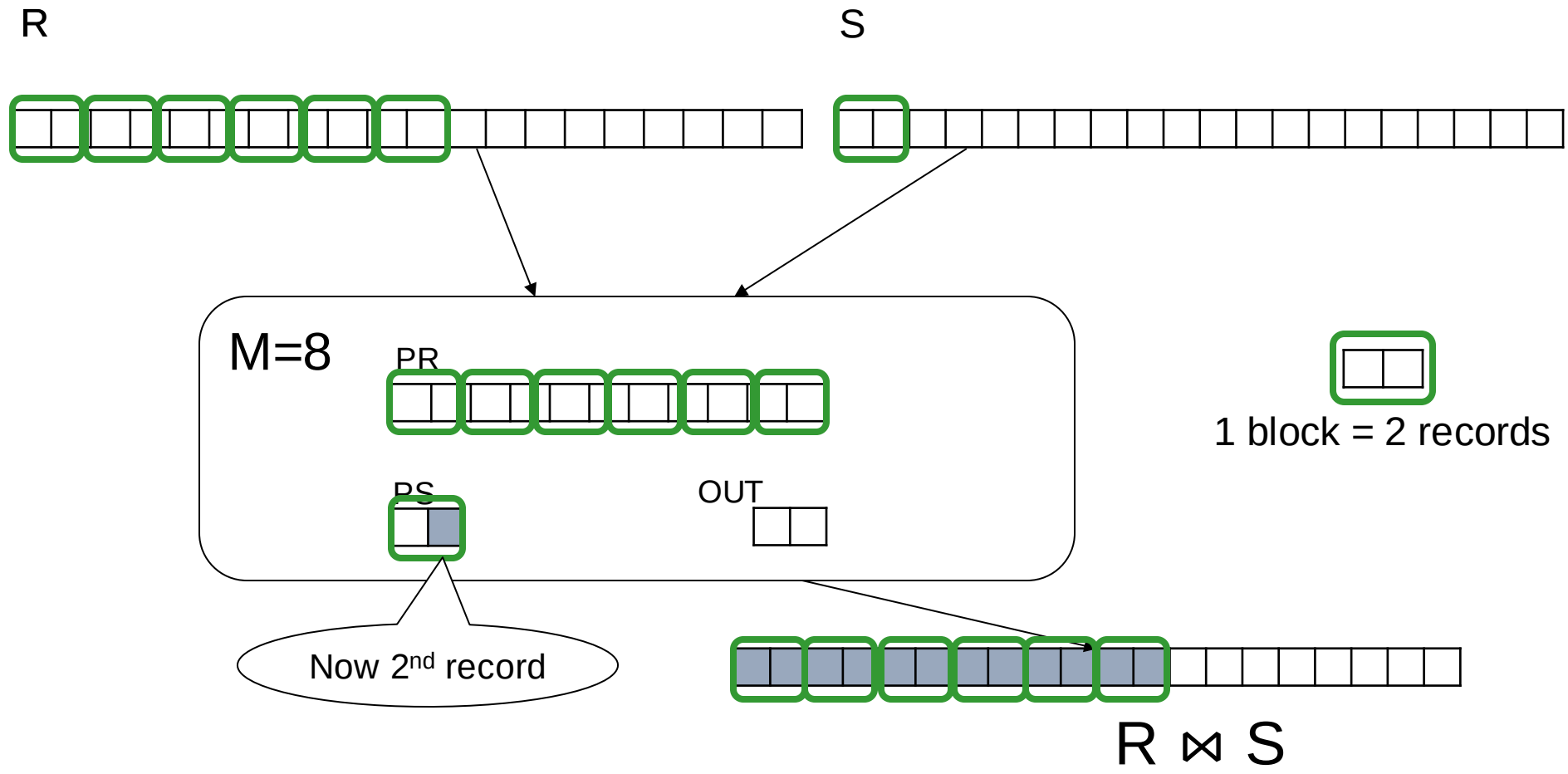
Block Nested Loop Join



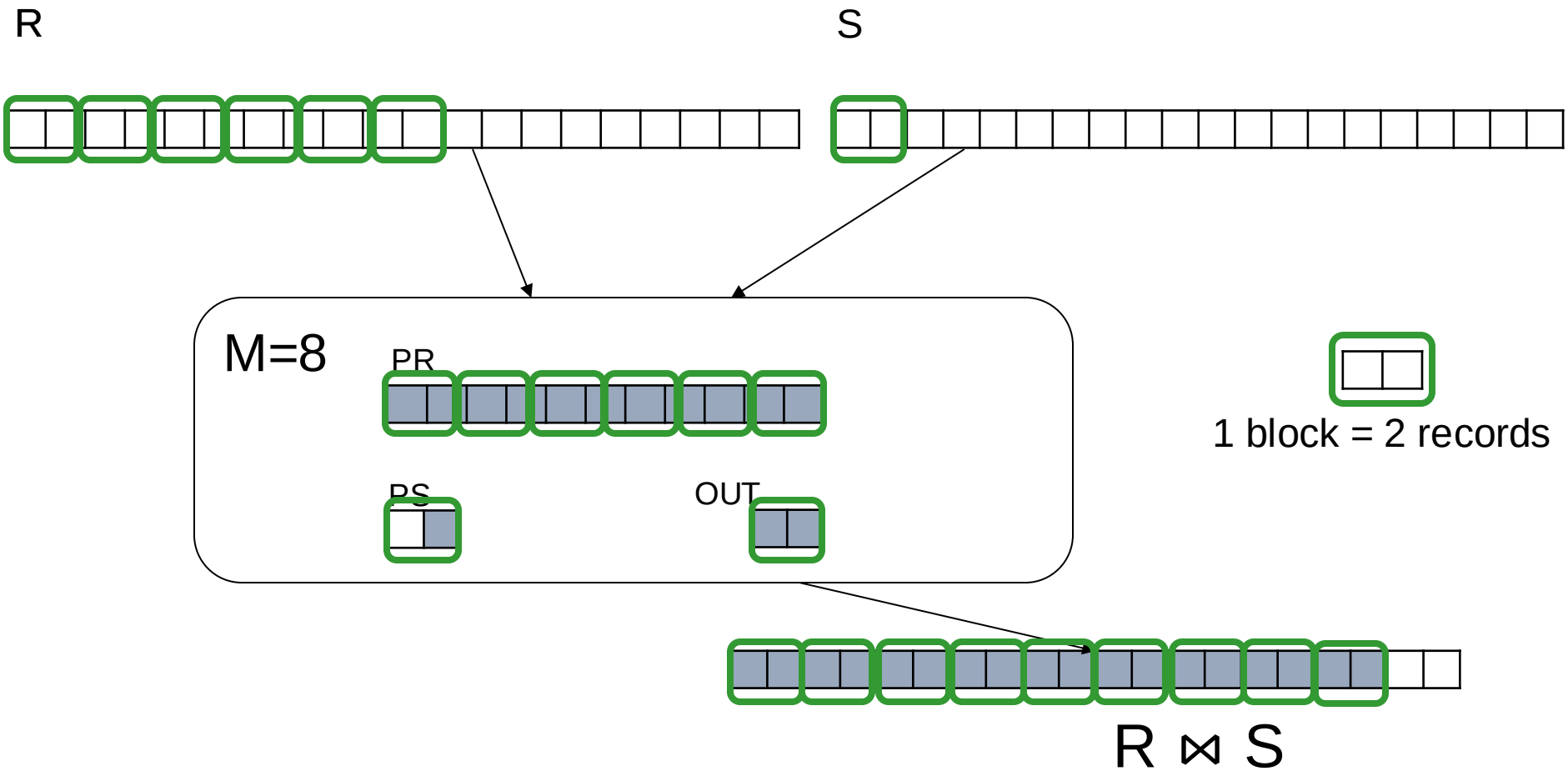
Block Nested Loop Join



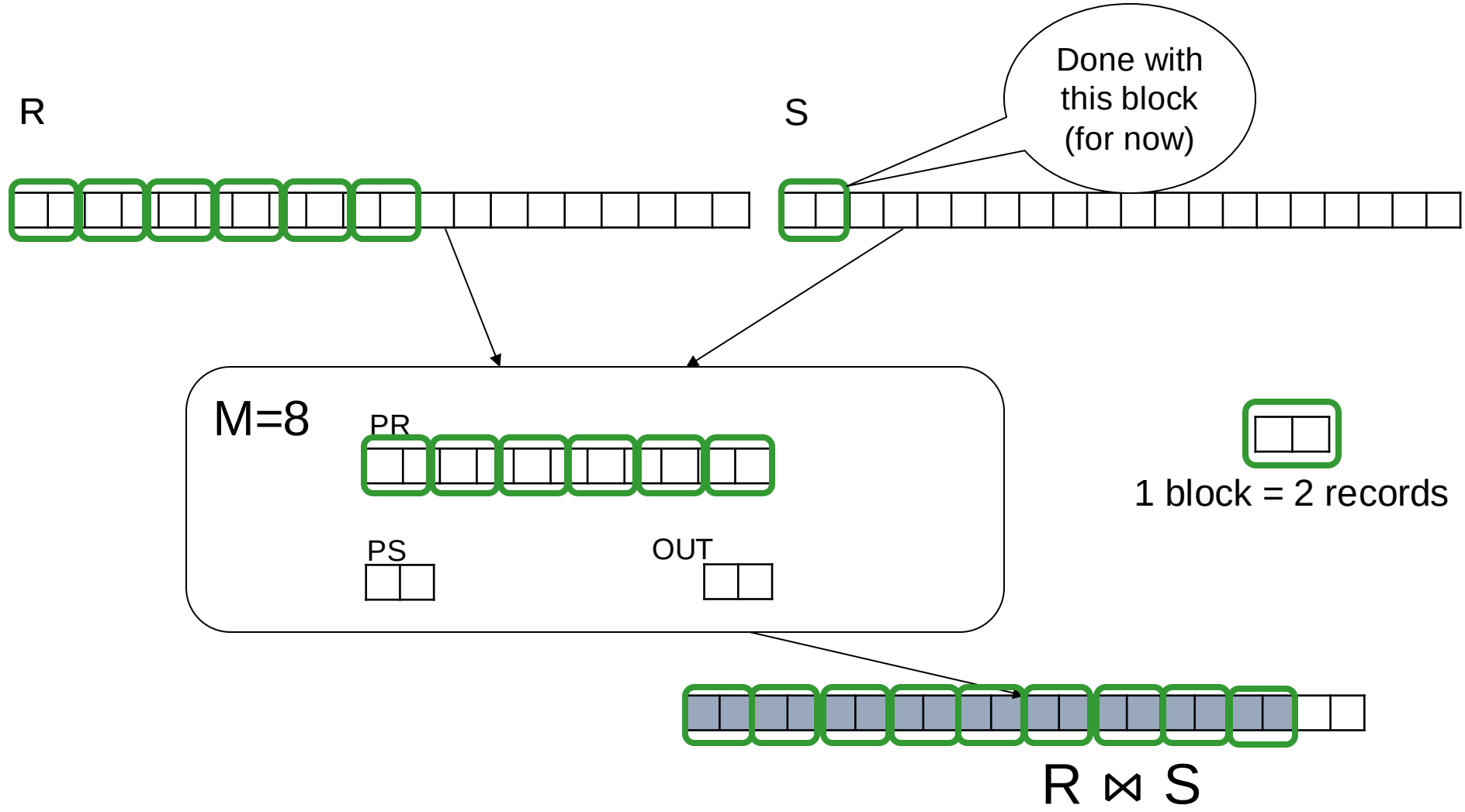
Block Nested Loop Join



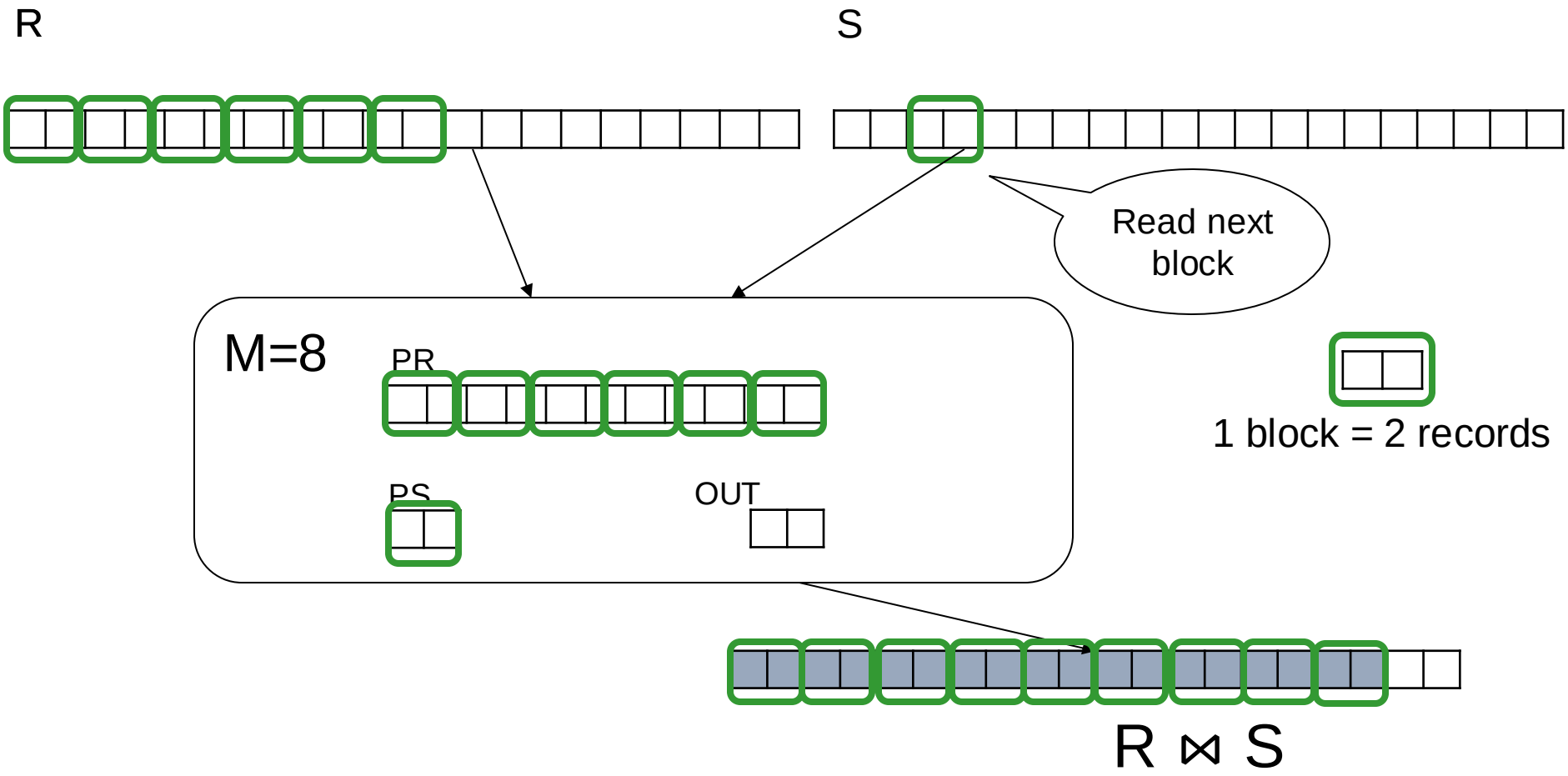
Block Nested Loop Join



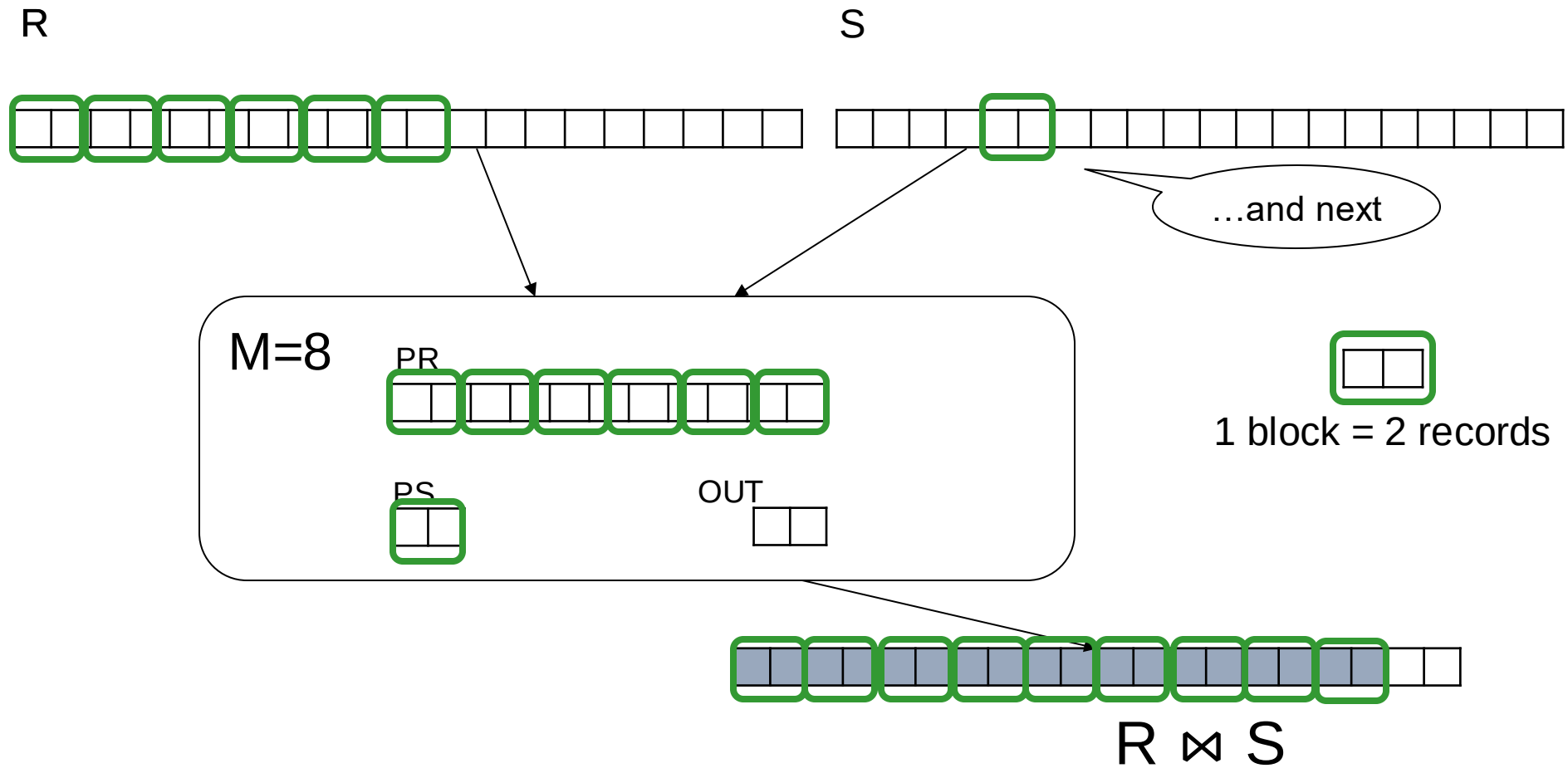
Block Nested Loop Join



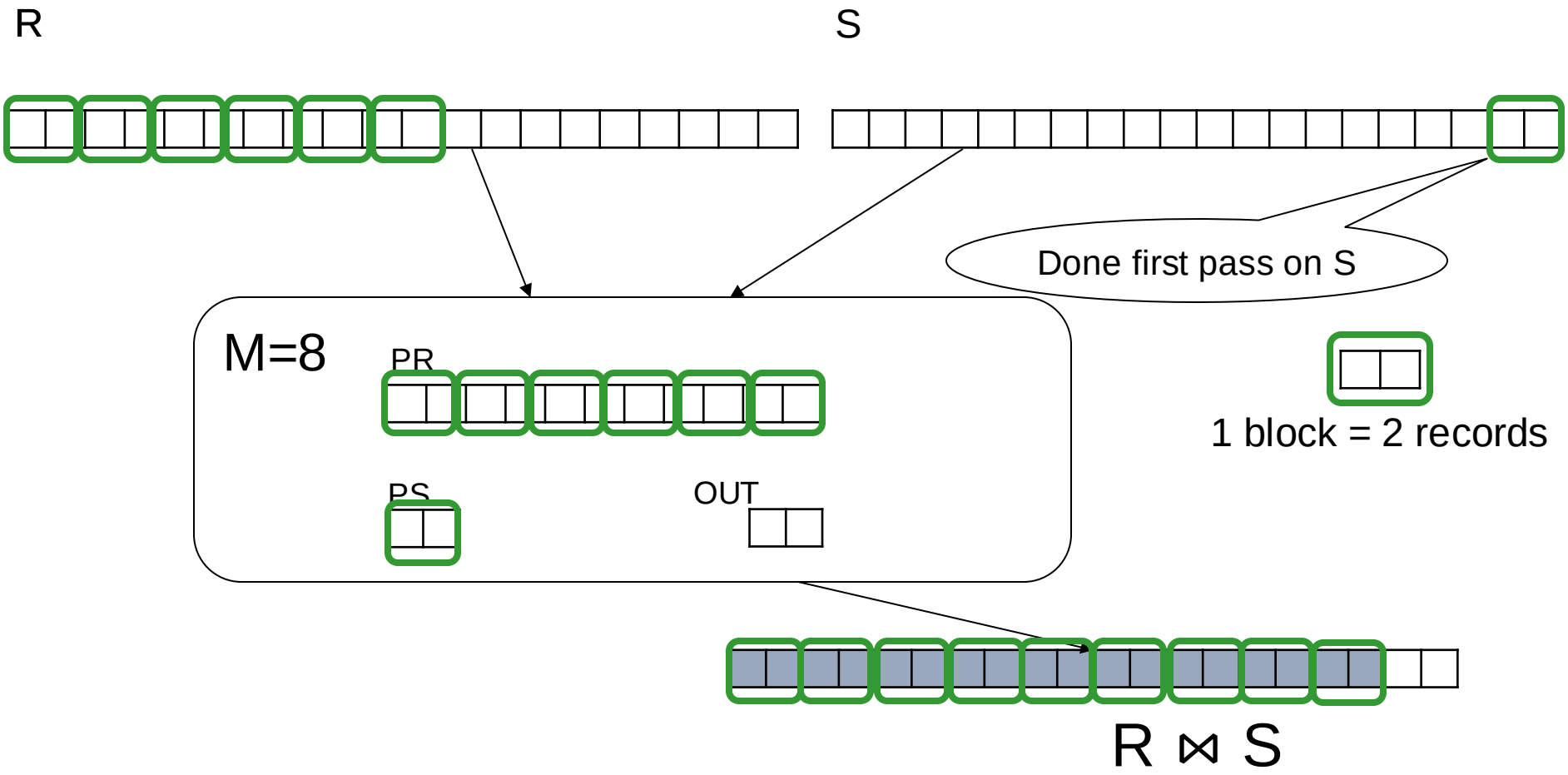
Block Nested Loop Join



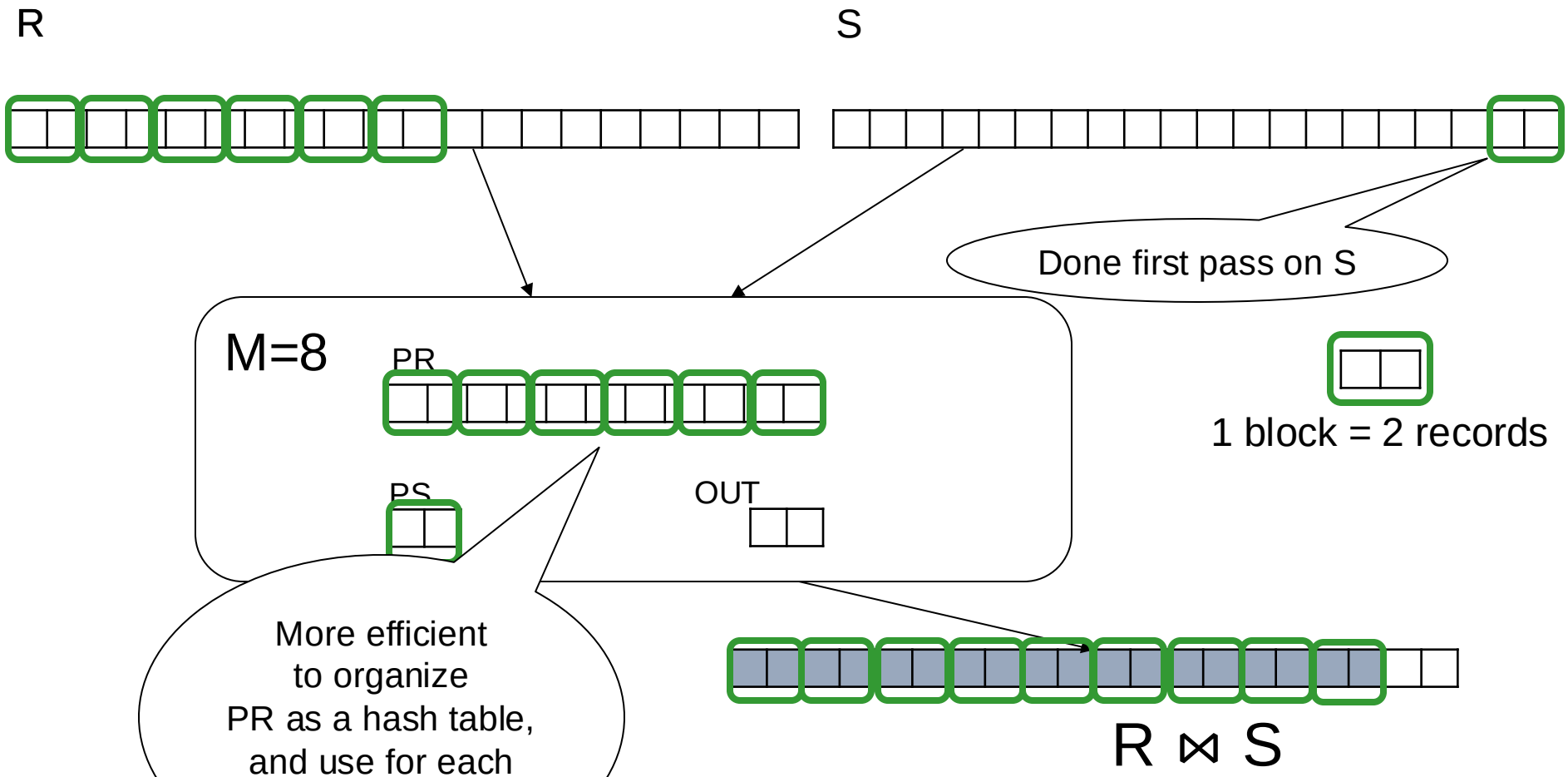
Block Nested Loop Join



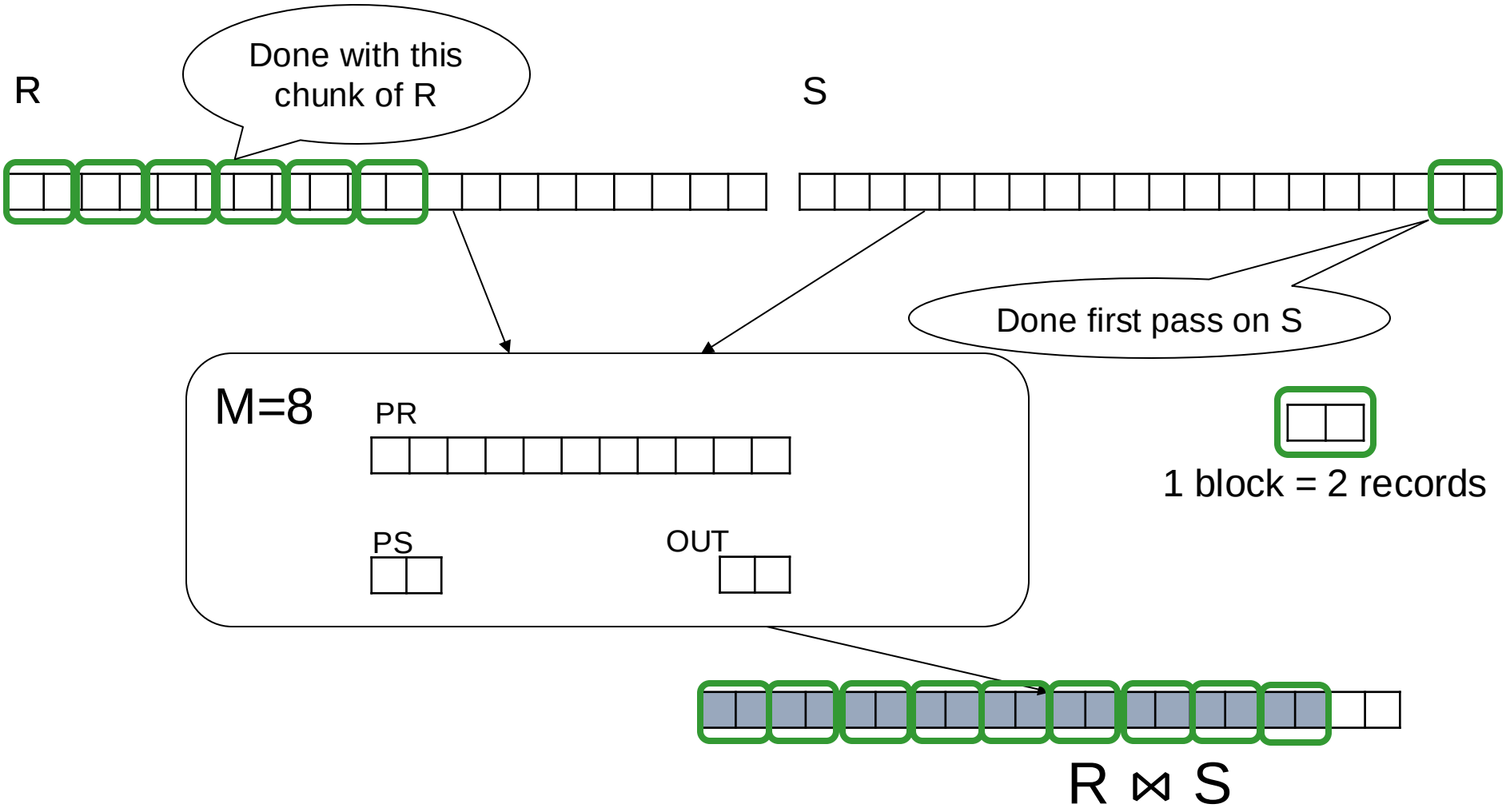
Block Nested Loop Join



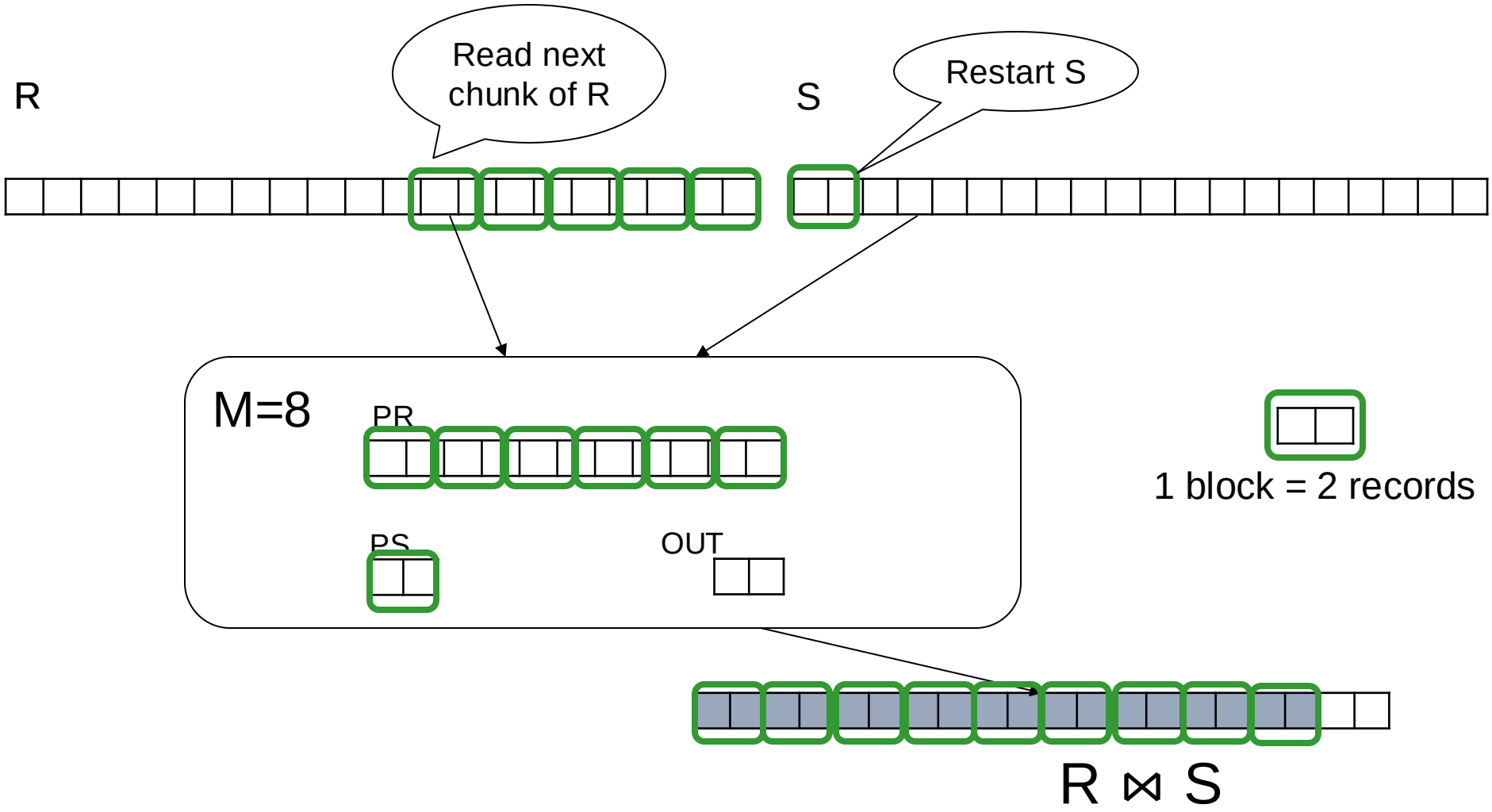
Block Nested Loop Join



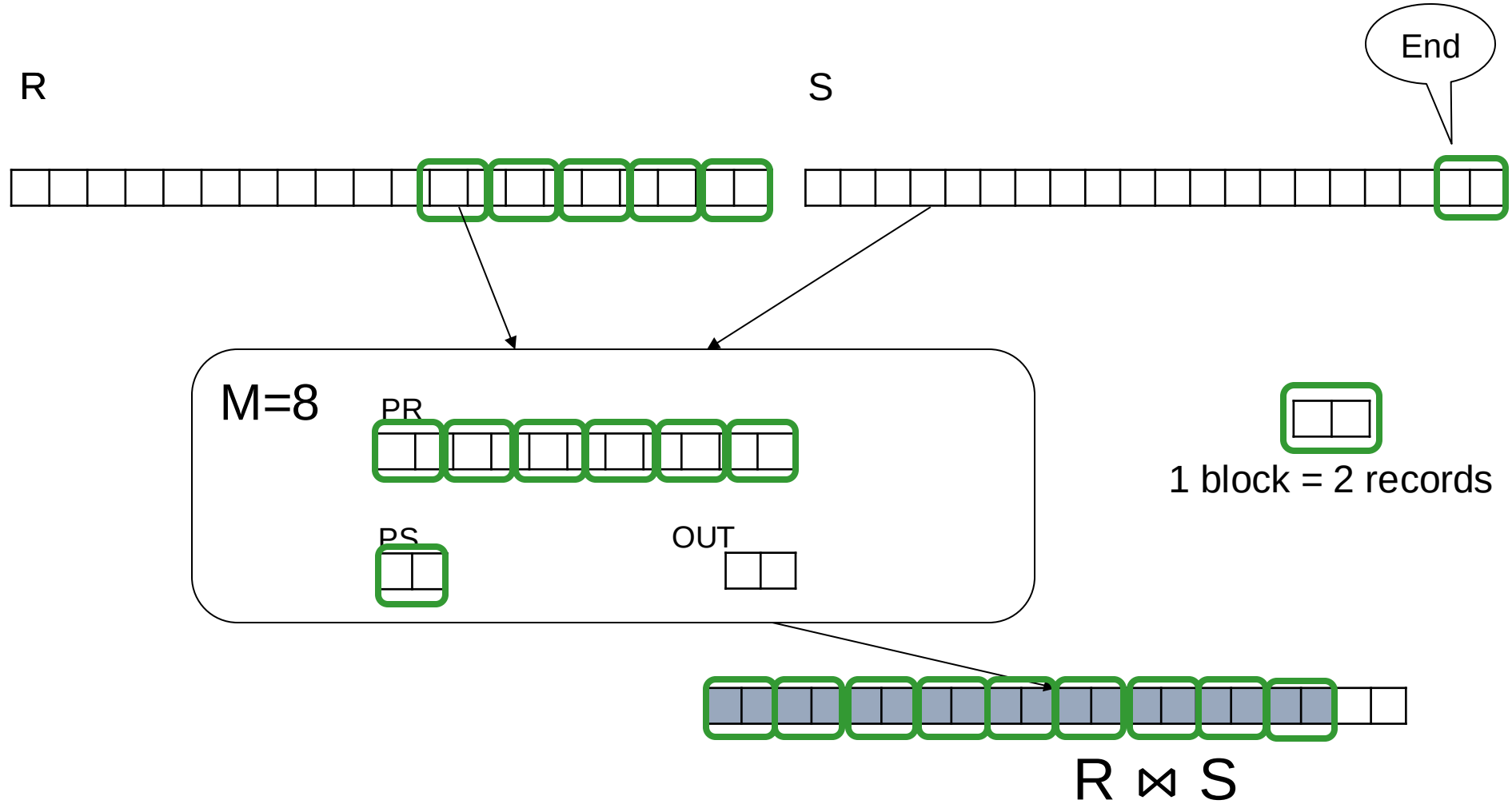
Block Nested Loop Join



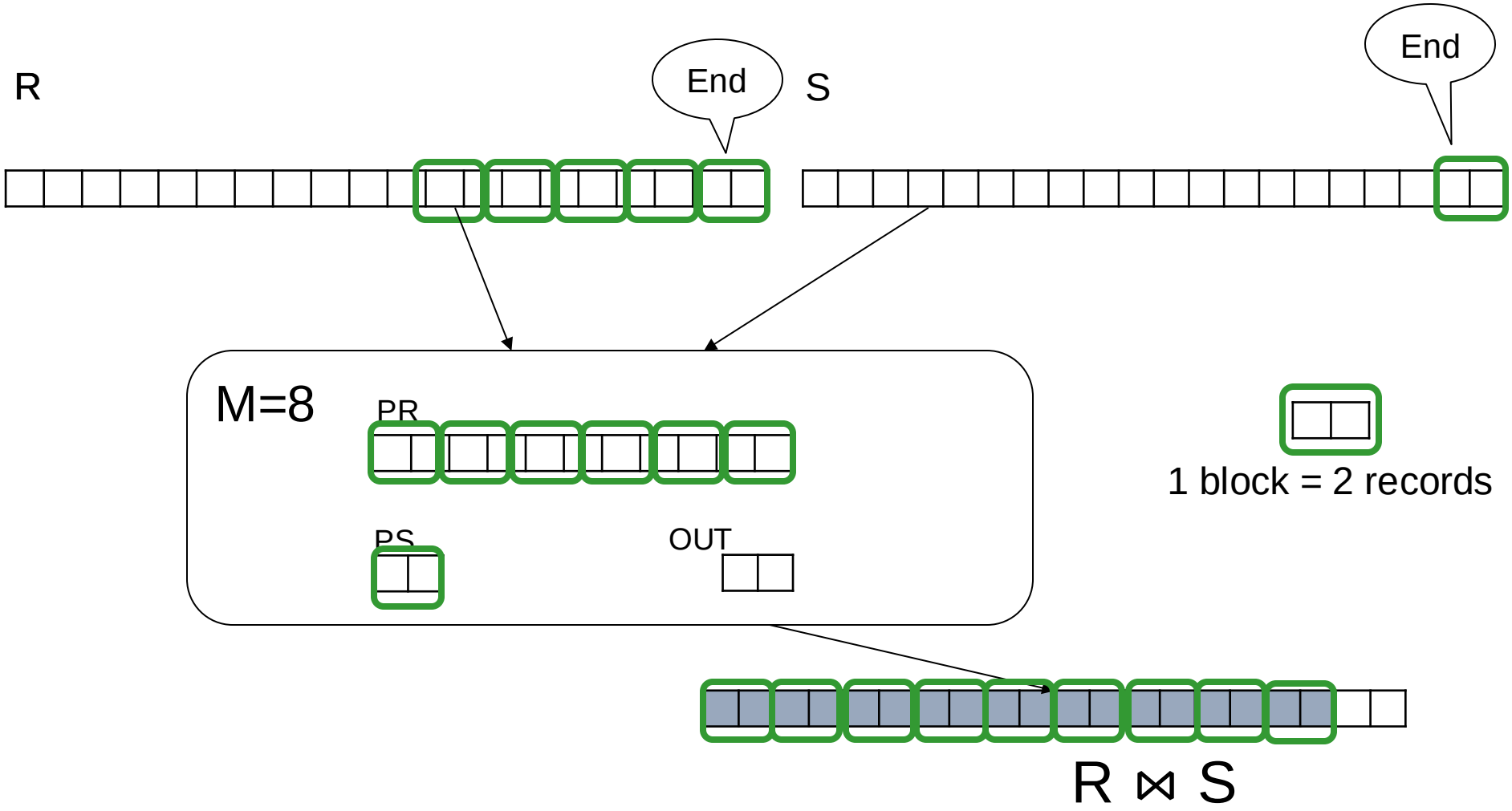
Block Nested Loop Join



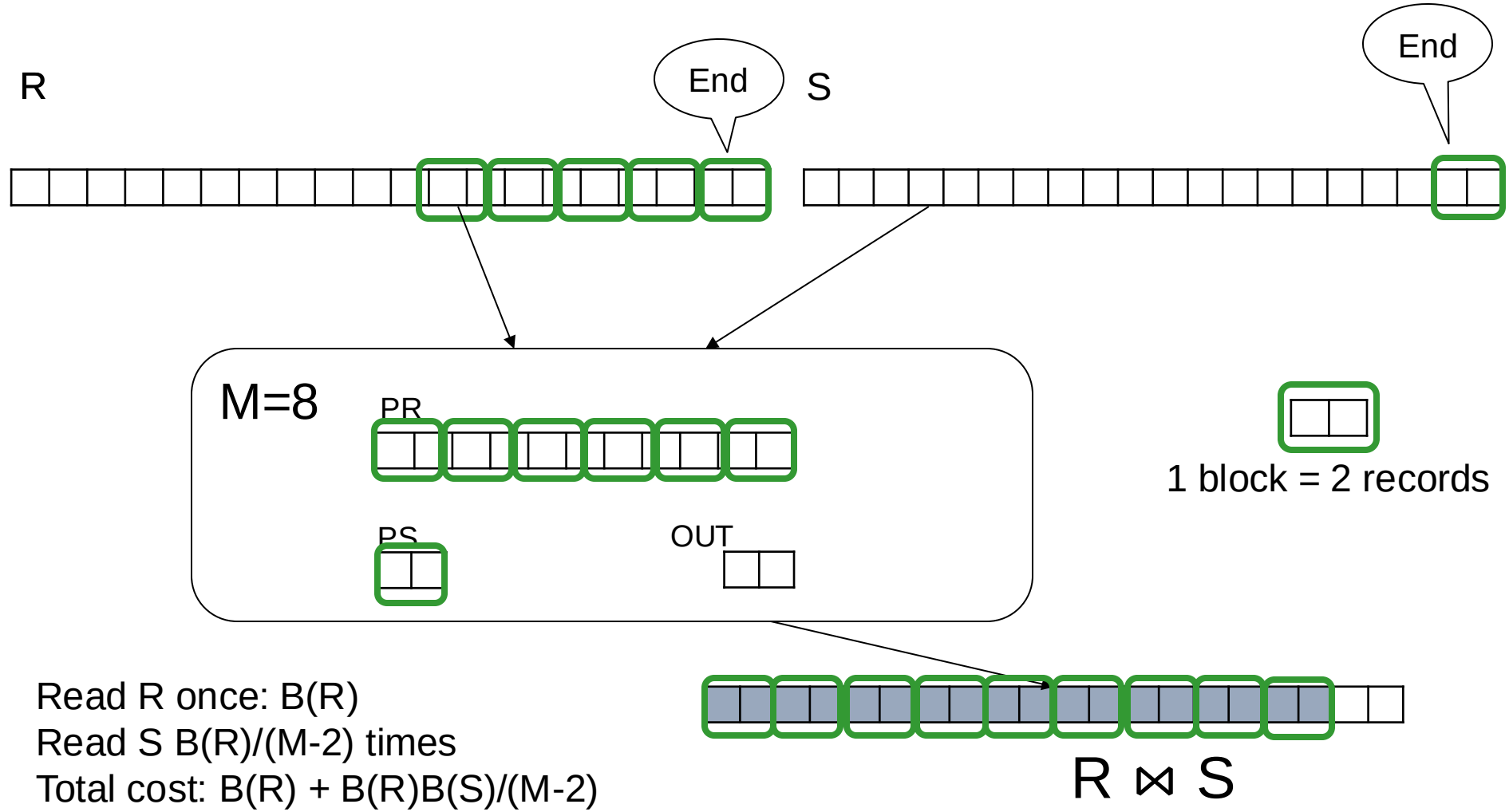
Block Nested Loop Join



Block Nested Loop Join



Block Nested Loop Join



Read R once: B(R)
Read S B(R)/(M-2) times
Total cost: B(R) + B(R)B(S)/(M-2)
We ignore the final write

Partitioned Hash-Join

Partitioned Hash-Join

- The outer relation R needs to fit in main memory
- The inner relation S doesn't need to fit

```
for x in R do
      insert(x.key, x)

for y in S do
      x = find(y.key);
      output(x,y);
```

Partitioned Hash-Join

- $R \bowtie S$, both bigger than main memory

Partitioned Hash-Join

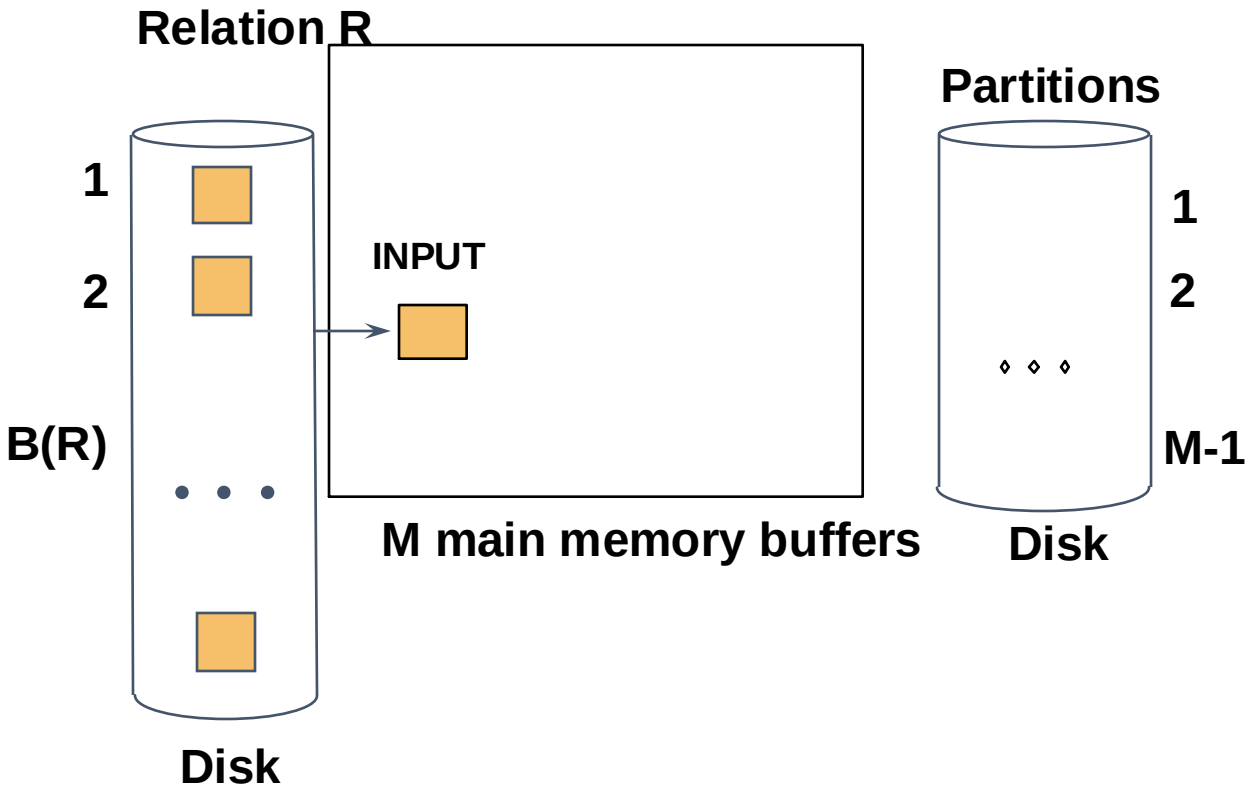
- $R \bowtie S$, both bigger than main memory
- Step 1:
  - Hash partition both R and S
  - Store buckets on disk

Partitioned Hash-Join

- $R \bowtie S$, both bigger than main memory
- Step 1:
  - Hash partition both R and S
  - Store buckets on disk
- Step 2:
  - Read one R-bucket in main memory
  - Hash-Join with corresponding S-bucket
  - Repeat for all buckets

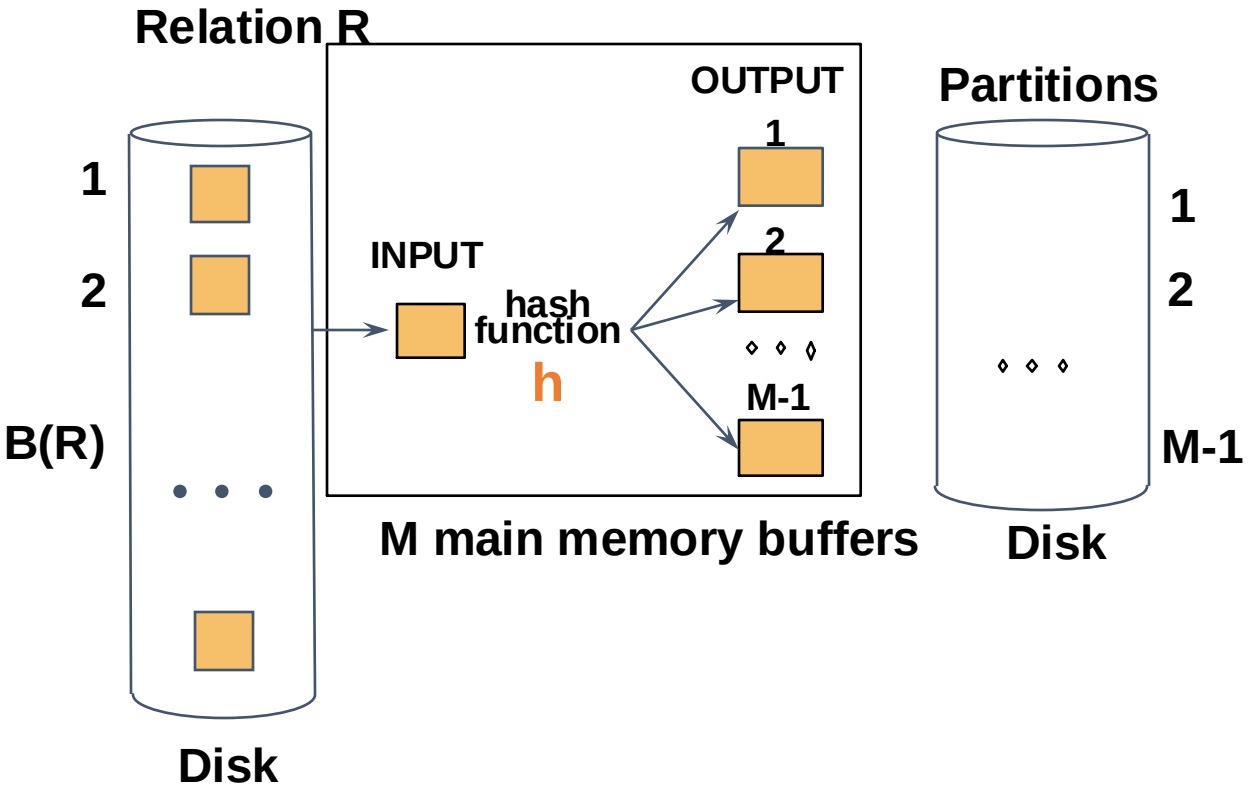
Step 1: Hash-partition

- Partition R into buckets, on disk



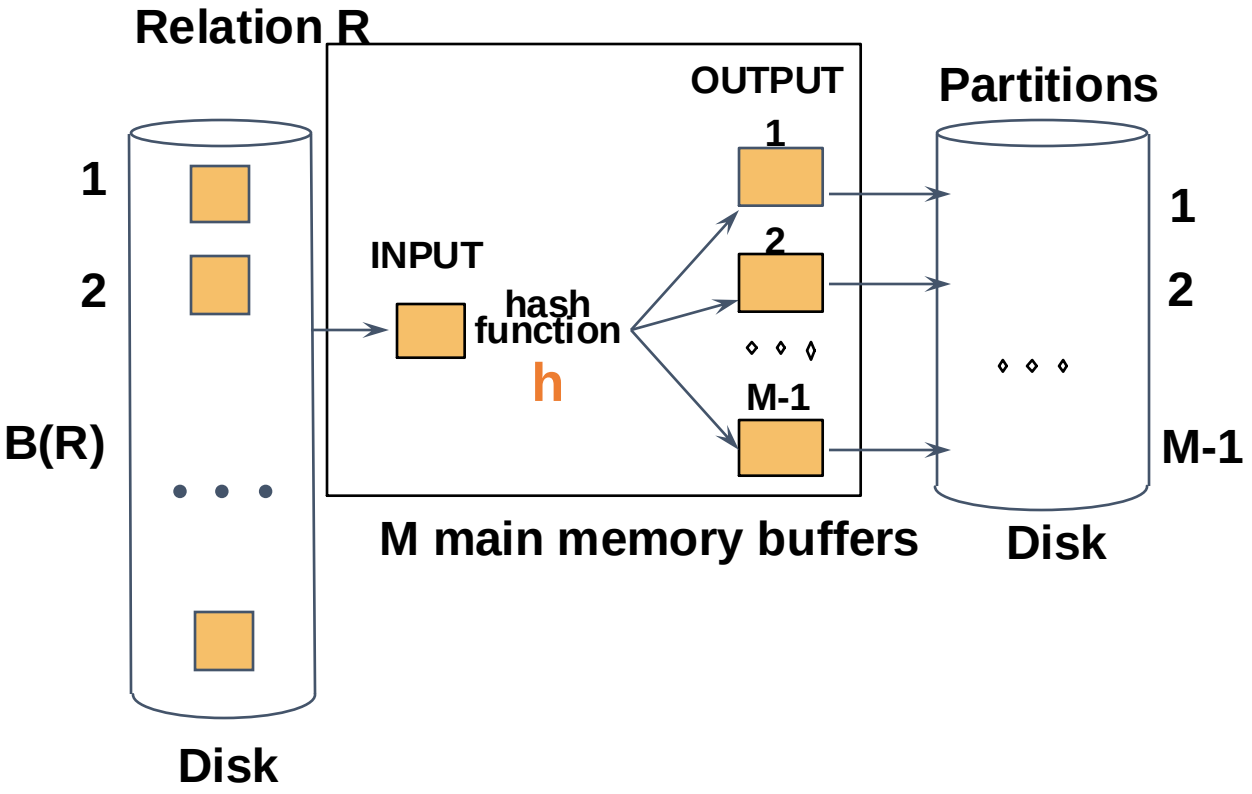
Step 1: Hash-partition

- Partition R into buckets, on disk



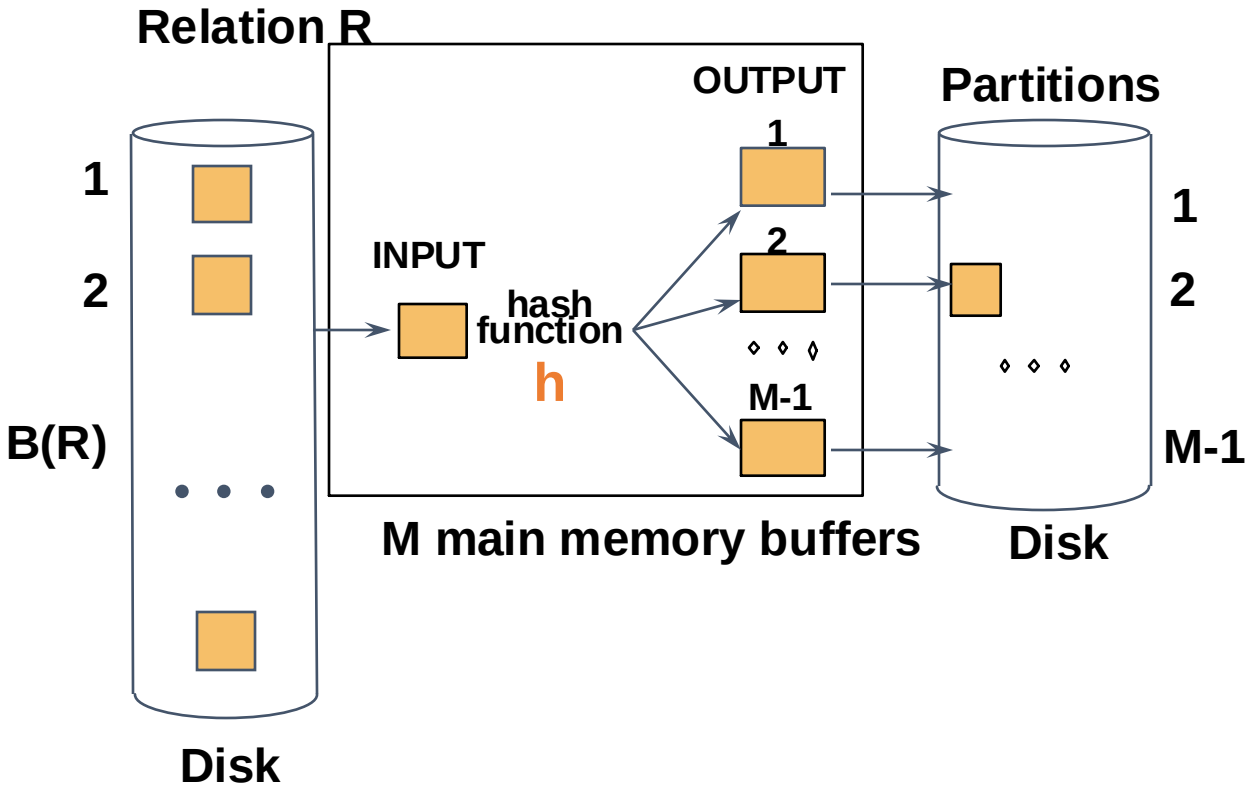
Step 1: Hash-partition

- Partition R into buckets, on disk



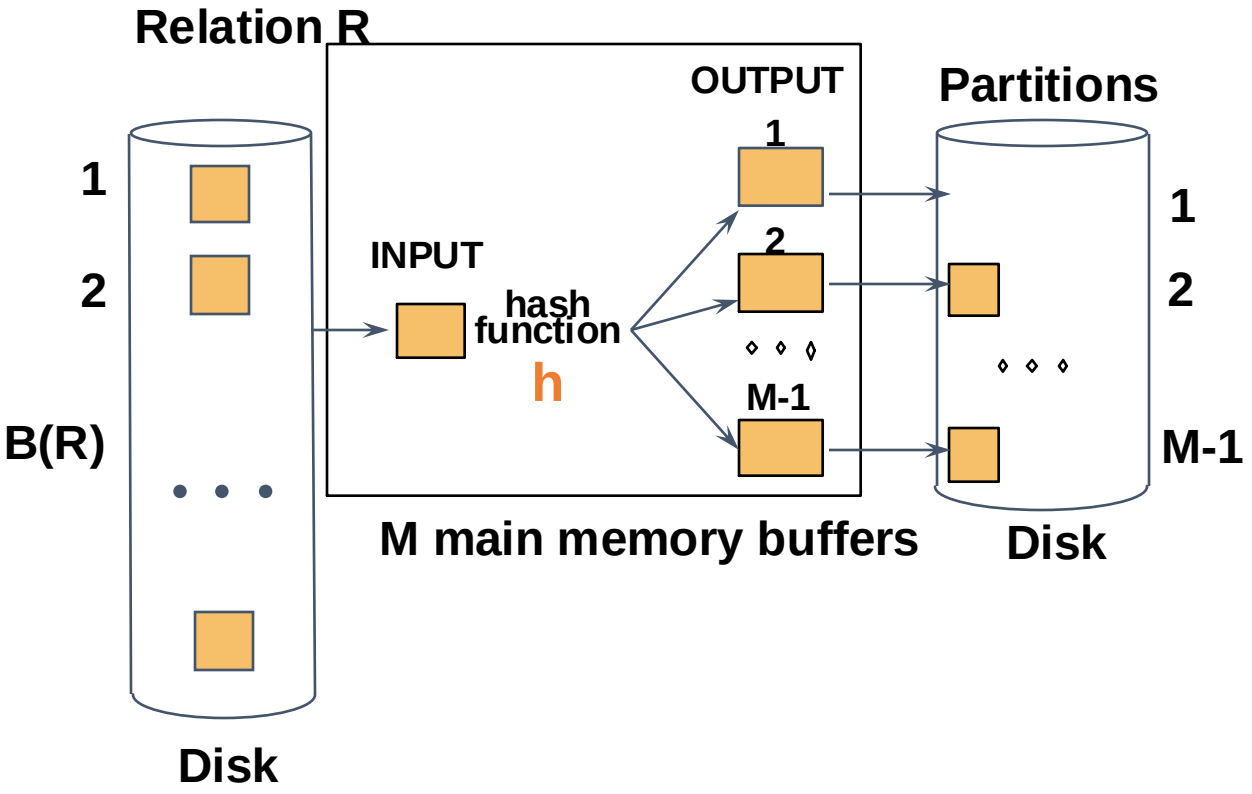
Step 1: Hash-partition

- Partition R into buckets, on disk



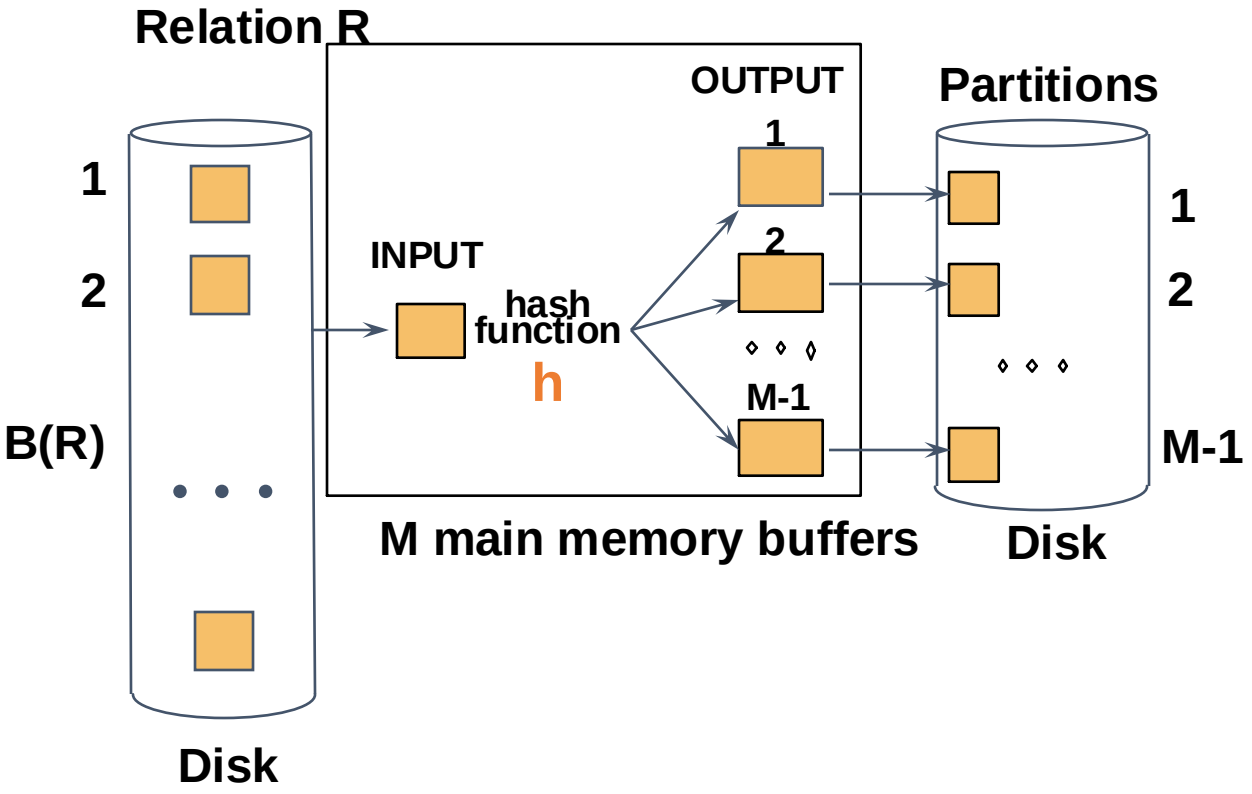
Step 1: Hash-partition

- Partition R into buckets, on disk



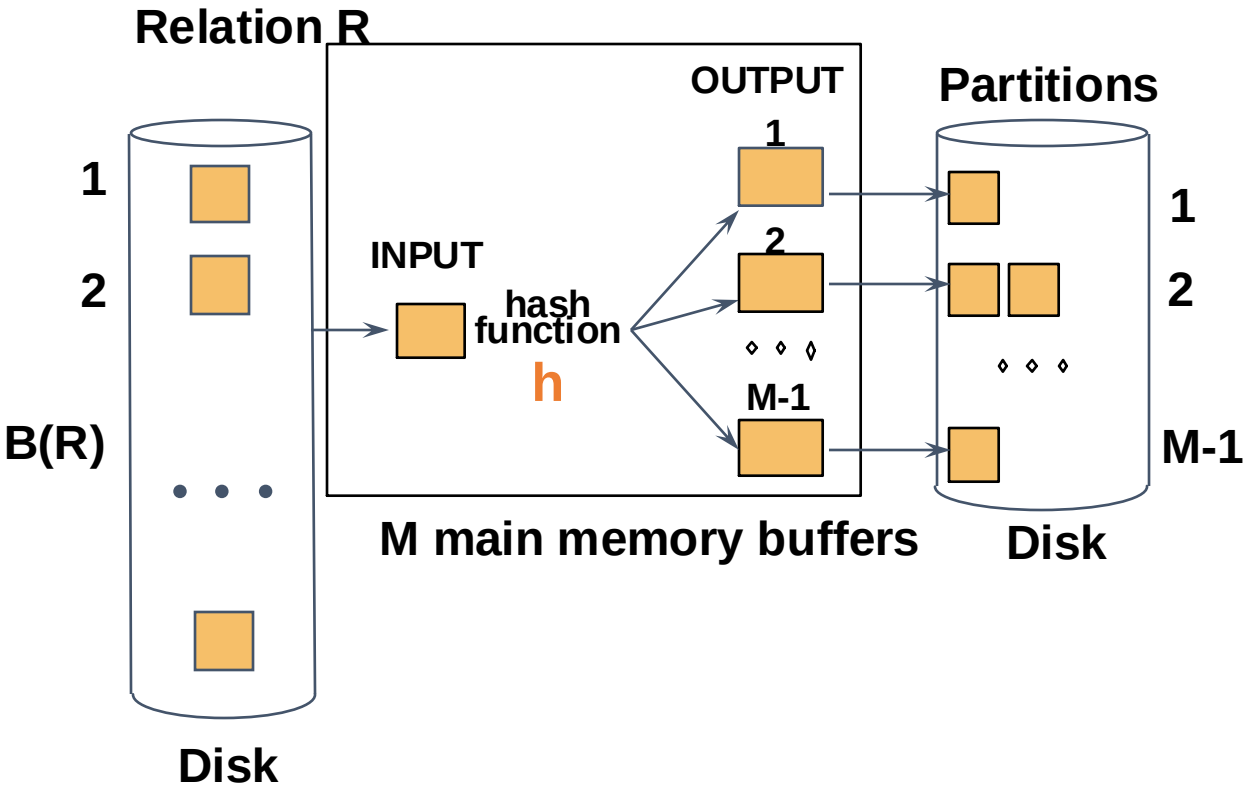
Step 1: Hash-partition

- Partition R into buckets, on disk



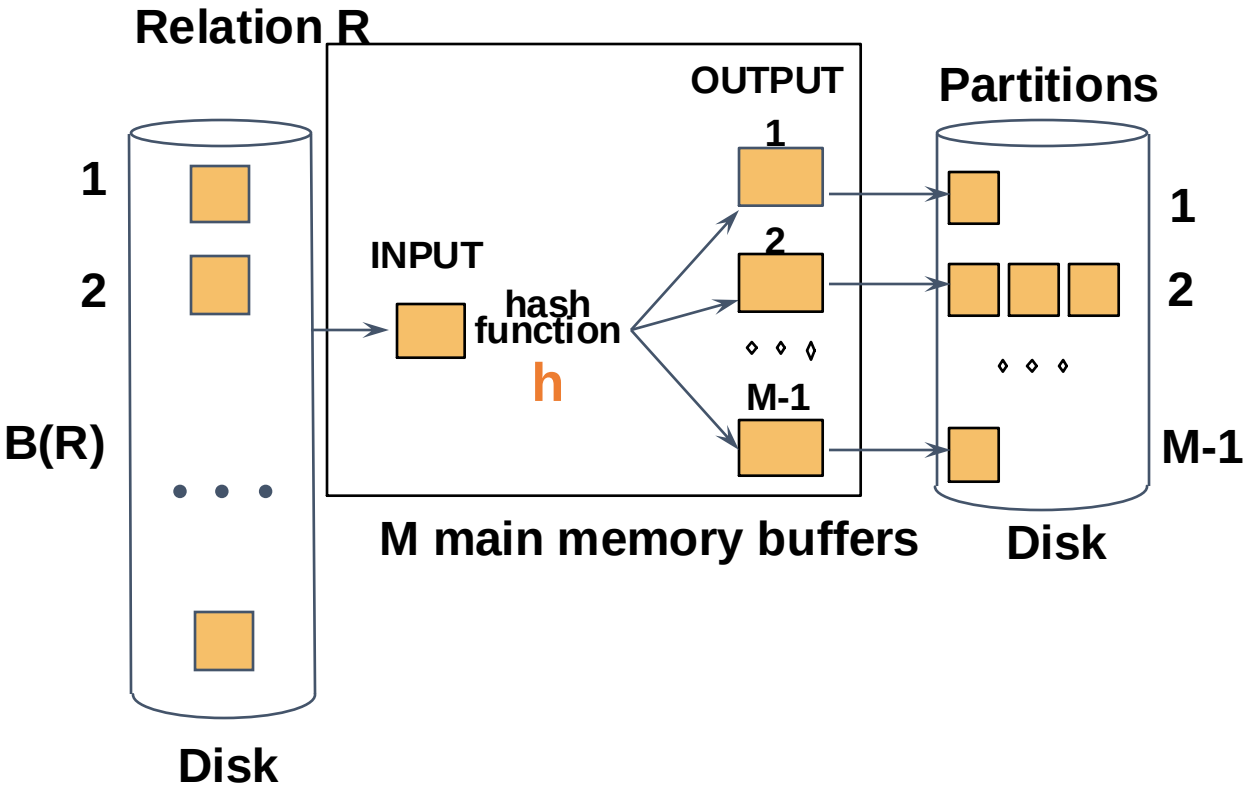
Step 1: Hash-partition

- Partition R into buckets, on disk



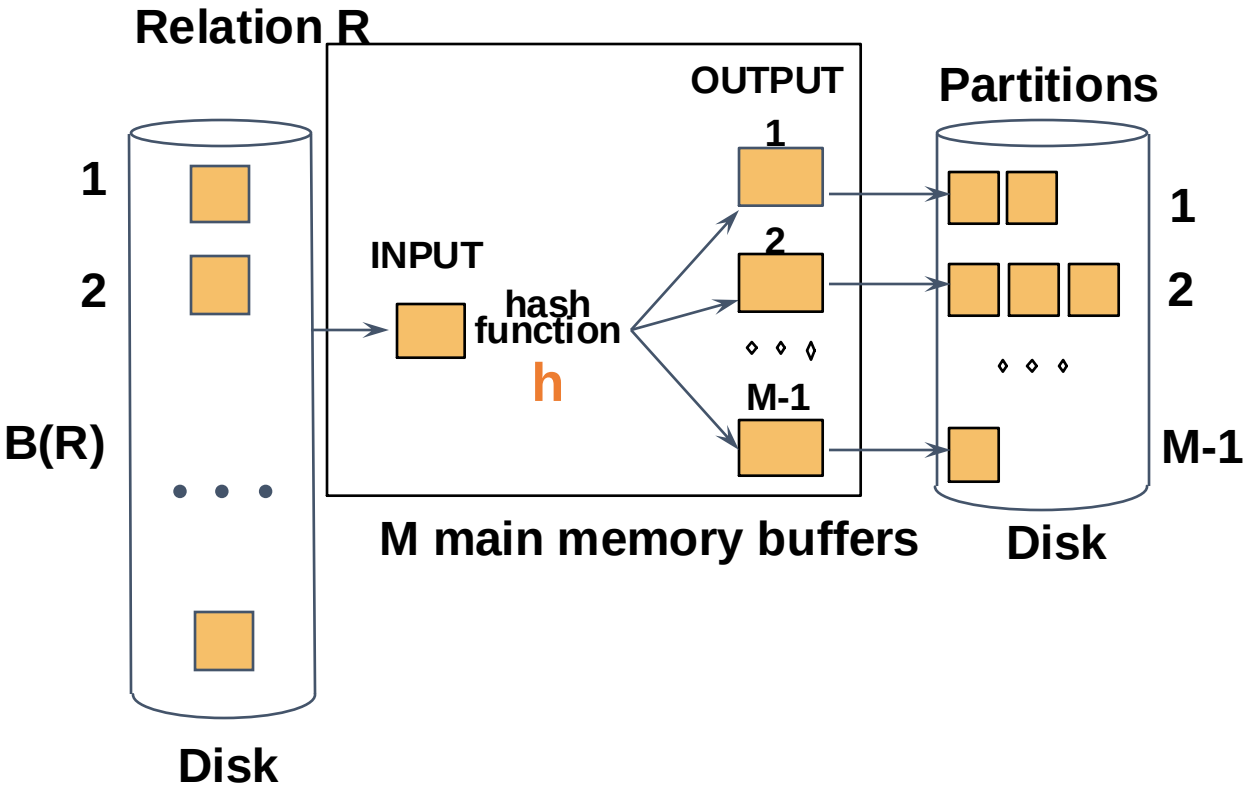
Step 1: Hash-partition

- Partition R into buckets, on disk



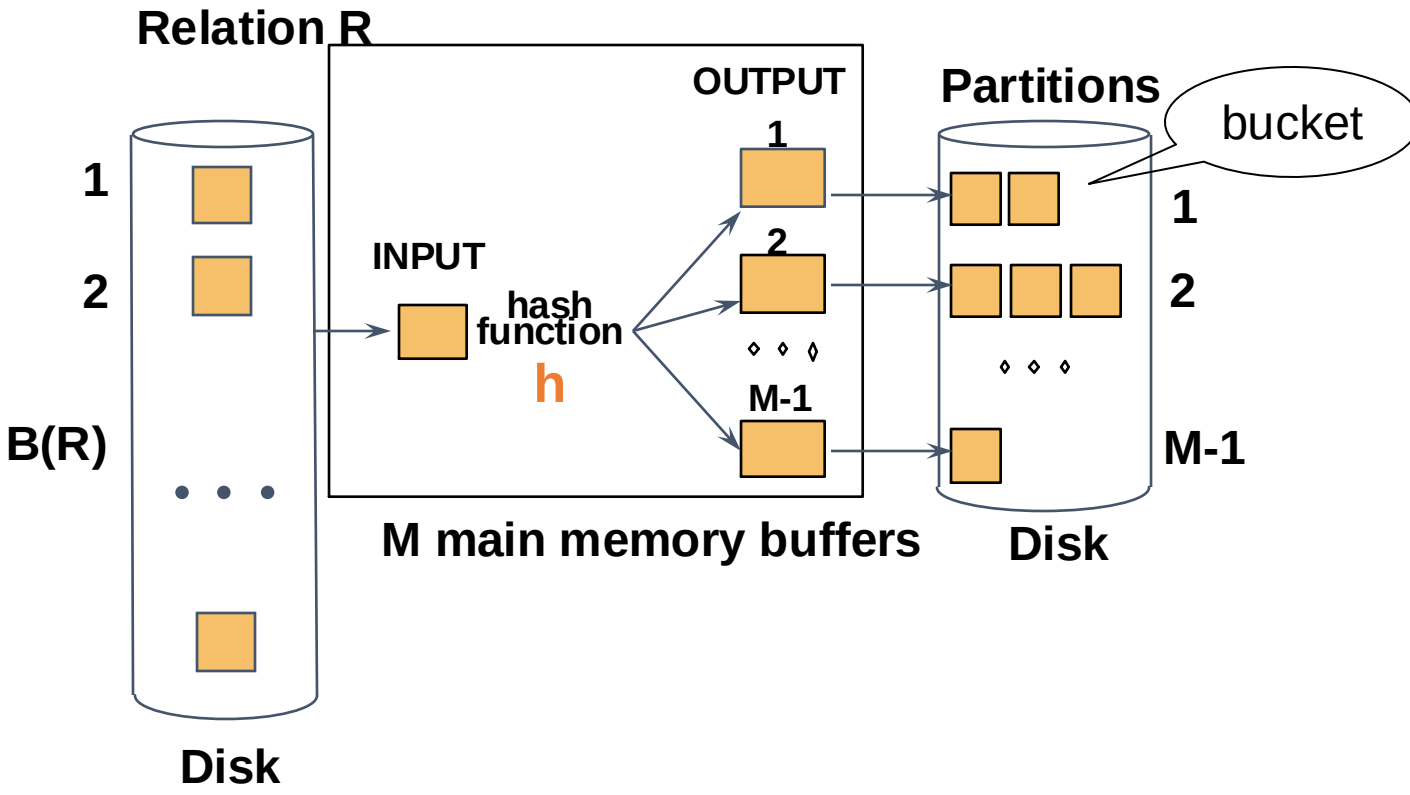
Step 1: Hash-partition

- Partition R into buckets, on disk



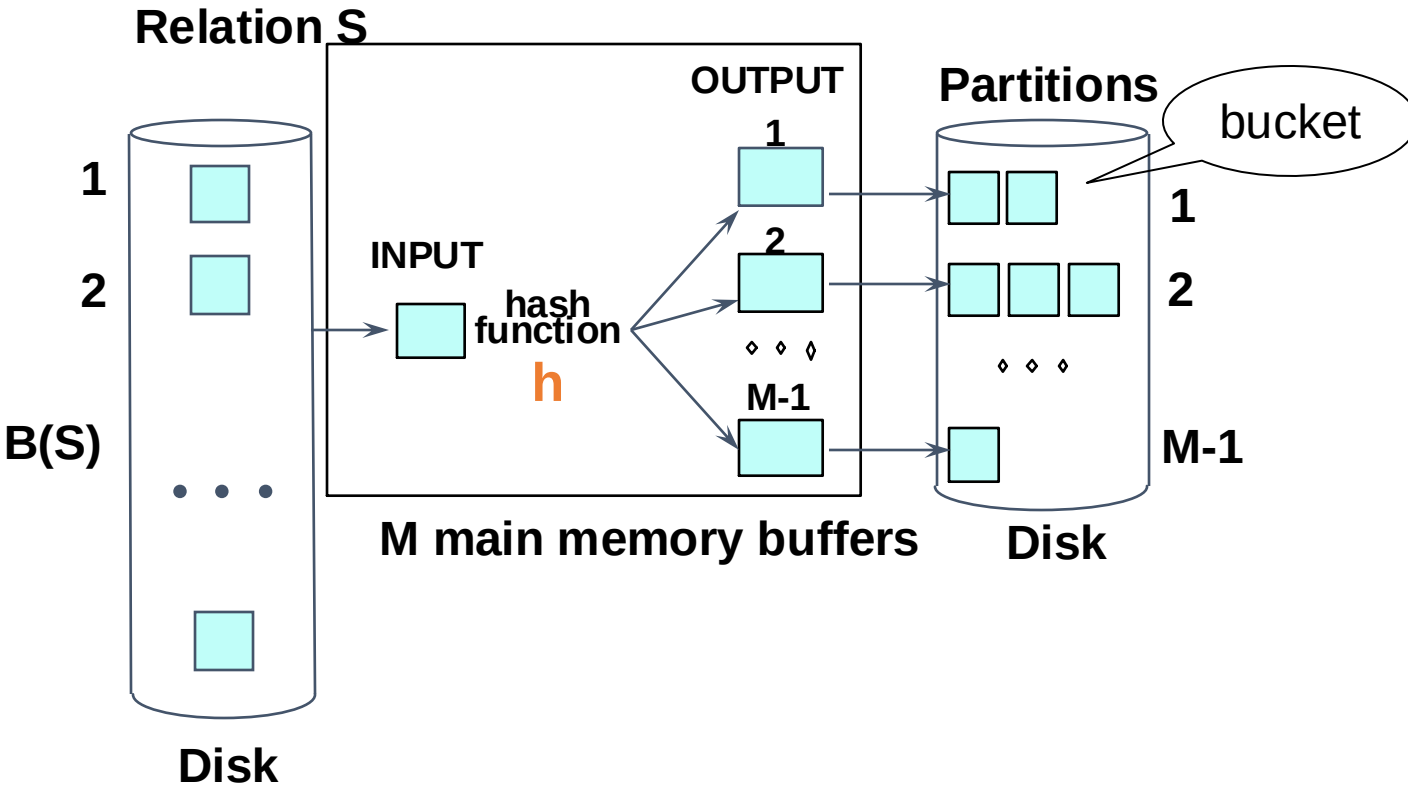
Step 1: Hash-partition

- Partition R into buckets, on disk



Step 1: Hash-partition

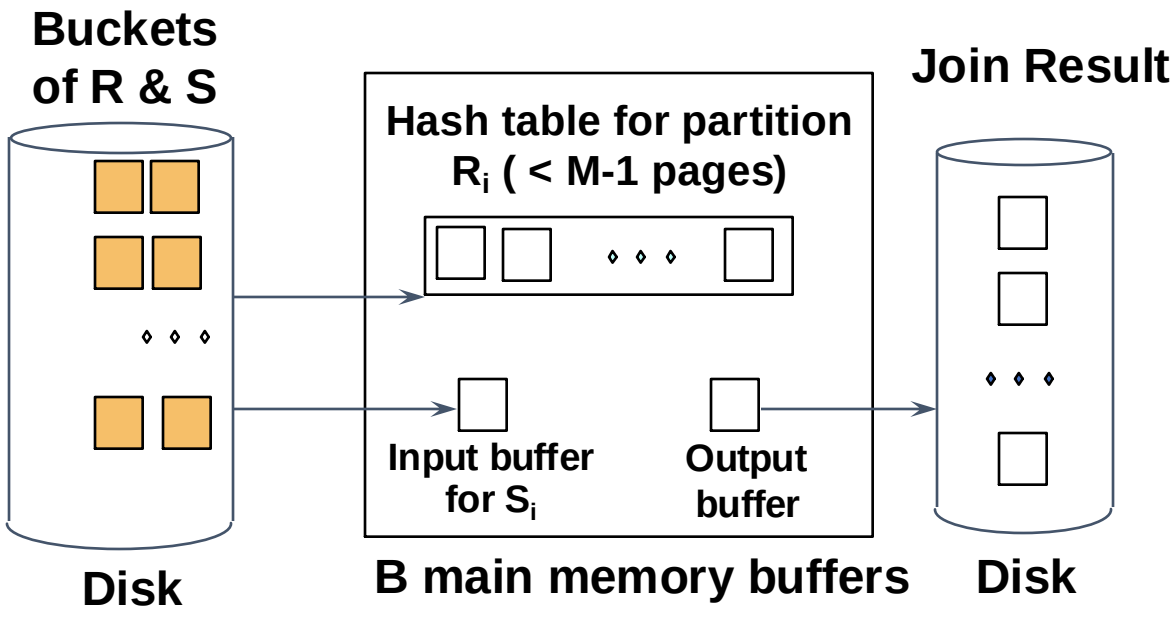
- Partition R into buckets, on disk
- Partition S



Step 2: Join Buckets

$R \bowtie S$

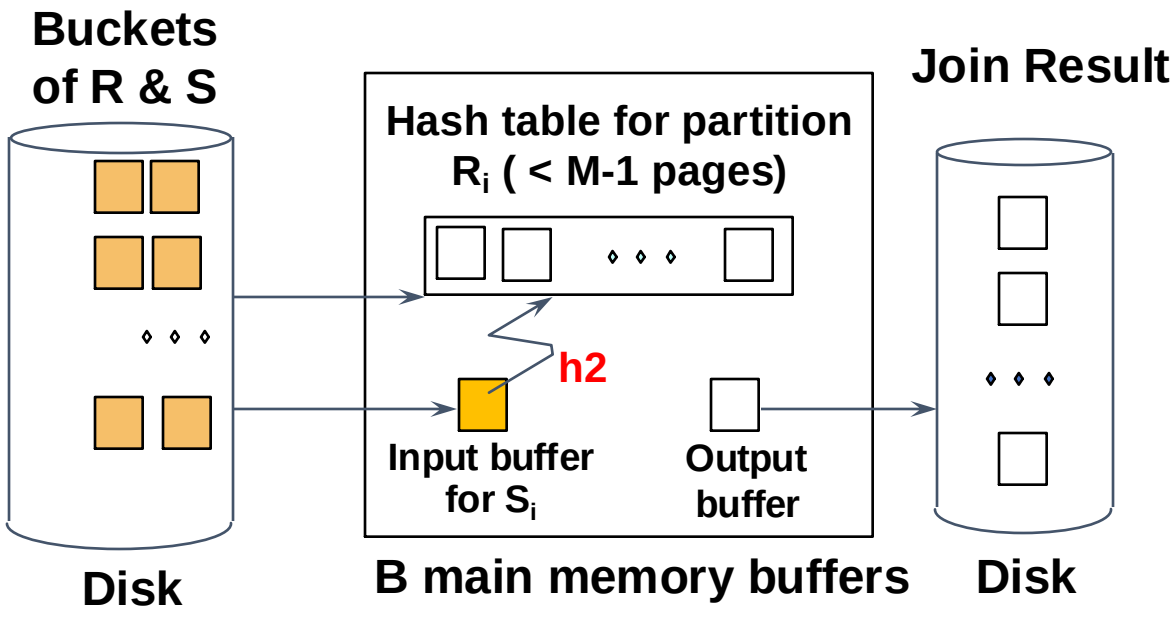
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

$R \bowtie S$

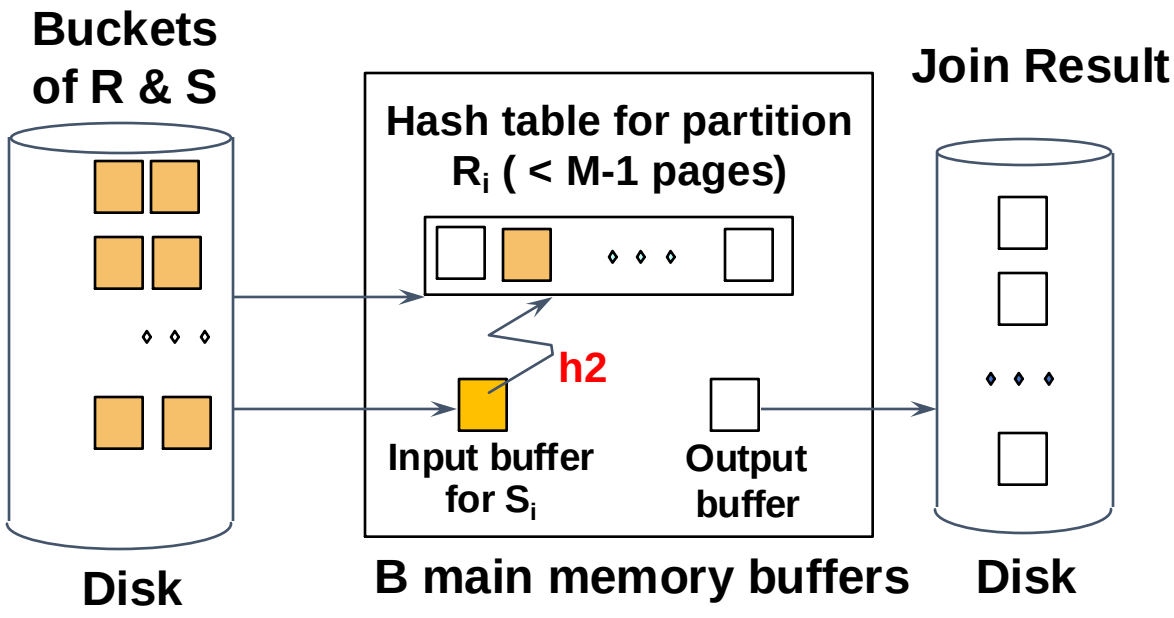
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

$R \bowtie S$

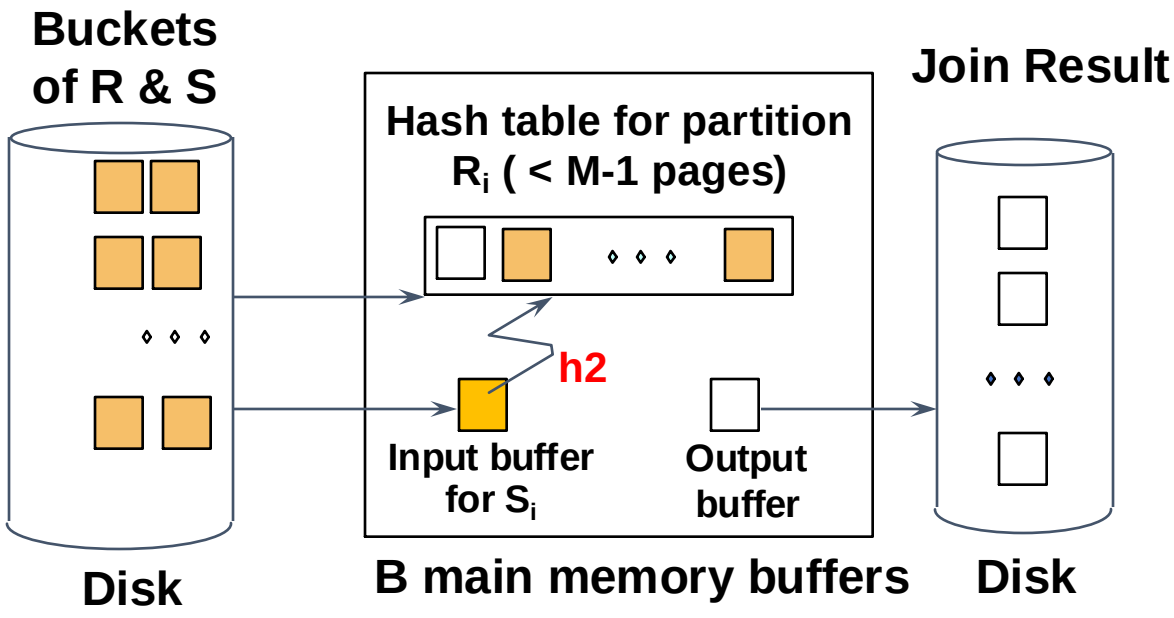
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

$R \bowtie S$

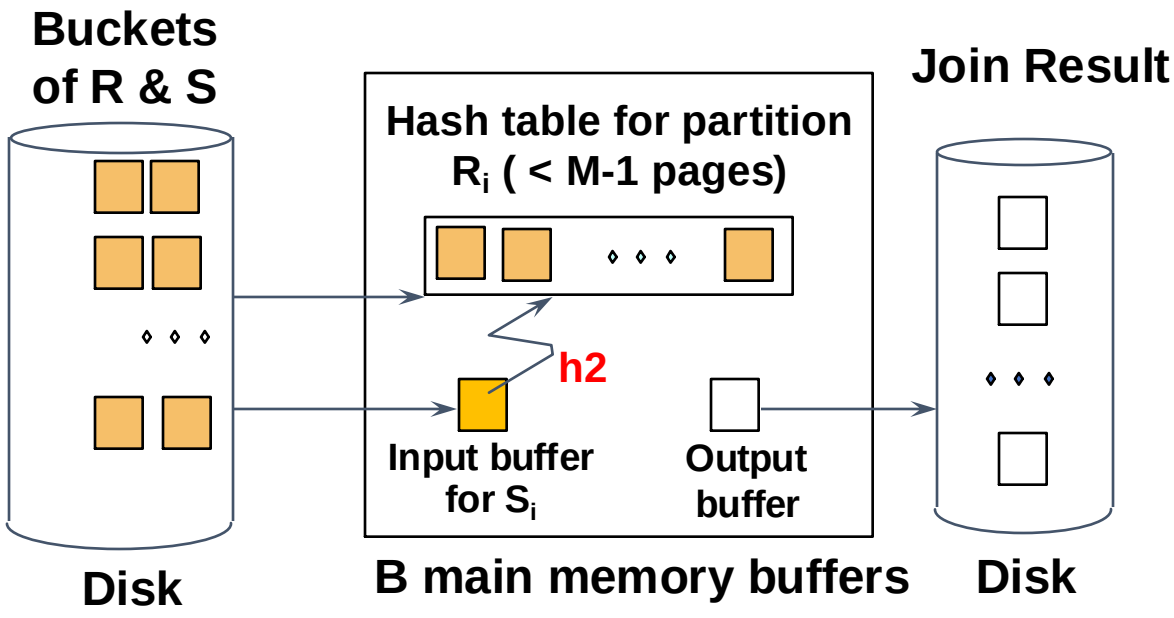
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

$R \bowtie S$

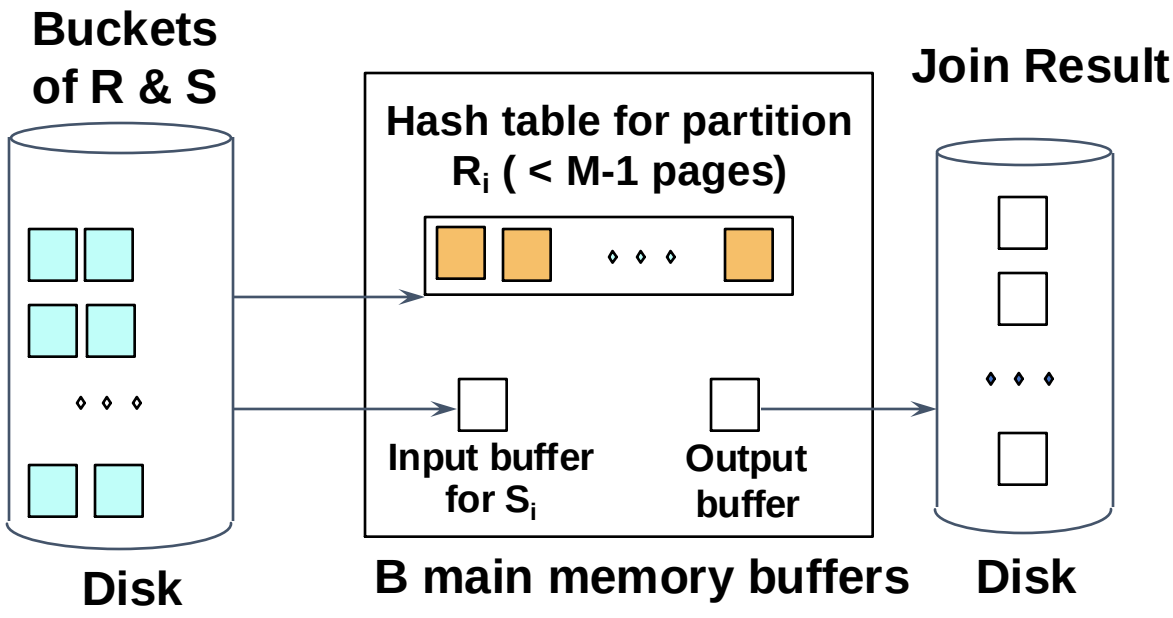
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)



Step 2: Join Buckets

$R \bowtie S$

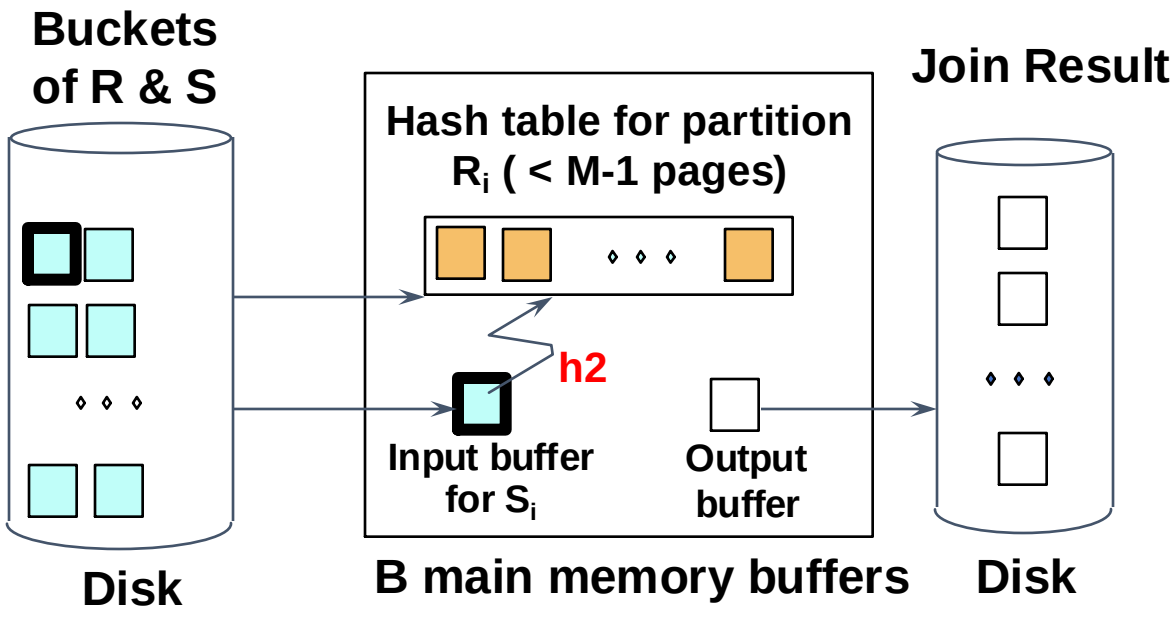
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

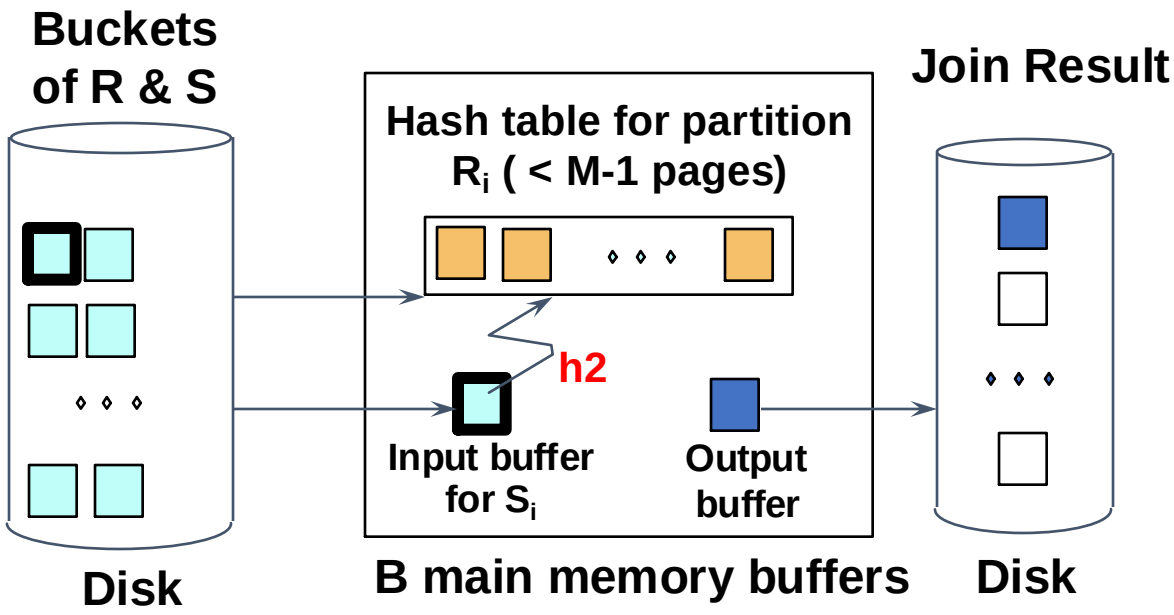
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

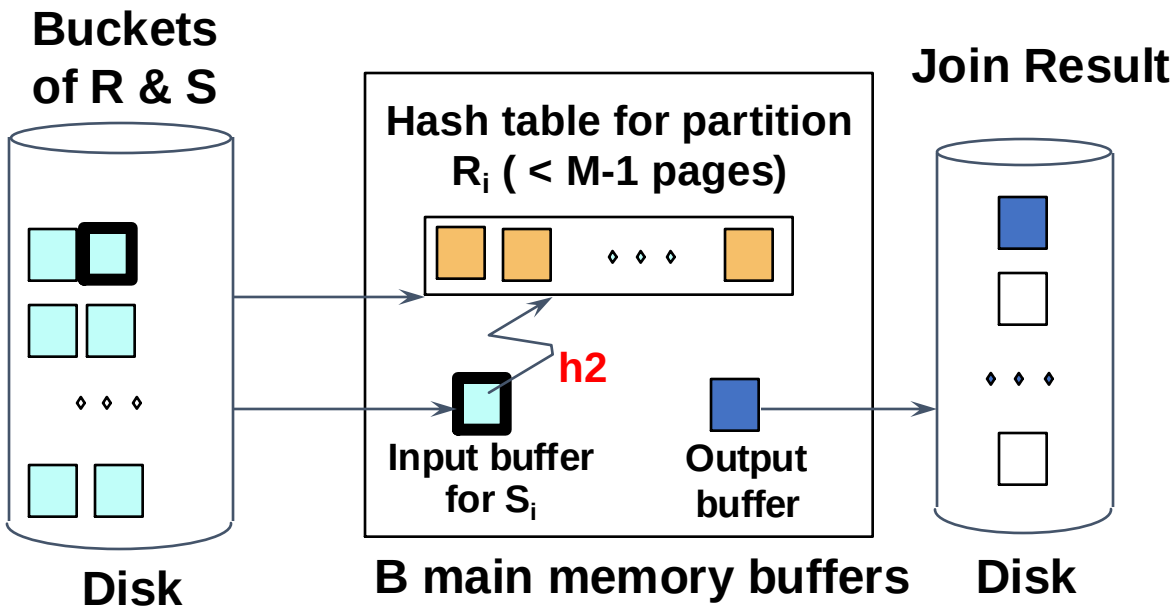
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

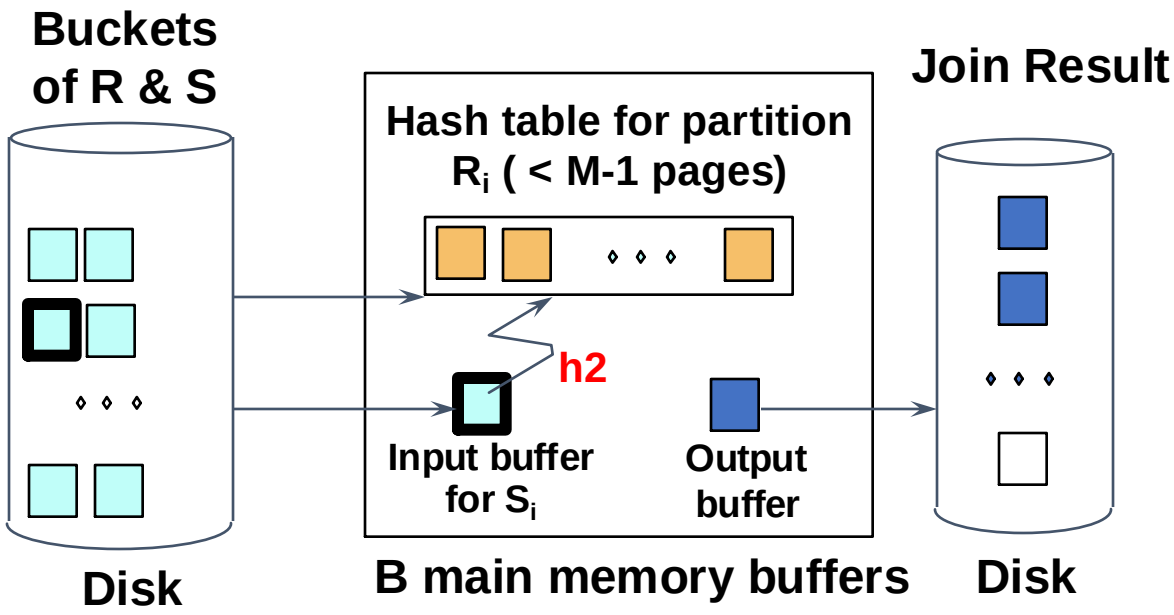
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

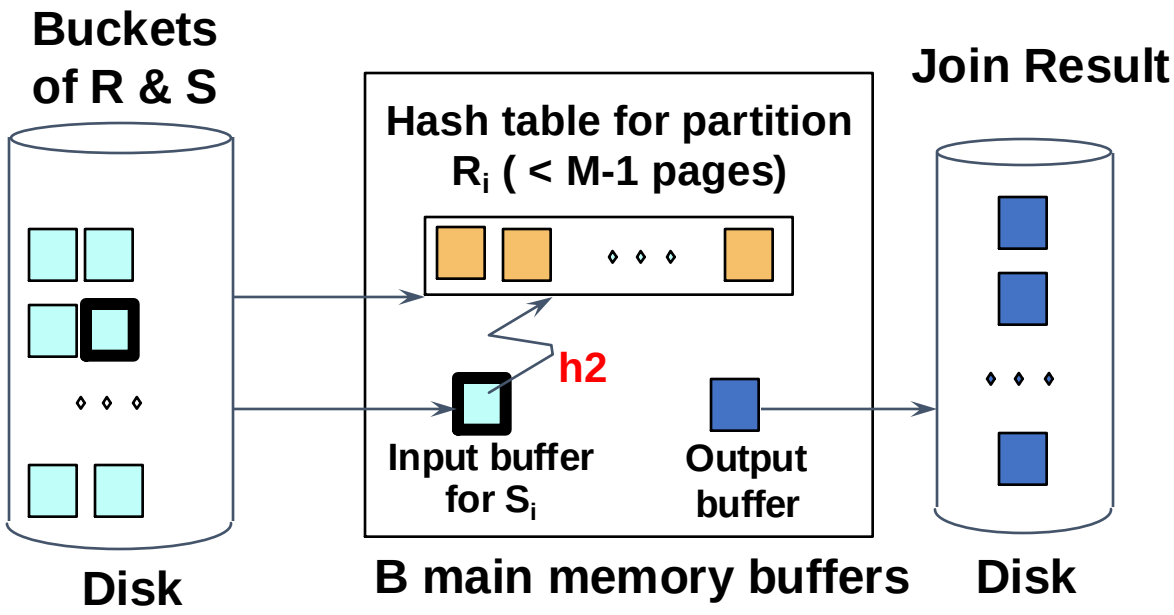
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

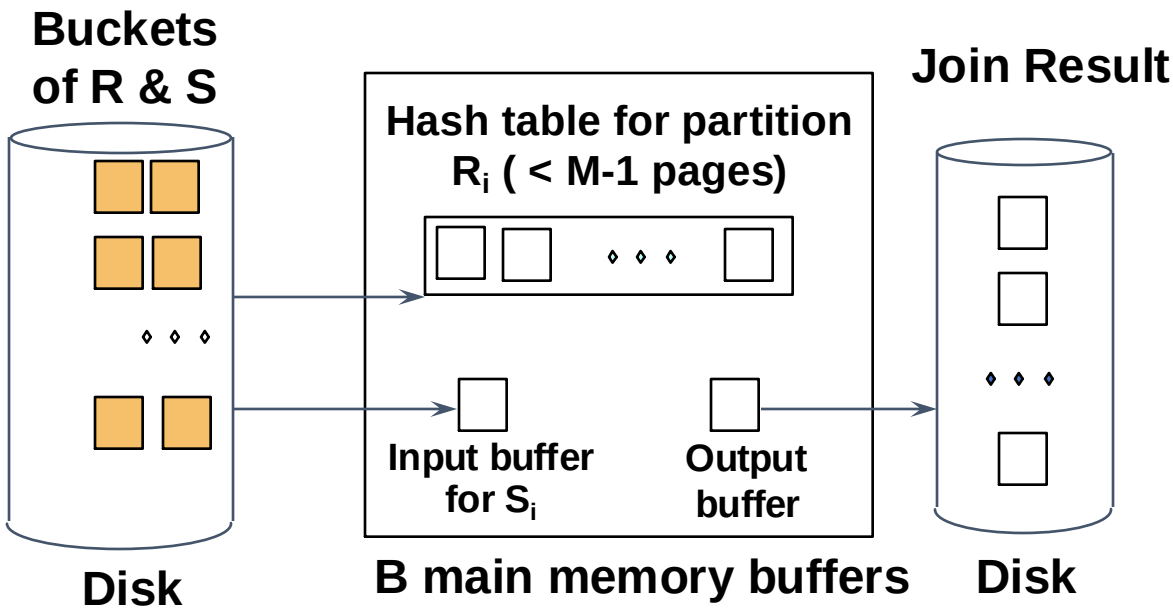
- Read one R-bucket; hash-partition it using h2 ($\neq$ h)
- Scan corresponding S bucket and join



Step 2: Join Buckets

$R \bowtie S$

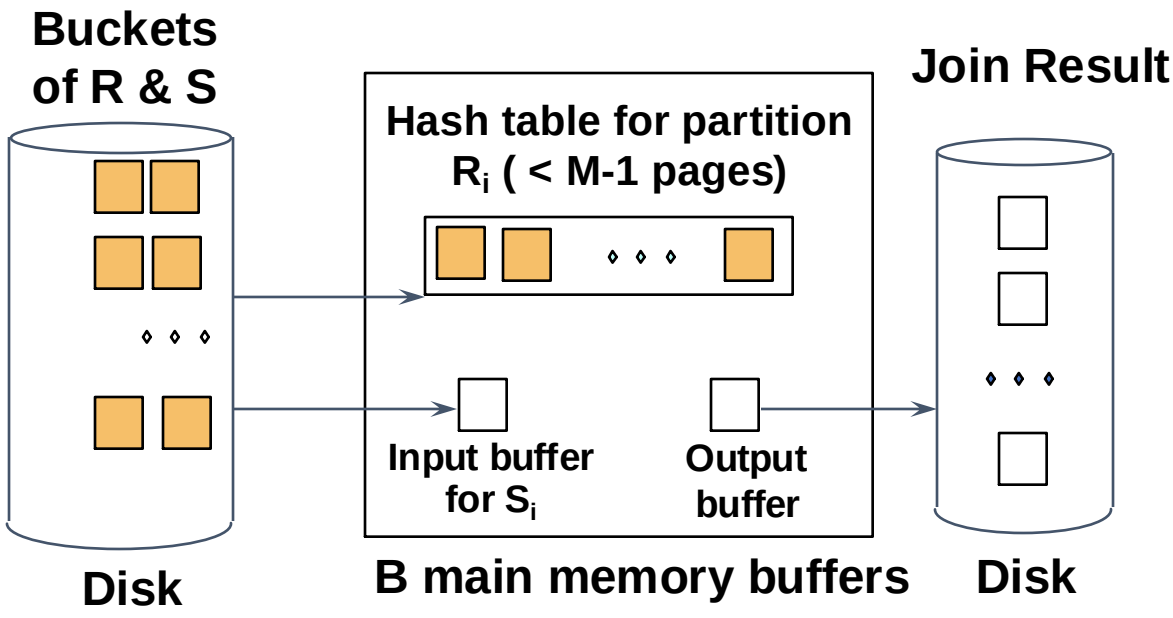
- Read the next R bucket



Step 2: Join Buckets

$R \bowtie S$

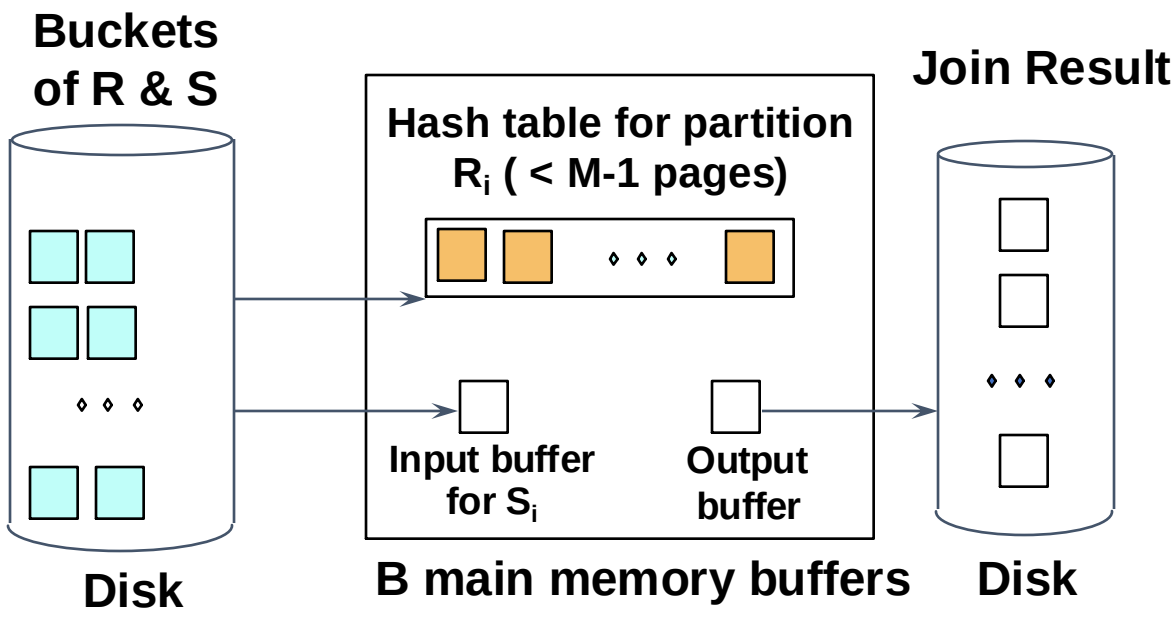
- Read the next R bucket



Step 2: Join Buckets

$R \bowtie S$

- Read the next R bucket
- Scan the matching S bucket



Partitioned Hash Join

- Cost: 3B(R) + 3B(S)
- Assumption: $\min(B(R), B(S)) \leq (M-1)^2$

Merge-Join

(if we have time…)

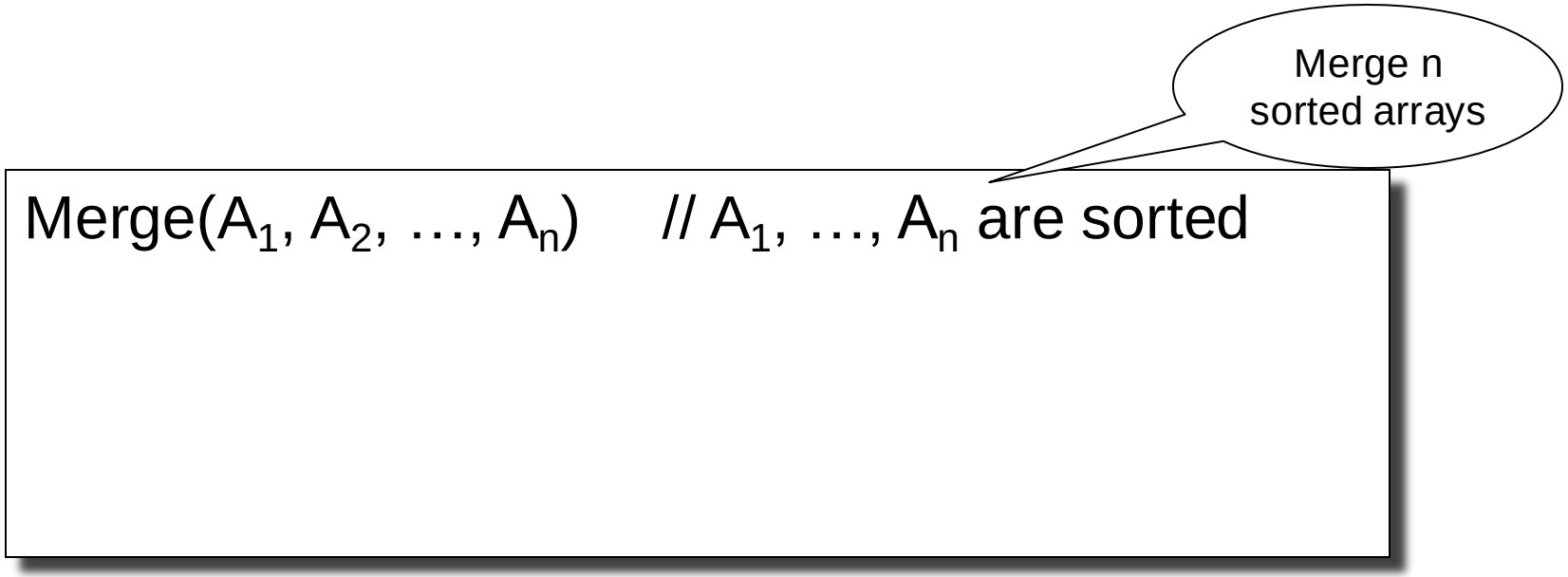
Merge-Sort

Merge-sort reads/writes sequentially

- Run lengths: 2, 4, 8, 16, …
- Need log(N) sequential reads and writes

$$\text{Cost} = 2 \log(N)\, B(R)$$

Merging n Arrays



Merging n Arrays

Merge(A_1, A_2, …, A_n) // A_1, …, A_n are sorted
  i_1 =0; …, i_n = 0; j=0
  while $i_1 \leq N$ or … or $i_1 \leq N$
    let $A_k[i_k] = \min(A_1[i_1], \ldots, A_n[i_n])$

Merging n Arrays

Merge(A_1, A_2, …, A_n) // A_1, …, A_n are sorted
  i_1 =0; …, i_n = 0; j=0
  while $i_1 \leq N$ or … or $i_1 \leq N$
    let $A_k[i_k] = \min(A_1[i_1], \ldots, A_n[i_n])$
    T[k++] = $A_k[i_k$++]

Merging n Arrays

Merge($A_1, A_2, \ldots, A_n$) // $A_1, \ldots, A_n$ are sorted
  $i_1 = 0; \ \ldots, i_n = 0; \ j=0$
  while $i_1 \leq N$ or … or $i_1 \leq N$
    let $A_k[i_k] = \min(A_1[i_1], \ldots, A_n[i_n])$
    $T[k{+}{+}] = A_k[i_k{+}{+}]$

Time = $O(|A_1| + |A_2| + \ldots |A_n|)$

Merging n Arrays

```
Merge(A_1, A_2, …, A_n)     // A_1, …, A_n are sorted
  i_1 =0;  …, i_n = 0;  j=0
  while i_1 ≤ N or … or i_1 ≤ N
    let  A_k[i_k] = min(A_1[i_1], …, A_n[i_n])
    T[k++] = A_k[i_k++]
```

$$\text{Time} = O(|A_1| + |A_2| + \ldots |A_n|)$$

Additional log n factor to find min using priority queue

Merging n Arrays

```
Merge(A_1, A_2, …, A_n)     // A_1, …, A_n are sorted
  i_1 =0;  …, i_n = 0;  j=0
  while i_1 ≤ N or … or i_1 ≤ N
    let  A_k[i_k] = min(A_1[i_1], …, A_n[i_n])
    T[k++] = A_k[i_k++]
```

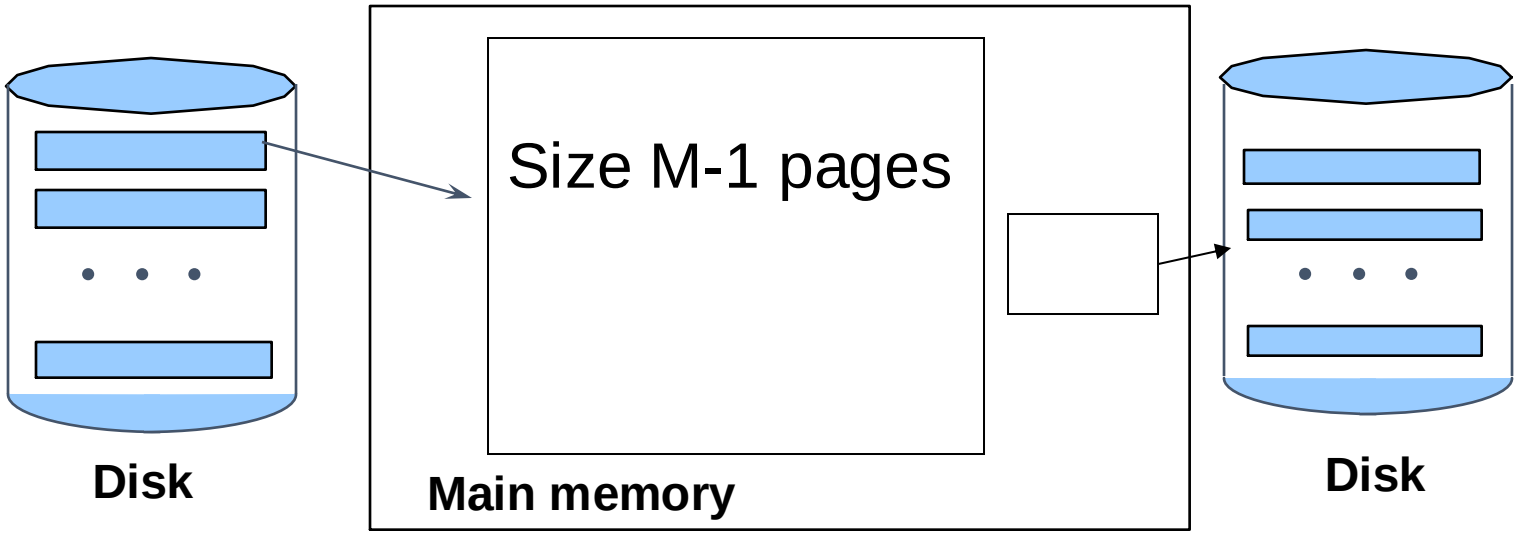
Important:
Need to read only one element from each array

$$\text{Time} = O(|A_1| + |A_2| + \ldots |A_n|)$$

Additional log n factor to find min using priority queue

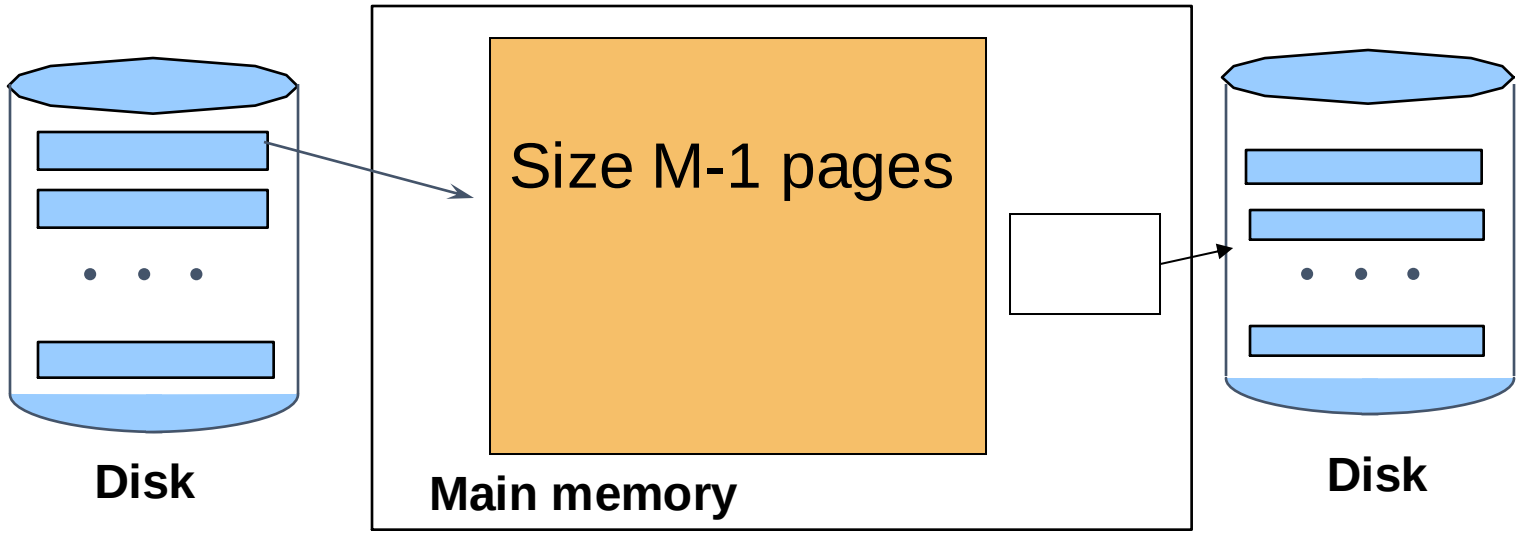
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



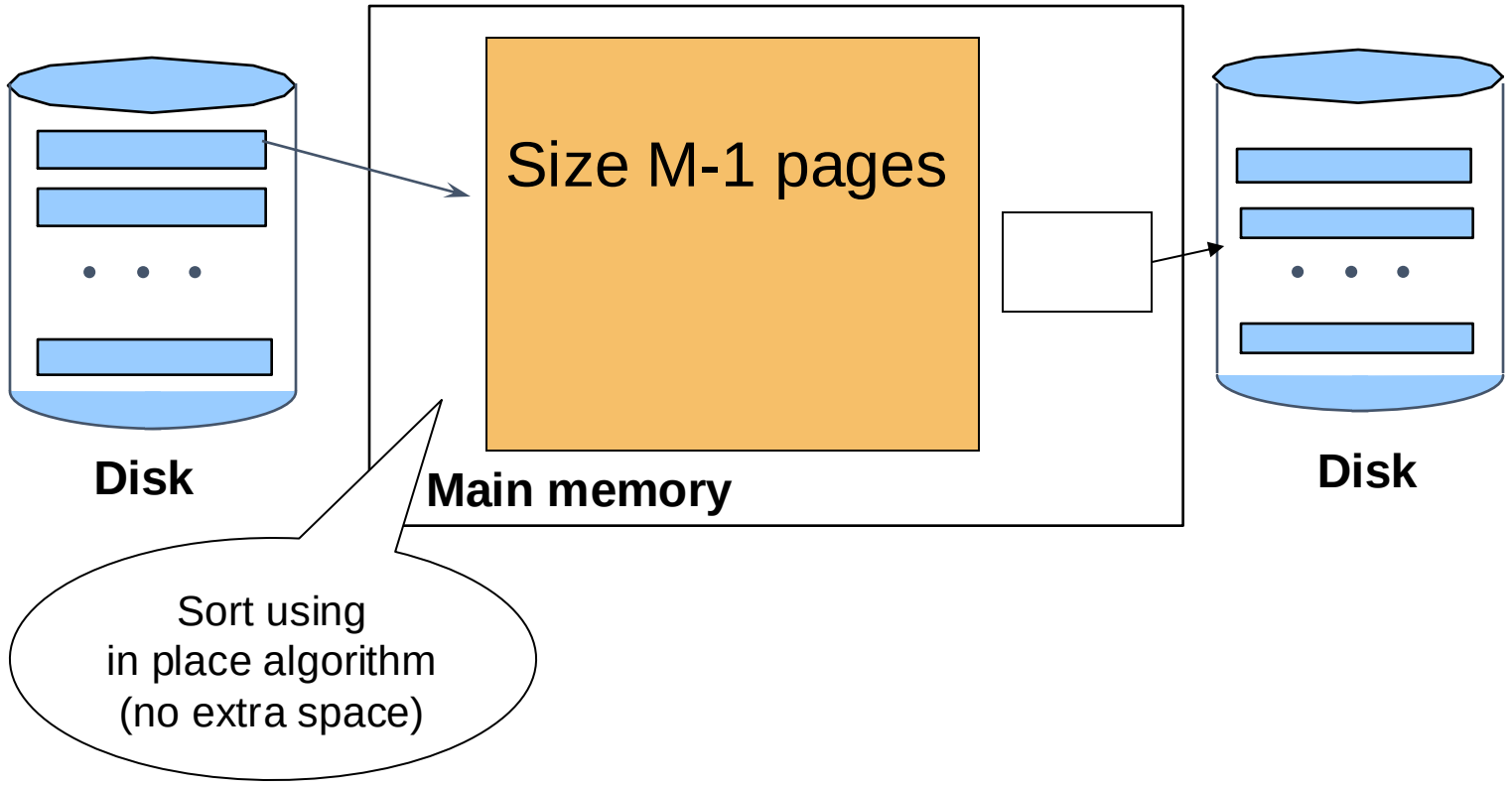
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



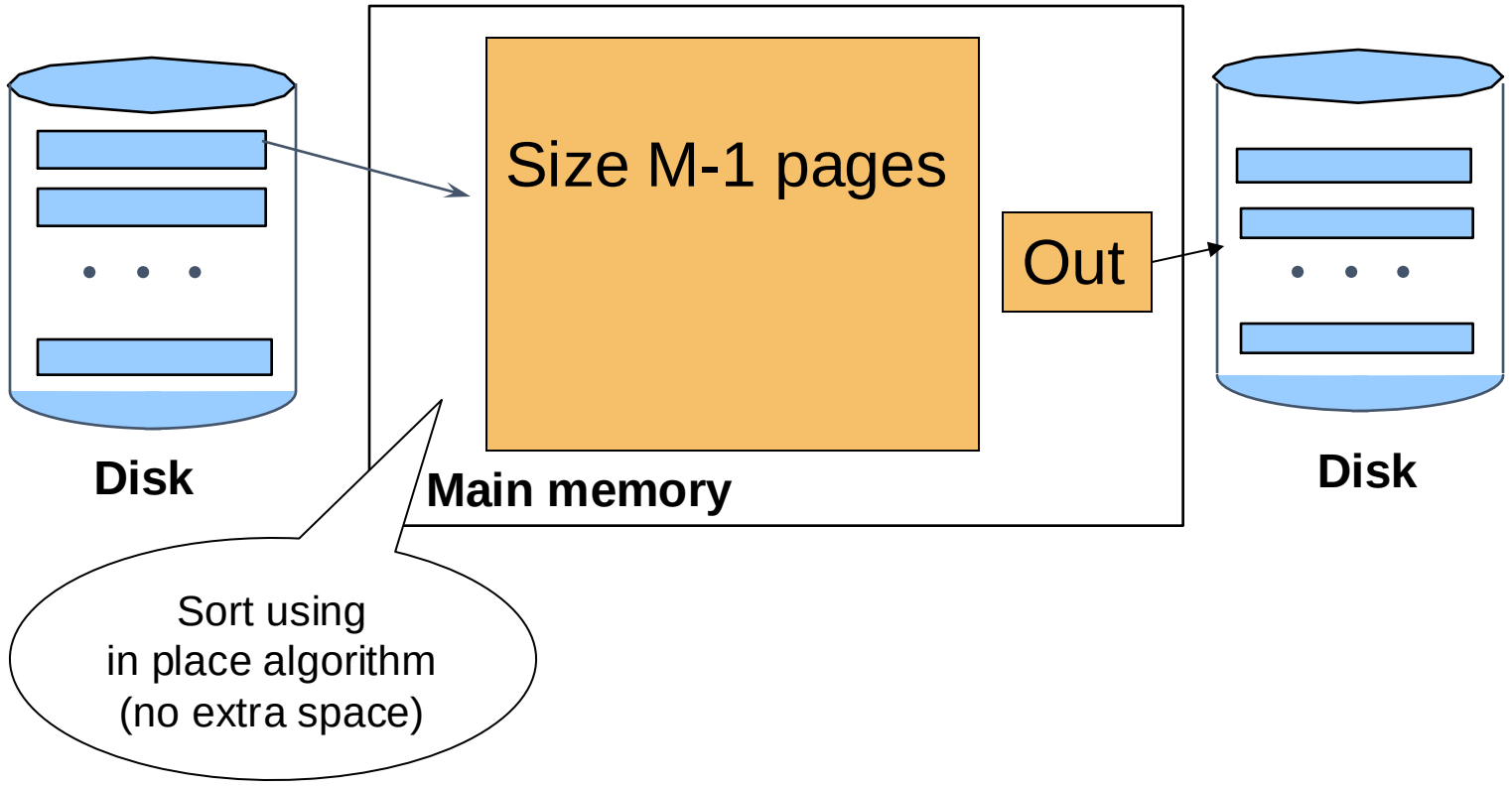
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



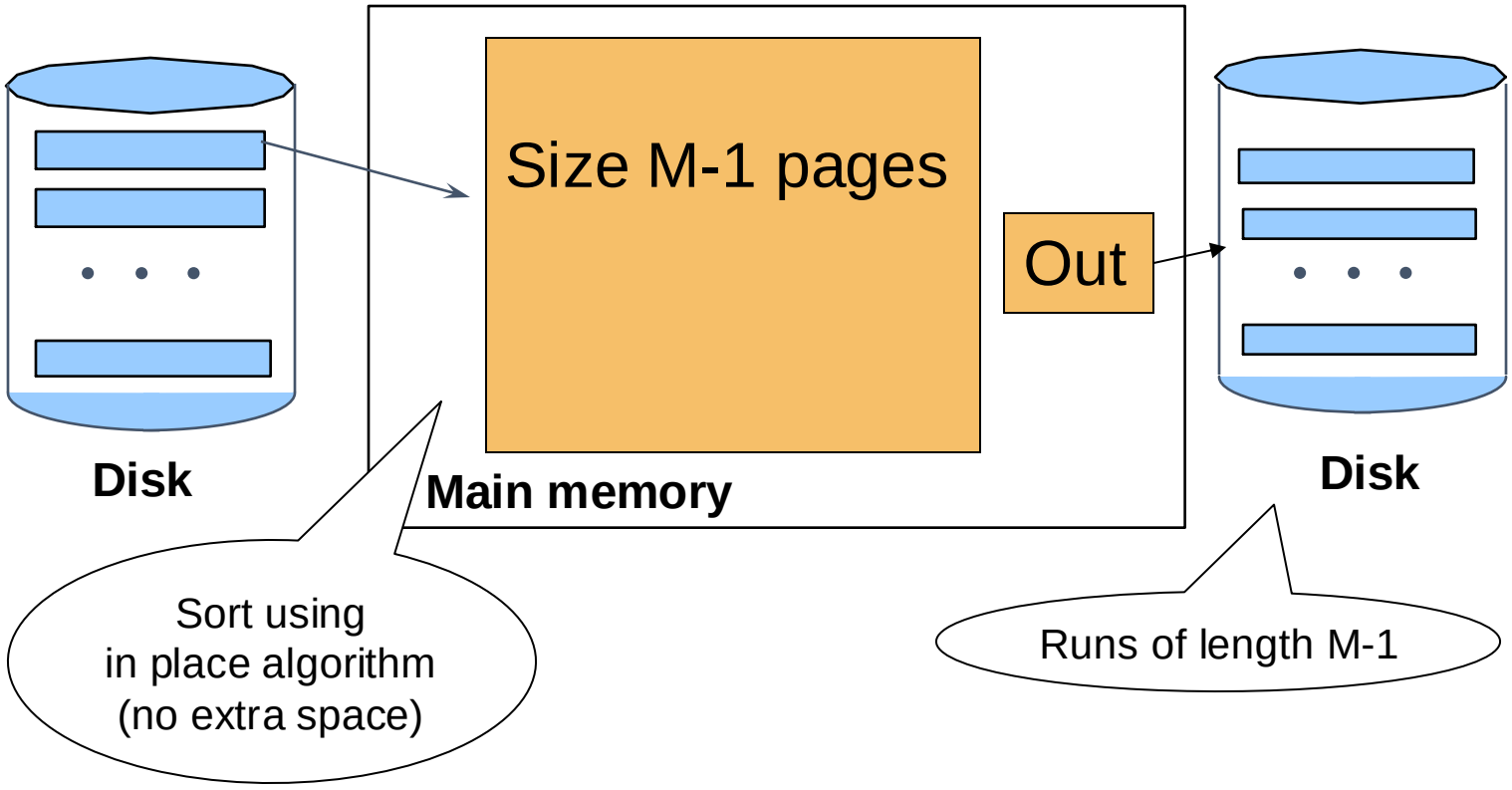
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



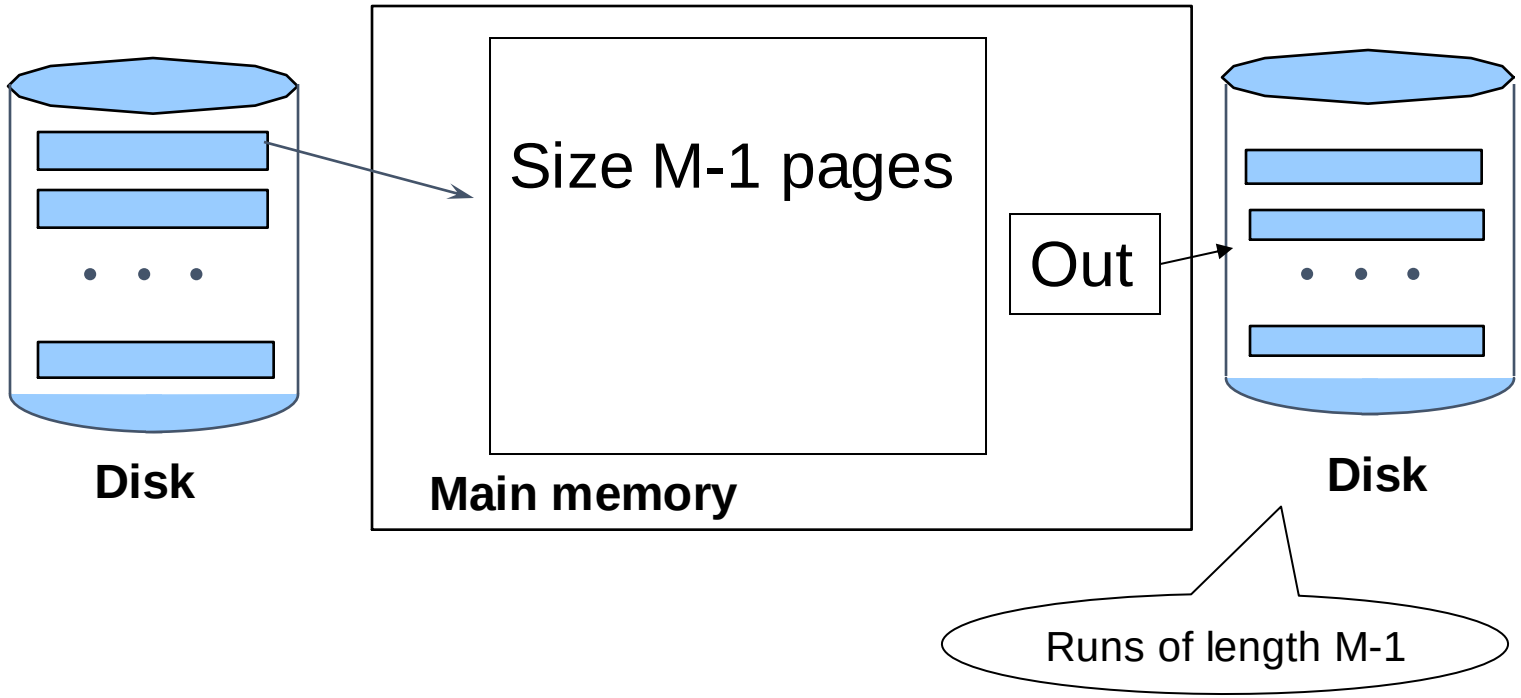
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



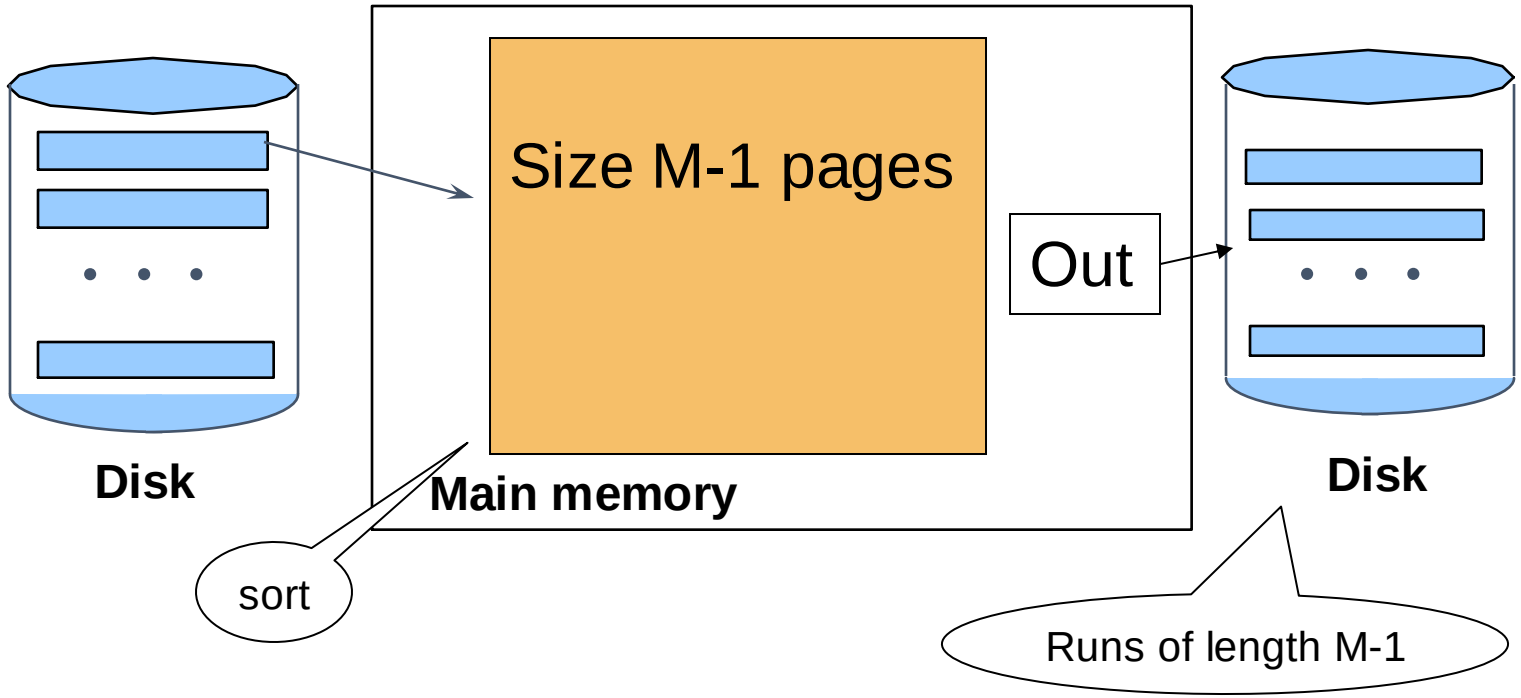
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



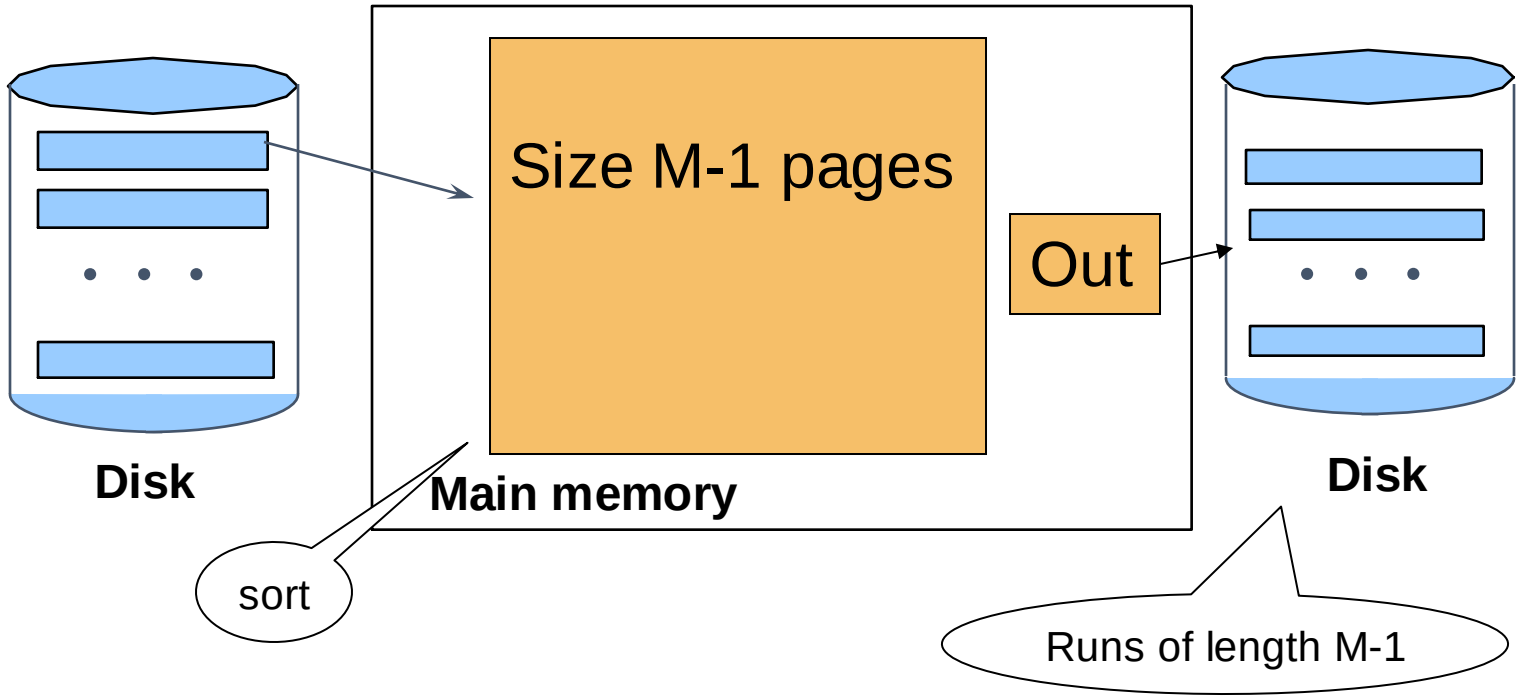
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



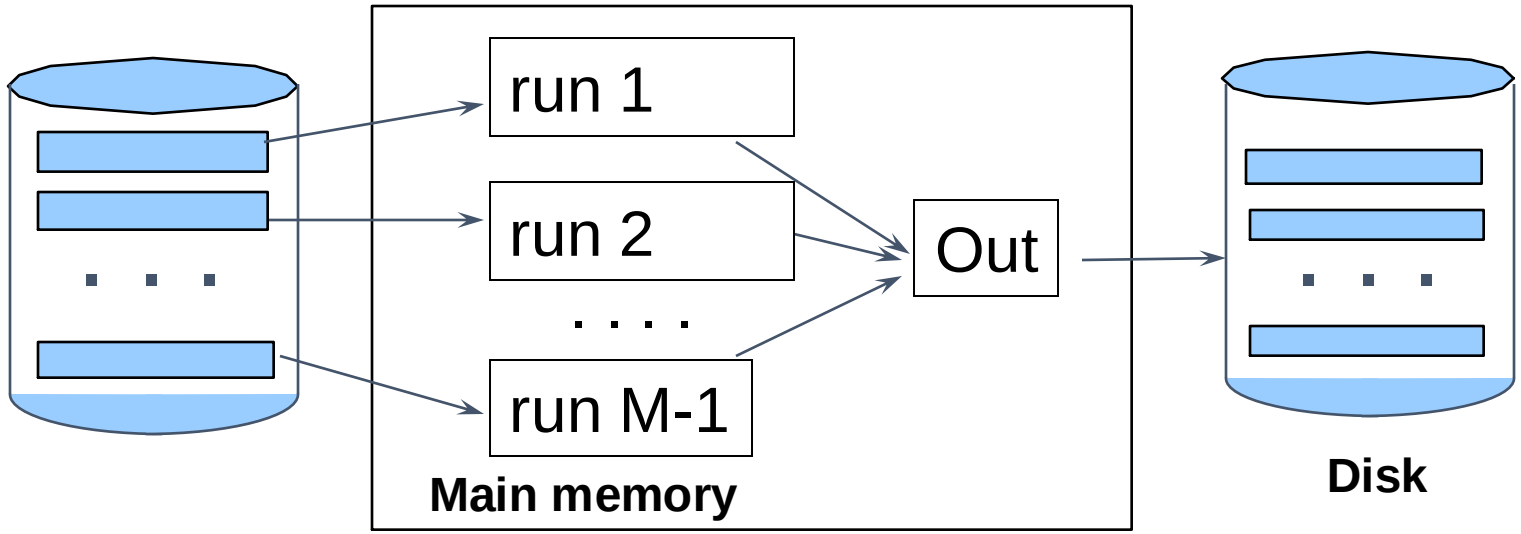
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort



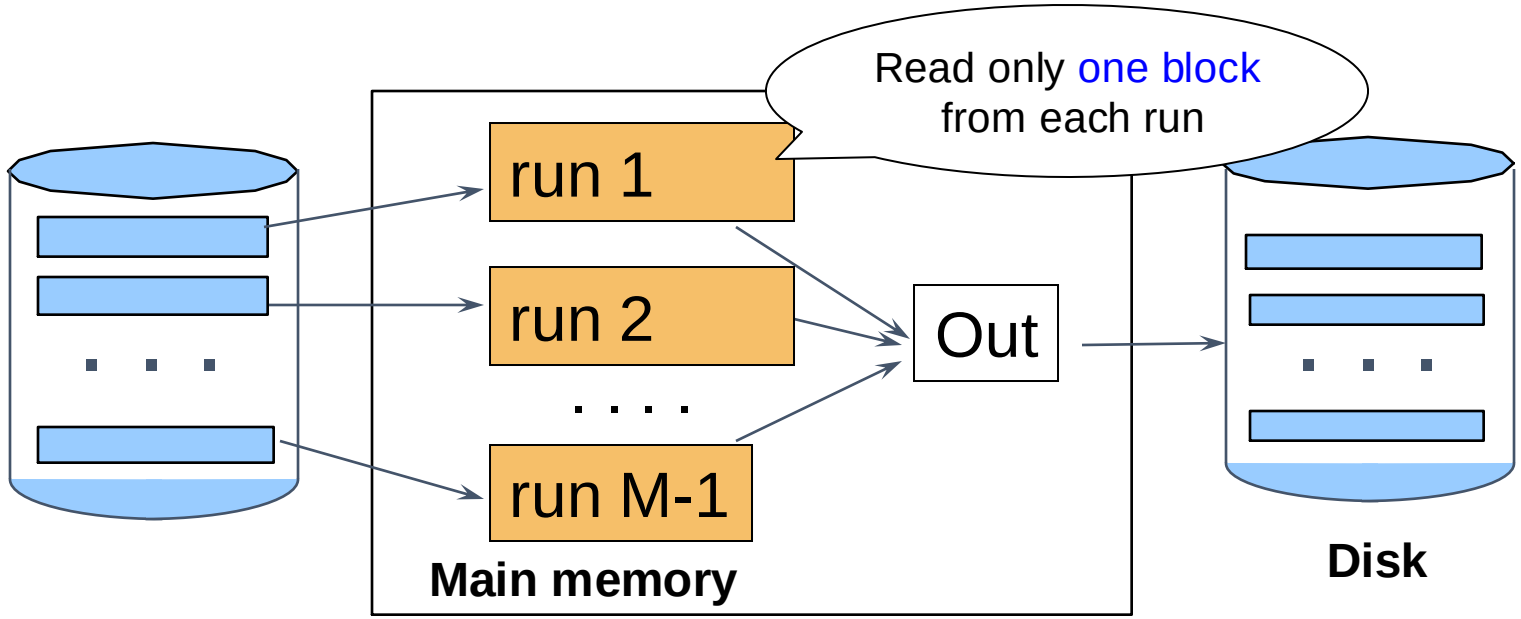
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



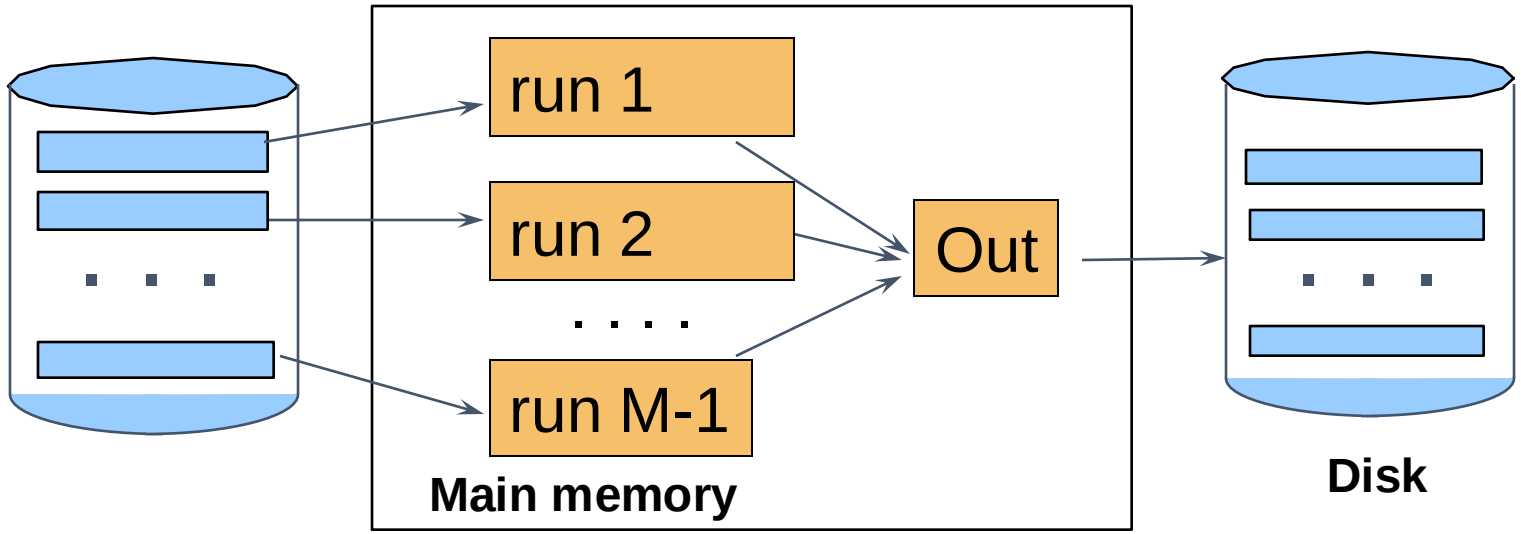
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



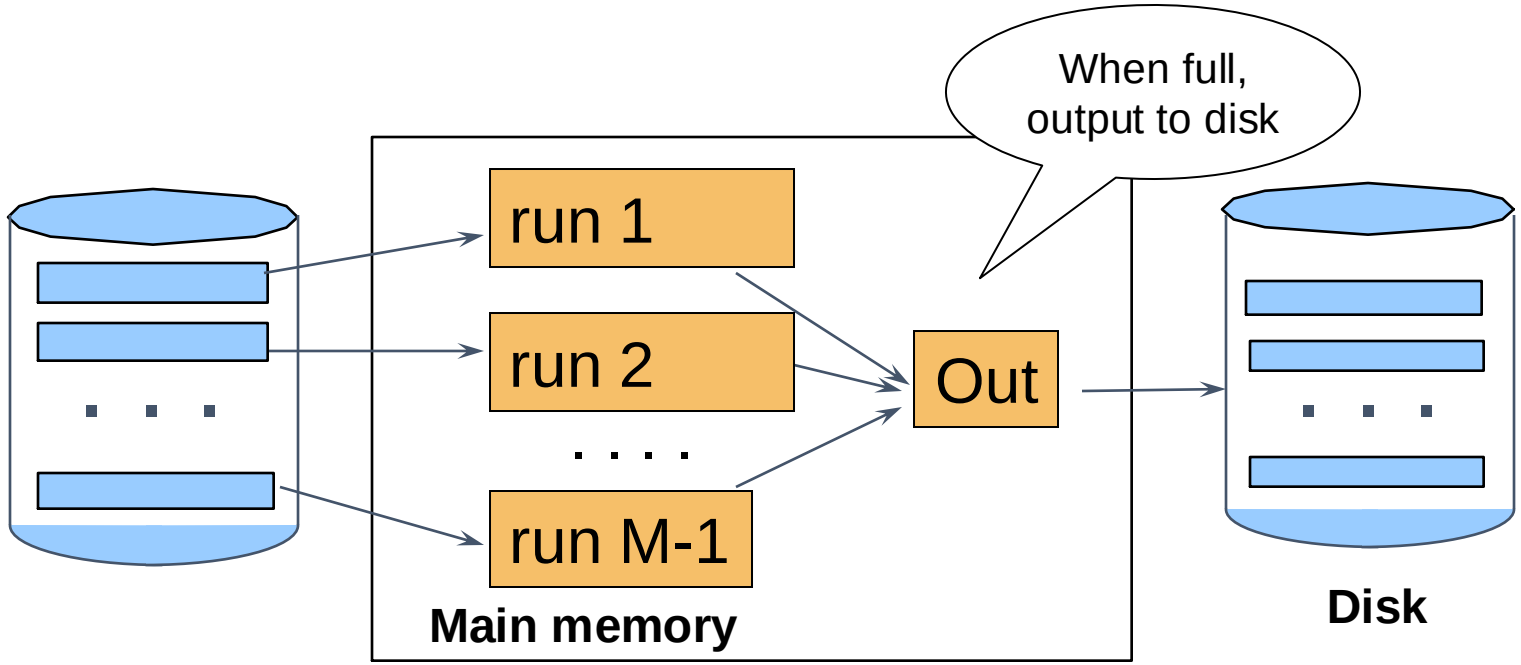
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



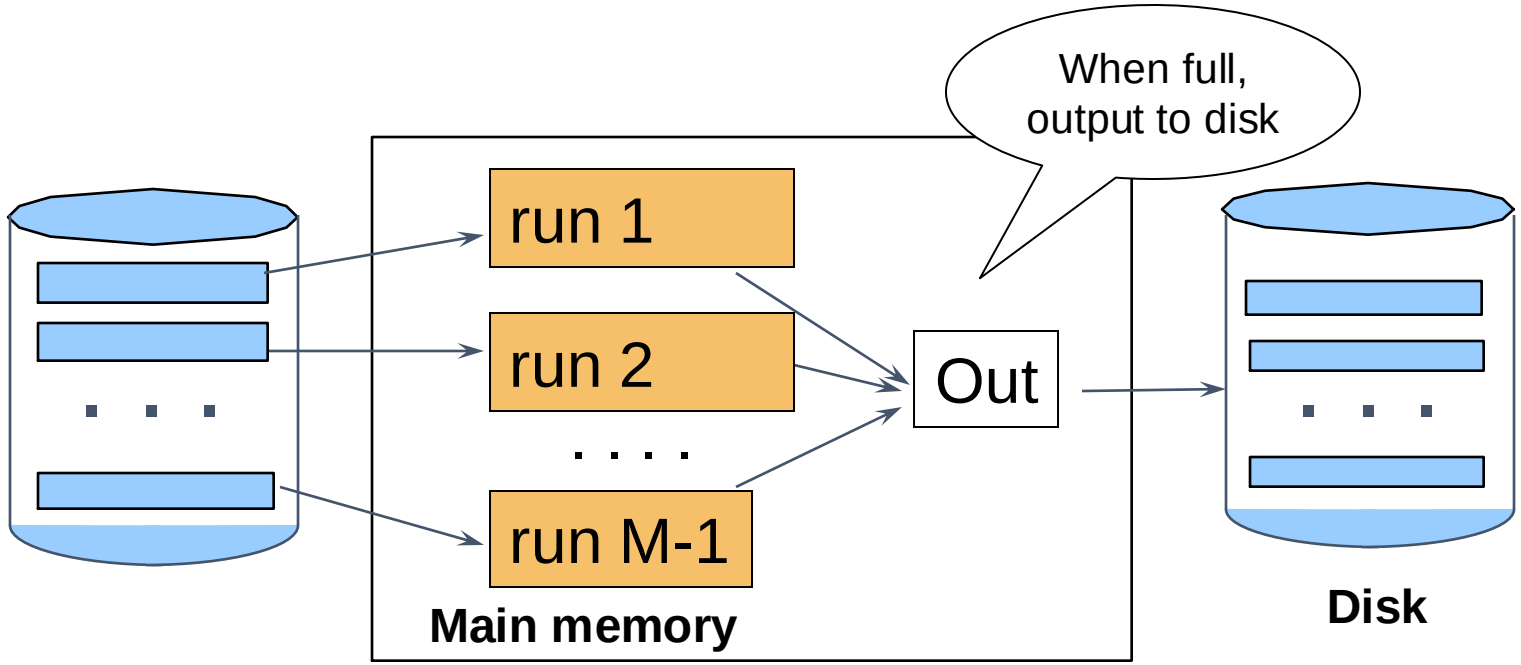
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



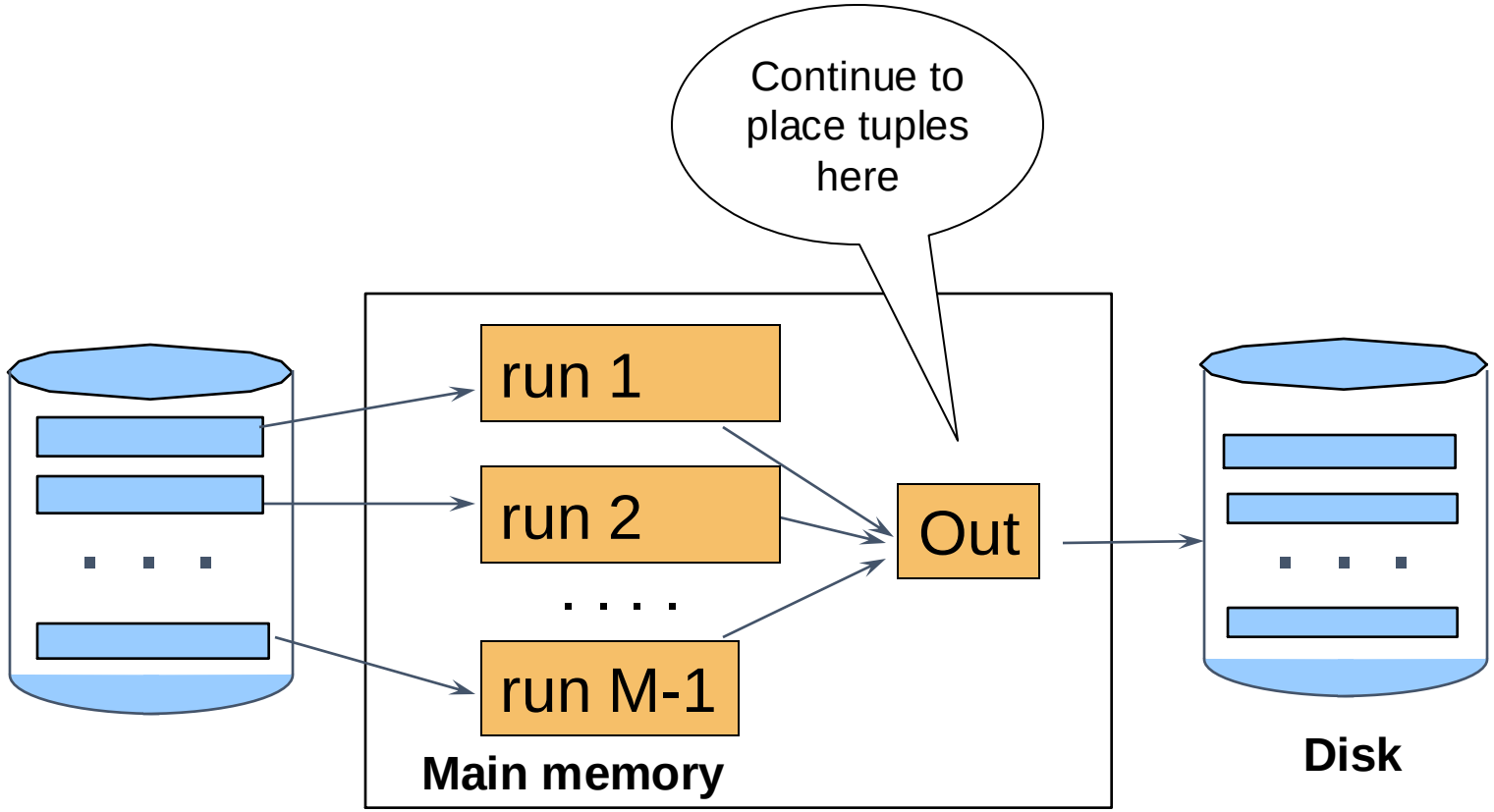
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



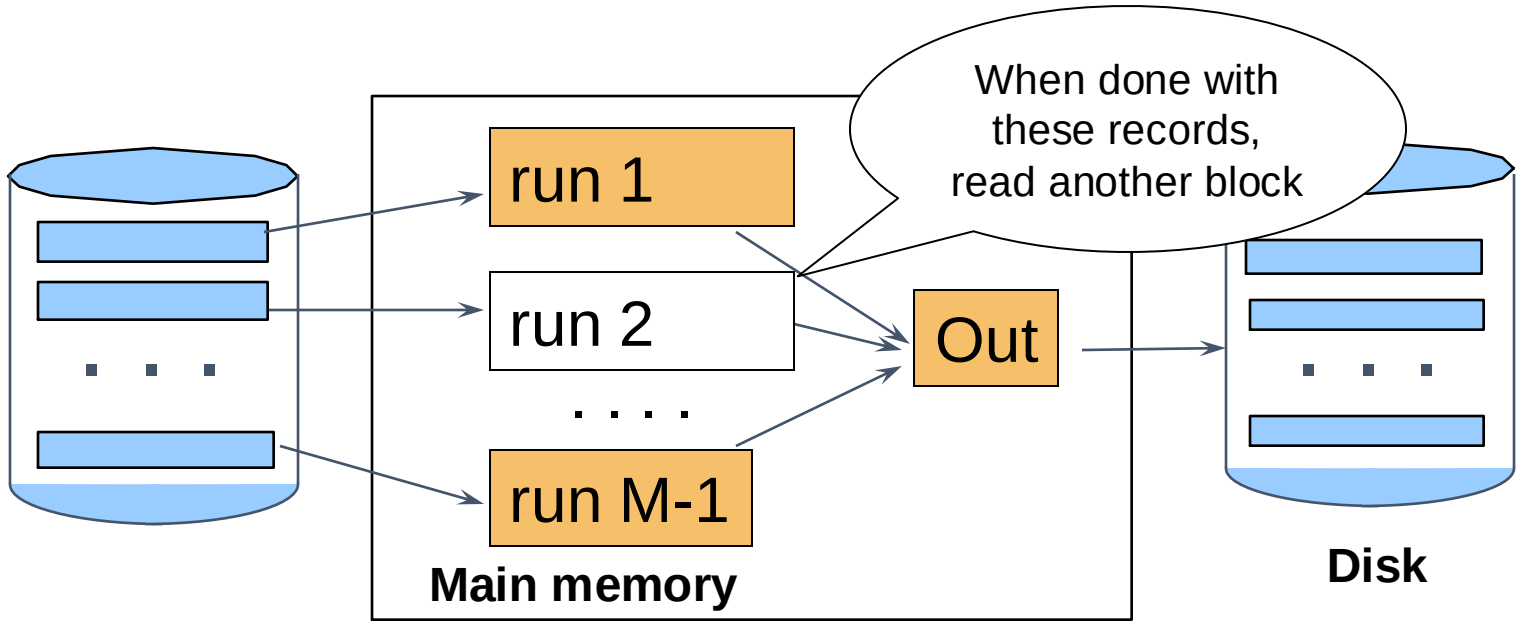
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



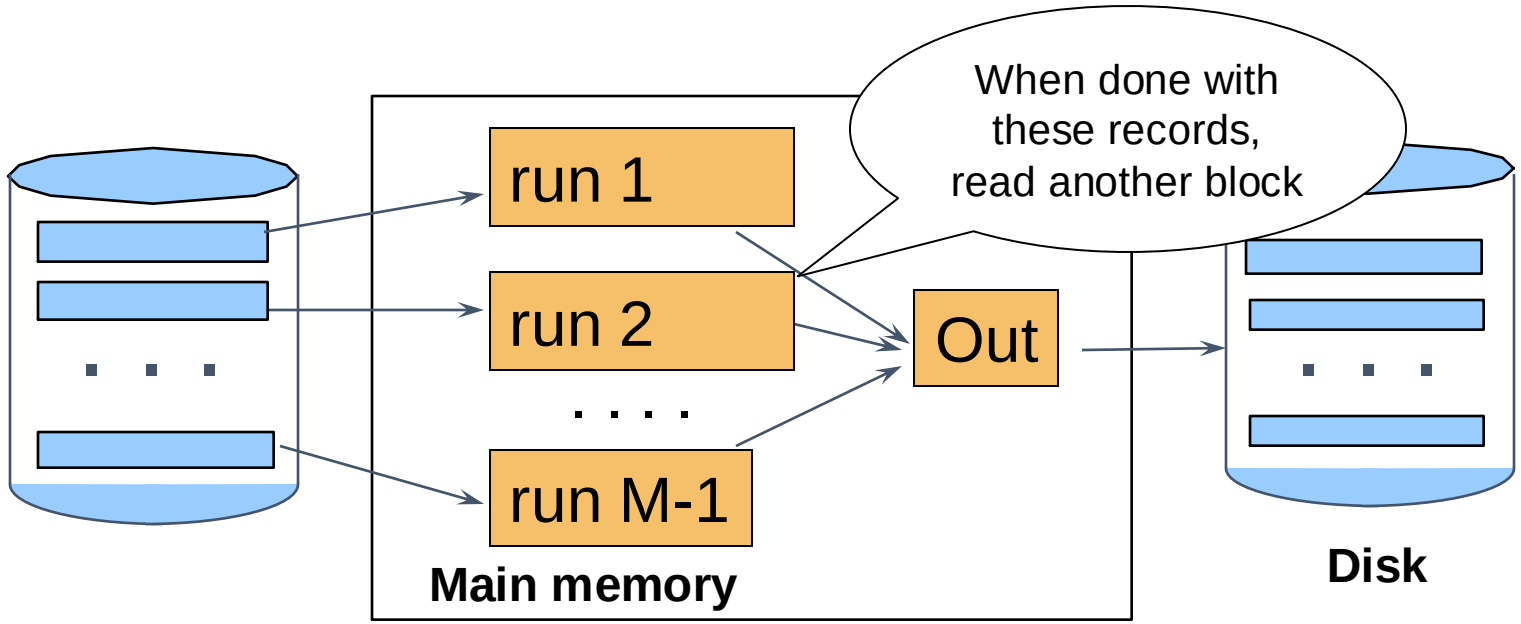
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



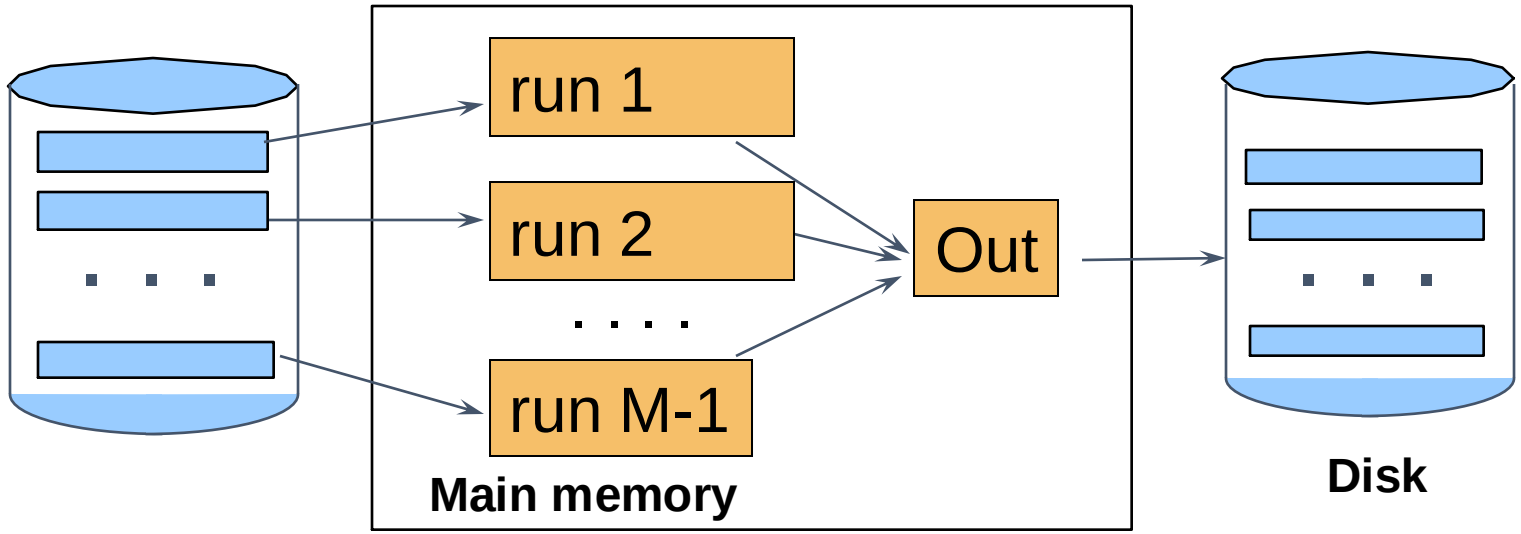
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



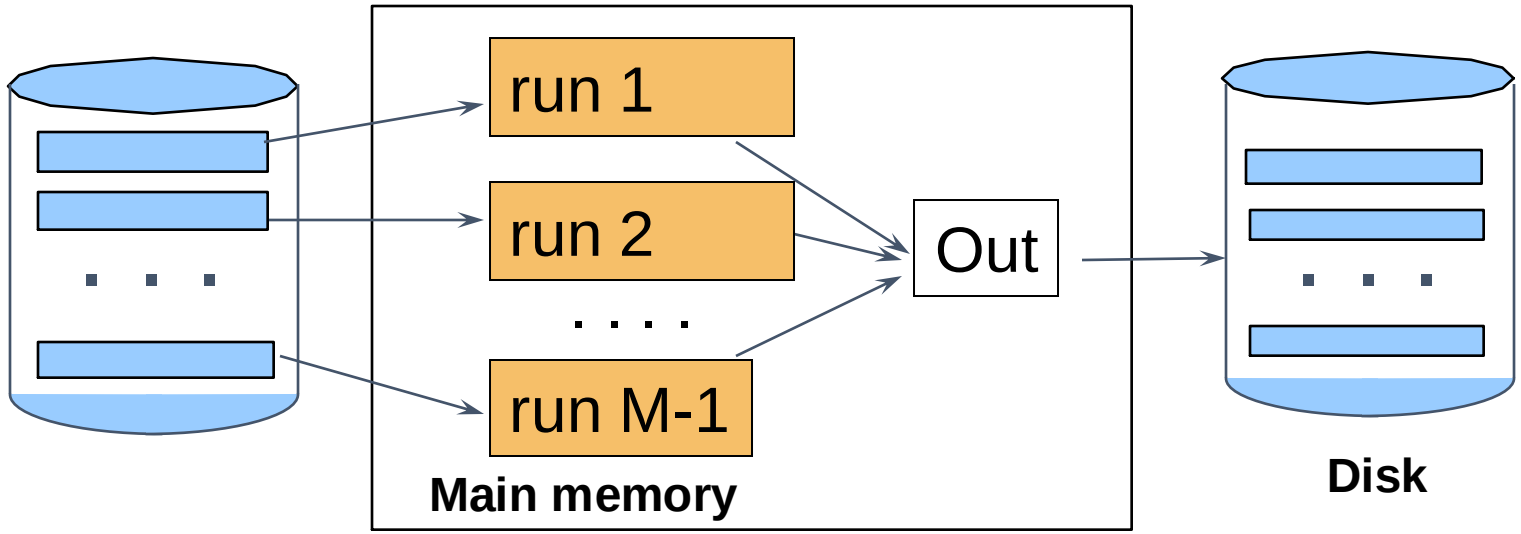
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



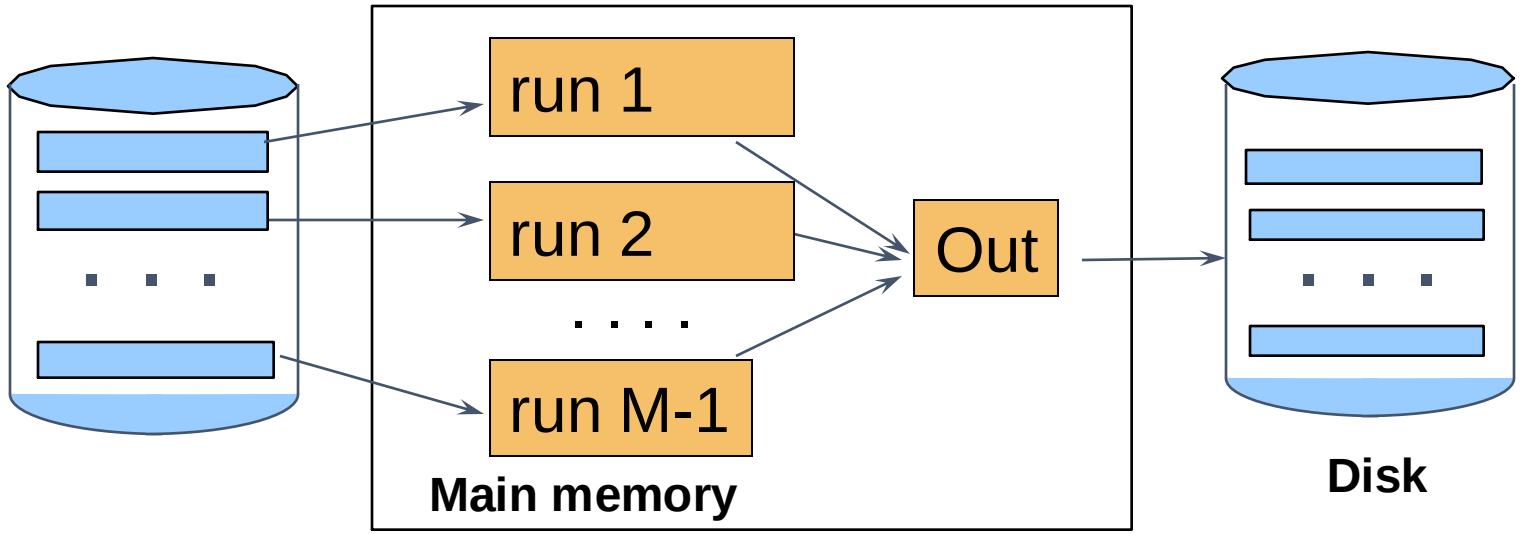
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



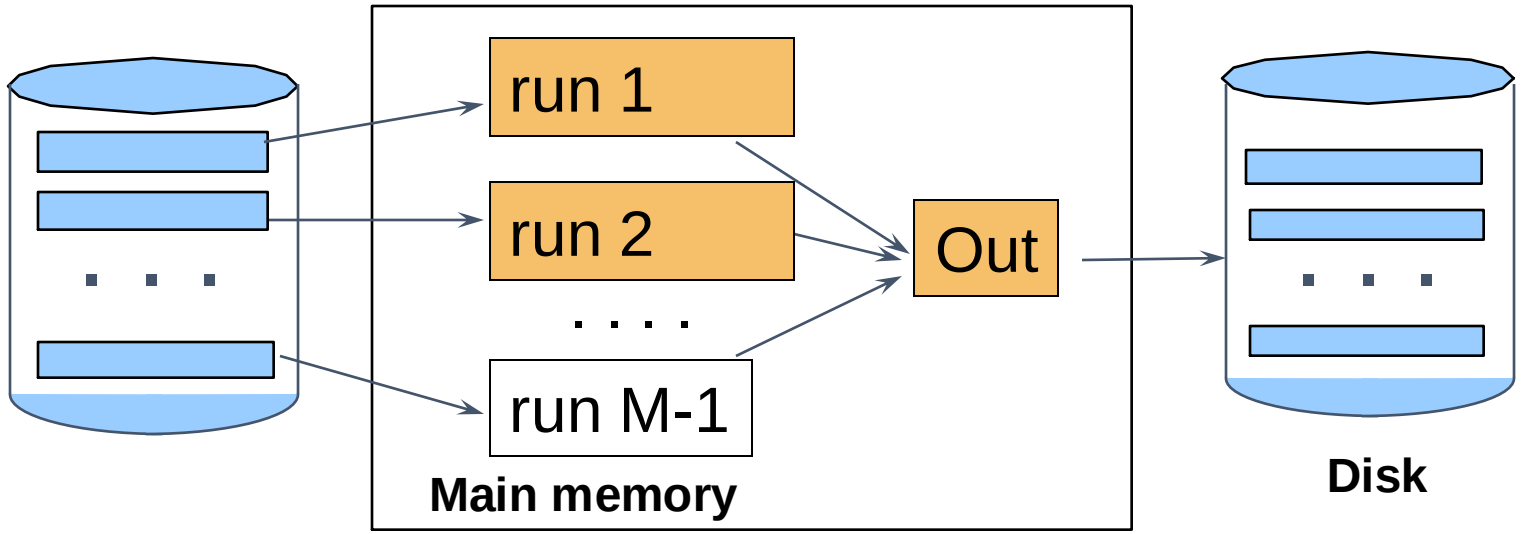
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



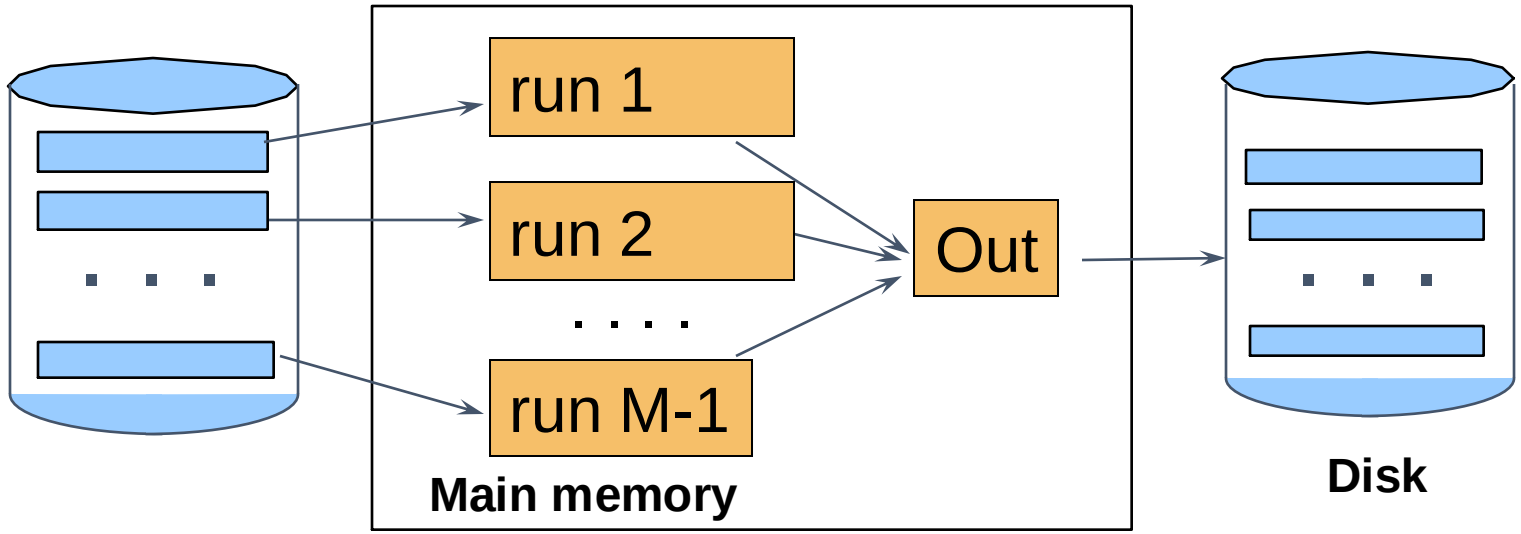
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



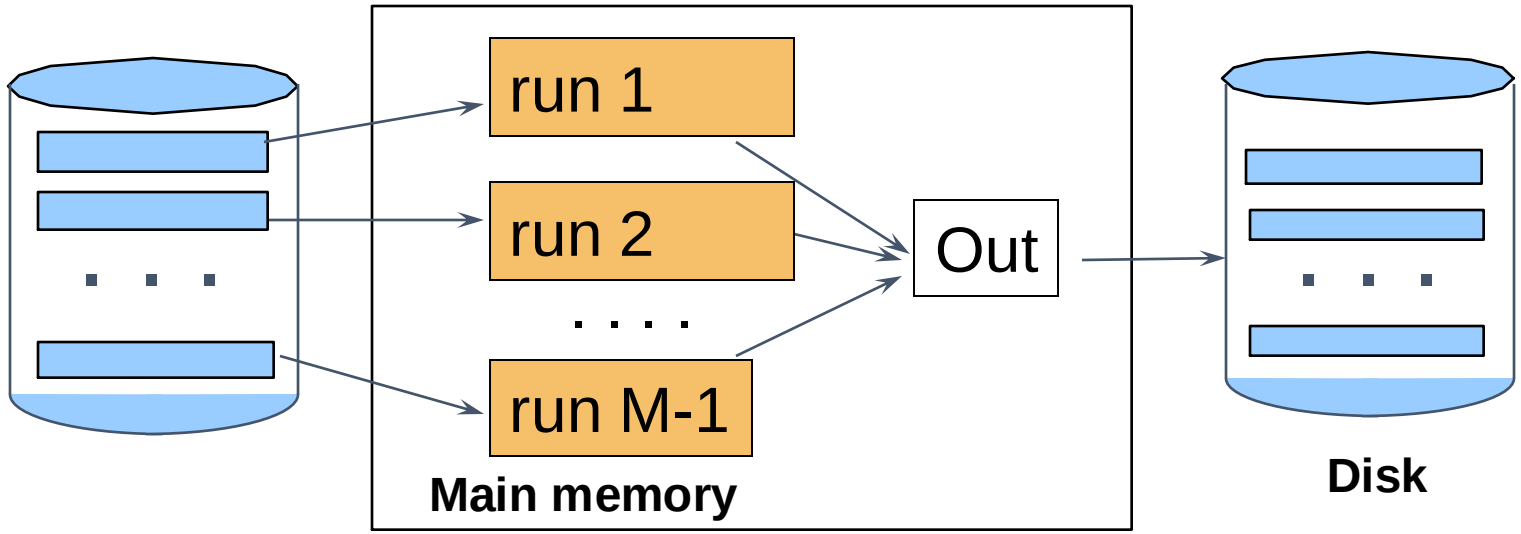
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



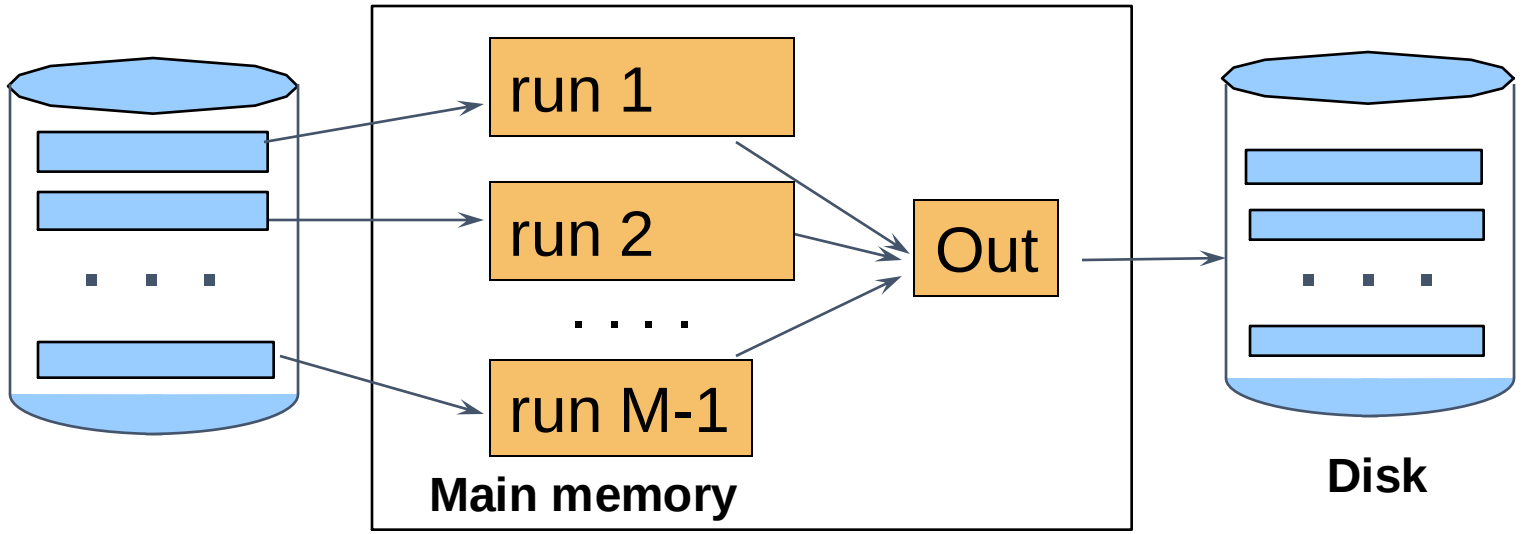
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



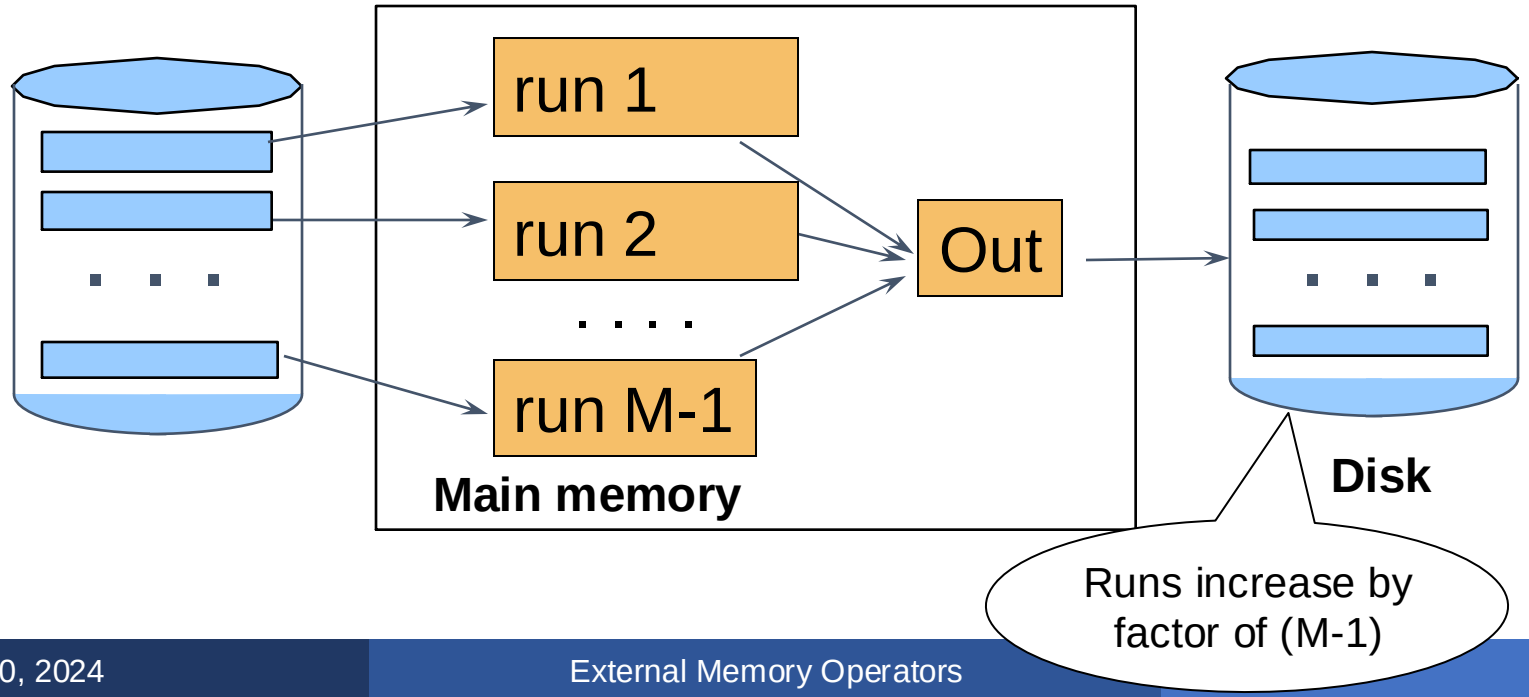
Merge-Sort: Steps 2, 3, …

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



Merge-Sort: Steps 2, 3, …

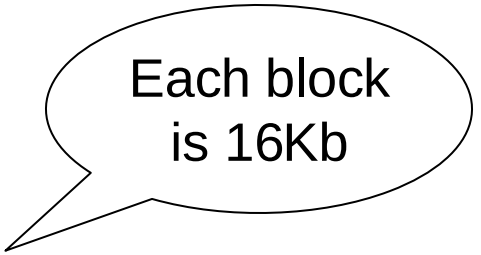
- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$,…



Merge-Sort: Discussion

Size of the runs increases fast

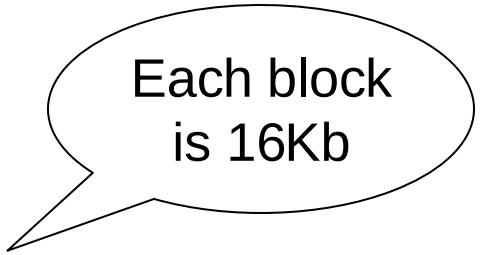
- Example: B=10001



Merge-Sort: Discussion

Size of the runs increases fast

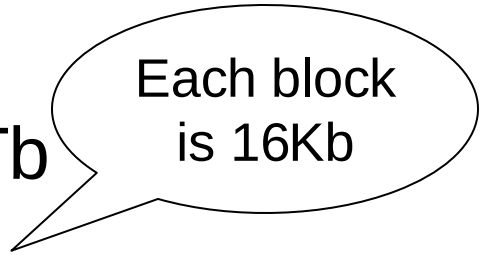
- Example: B=10001
- Step 1: run length 10000 = 160Mb



Merge-Sort: Discussion

Size of the runs increases fast

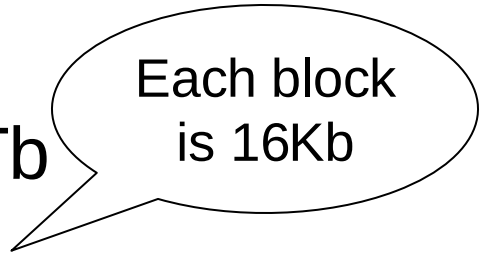
- Example: B=10001
- Step 1: run length 10000 = 160Mb
- Step 2: run length 100000000 = 1.6Tb
- …



Merge-Sort: Discussion

Size of the runs increases fast

- Example: B=10001
- Step 1: run length 10000 = 160Mb
- Step 2: run length 100000000 = 1.6Tb
- …



Usually, 2 steps suffice

Cost = 3B(R)

Finally: Merge-Join

$R \bowtie S$

Finally: Merge-Join

$R \bowtie S$

- Create runs of R → B(R)/(M-1) runs
- Create runs of S → B(S)/(M-1) runs

Finally: Merge-Join

$R \bowtie S$

- Create runs of R → B(R)/(M-1) runs
- Create runs of S → B(S)/(M-1) runs
- Use N-way merge to compute the join

Finally: Merge-Join

$R \bowtie S$

- Create runs of R → B(R)/(M-1) runs
- Create runs of S → B(S)/(M-1) runs
- Use N-way merge to compute the join

- Need $B(R)+B(S) \leq (M-1)^2$ why?

Cost = 3(B(R)+B(S))

Summary

- Basic algorithms:
  - Nested loop
  - Hash-based
  - Sort-based
- If larger than main memory, partition data by using temporary files on disk
- Usually one partitioning step suffices: create runs, or hash-partition